On the Expressiveness of Infinite Behavior and Name Scoping in Process Calculi

Pablo Giambiagi[1], Gerardo Schneider[2,3⋆], and Frank D. Valencia[3⋆⋆]

[1] KTH Royal Institute of Technology, IMIT, Electrum 229, 164 40 Kista, Sweden
{pgiamb@imit.kth.se}
[2] IRISA/CNRS, Campus de Beaulieu F-35042 Rennes, France
[3] Uppsala University, Dept. of Computer Systems, Box 337, 751 05 Uppsala, Sweden
{gerardos@it.uu.se; frankv@it.uu.se}

Abstract. In the literature there are several CCS-like process calculi differing in the constructs for the specification of infinite behavior and in the scoping rules for channel names. In this paper we study various representatives of these calculi based upon both their relative expressiveness and the decidability of *divergence*. We regard any two calculi as being *equally expressive* iff for every process in each calculus, there exists a *weakly bisimilar* process in the other.
By providing *weak bisimilarity* preserving mappings among the various variants, we show that in the context of *relabeling-free* and *finite summation* calculi: (1) CCS with *parameterless* (or *constant*) definitions is equally expressive to the variant with *parametric* definitions. (2) The CCS variant with *replication* is equally expressive to that with *recursive expressions* and *static* scoping. We also state that the divergence problem is undecidable for the calculi in (1) but decidable for those in (2). We obtain this from (un)decidability results by Busi, Gabbrielli and Zavattaro, and by showing the relevant mappings to be computable and to preserve divergence and its negation. From (1) and the well-known fact that parametric definitions can replace injective relabelings, we show that injective relabelings are redundant (i.e., derived) in CCS (which has constant definitions only).

1 Introduction

The study of concurrency is often conducted with the aid of process calculi. Undoubtedly CCS [9], a calculus for synchronous communication, remains a standard representative. In fact, many foundational ideas in the theory of concurrency have grown out of this calculus.

Nevertheless, there are several variants of CCS in the literature. This is reasonable as a variant may simplify the presentation of the calculus or be tailored to specific applications. Given two variants, a legitimate question is whether they are equally *expressive.* To answer this question one has to agree on what it means for one calculus to be as expressive as the other. A natural way of doing this in CCS is by comparing w.r.t. some standard process equivalence such as *(weak) bisimilarity*: If for every process P in one calculus there is a process Q in the other calculus such that Q is (weakly) bisimilar to

⋆ Work supported by European project ADVANCE, Contract No. IST–1999–29082.
⋆⋆ Work supported by European project PROFUNDIS.

P then we say that the second calculus is at least as expressive as the first one. Another legitimate question, given a variant, is whether some fundamental property such as *divergence* (i.e., the existence of divergent computations) becomes simpler or harder to analyze.

In this paper, we study both the relative expressiveness w.r.t. weak bisimilarity and the decidability of divergence for various CCS-like calculi. We shall focus upon two sources of variation found in the CCS literature: The constructs used to express infinite behavior and the way in which scoping of channel (port) names is dealt with. As for the constructs for finite behavior, in all the calculi we confine our attention to prefix, finite sums, restriction, and parallel composition. The calculi here studied can be described as follows:

- CCS_{k}: Infinite behavior is given by a *finite* set of *constant* (i.e., parameterless) definitions of the form $A \stackrel{\text{def}}{=} P$. The calculus is essentially CCS [9] with neither relabelings nor infinite summations.
- CCS_{p}: Like CCS_{k} but using *parametric definitions* of the form $A(x_1, \ldots, x_n) \stackrel{\text{def}}{=} P$. The calculus is the variant in [10], Part I.
- $\text{CCS}_{!}$: Infinite behavior given by *replication* of the form $!P$. This variant is presented in [3].
- CCS_{μ}: Infinite behavior given by *recursive expressions* of the form $\mu X.P$ as in [9]. However, we adopt *static scoping* of channel names in the sense discussed in [5].

In particular, we show that (1) CCS_{k} is exactly as expressive as CCS_{p} while (2) CCS_{μ} is exactly as expressive as $\text{CCS}_{!}$. We use recent work by Busi et al. [3] to also state that (3) the divergence problem is undecidable for the calculi in (1) but decidable for those in (2). The results (1-3) are summarized in Figure 1.

Also, as a consequence of (1), we prove that (4) injective relabelings, from the expressiveness point of view, are redundant operators in CCS. More precisely, the behavior of any CCS process involving relabelings (all of them being injective) can be expressed up to strong bisimilarity by a CCS_{k} process. Furthermore, we also illustrate that CCS_{k} exhibits *dynamic scoping* of channel names and that it does not satisfy α-conversion. 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.

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 an also *finite* set of parameterless definitions using neither infinite summations nor relabelings. 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 [9] which shows that, in the context of value-passing CCS, a parametric definition can be encoded using an *infinite* set of constant definitions and infinite sums.

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 CCS_{p}; an unfolding of 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.

The interesting point about (4) is that injective relabelings are perhaps the most used kind of relabelings (e.g., injective relabelings are used in [9] to define linking operators,

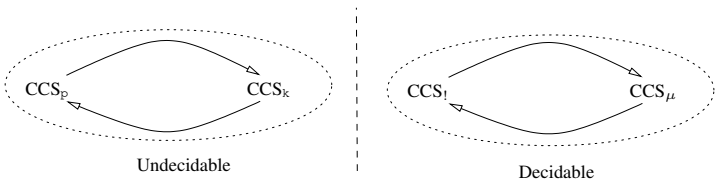

Fig. 1. Classification of CCS variants. An arrow from X to Y indicates that Y is at least as expressive as X. (Un)decidability is understood w.r.t. the existence of divergent computations

buffers, counters and stacks). In fact, [9] points out that the CCS laws for equational reasoning with injective relabelings as side conditions can usually be applied as one mostly works with this kind of relabeling. In the context of SCCS, another CCS variant where interaction is synchronous, *idempotent relabelings* are known to be redundant [8]. In fact, under some natural assumptions, the same holds for general relabelings in SCCS.

Another noteworthy aspect of our results is the qualitative distinction between static and dynamic name scoping for the calculi under consideration. Static scoping renders the calculus decidable (w.r.t. the divergence problem) and as expressive as that with replication. In contrast, dynamic scoping renders the calculus undecidable and as expressive as that with parametric definitions. This is interesting, since as we shall see, 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 [5], an extension of CCS whose ideas lead to the design of the π-calculus [10].

It should be noticed that preservation of divergence is not a requirement for equality of expressiveness; *weak bisimilarity does not preserve divergence*. Hence, although the results in [3] prove that divergence is decidable for CCS$_{!}$ (and undecidable for CCS$_{\mathtt{p}}$), it does not follow directly from the arrows in Figure 1 that it is also decidable for CCS$_{\mu}$.

Finally, it is worth pointing out that, as exposed in [7], decidability of divergence does not imply lack of *Turing* expressiveness. In fact the authors in [2] show that CCS$_{!}$ is Turing-complete. But this does not imply that CCS$_{!}$ is equally expressive to CCS$_{\mathtt{p}}$ either; the notions of expressiveness used in concurrency theory may not coincide with those in computability. For example, [11] shows that under some reasonable assumptions the asynchronous version of the π-calculus, which can certainly encode Turing Machines, is strictly less expressive than the synchronous one.

Overall, the general contribution of this paper is to provide and clarify some qualitative and semantics distinctions among various CCS variants.

2 CCS-like Calculi

We shall classify CCS-like calculi that differ in their way of specifying infinite behavior and name scope. Let us begin with their common finite fragment.

In CCS, processes can perform actions or synchronize on them. These actions can be either offering port *names* for communication, or the so-called *silent* action τ. We presuppose a countable set $\mathcal{N}$ of port *names*, ranged over by $a, b, x, y \ldots$ and their

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

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

Table 1. An operational semantics for finite processes

primed versions. We then introduce a set of *co-names* $\overline{\mathcal{N}} = \{\overline{a} \mid a \in \mathcal{N}\}$ disjoint from $\mathcal{N}$. The set of *labels*, ranged over by l and l', is $\mathcal{L} = \mathcal{N} \cup \overline{\mathcal{N}}$. The set of *actions* Act, ranged over by α and β, extends $\mathcal{L}$ with a new symbol τ. Actions a and $\overline{a}$ are thought of as *complementary*, so we decree that $\overline{\overline{a}} = a$. We also decree that $\overline{\tau} = \tau$.

The processes specifying finite behavior are given by:

$$P, Q \ldots ::= \textstyle\sum_{i\in I} \alpha_i.P_i \mid P\backslash a \mid P \parallel Q \tag{1}$$

Intuitively $\sum_{i\in I} \alpha_i.P_i$, where I is a finite set of indexes, represents a process able to perform one–but only one–of its α_i's actions and then behave as the corresponding P_i. We write the summation as $\mathbf{0}$ if $|I| = 0$, and drop the "$\sum_{i\in I}$" if $|I| = 1$. The restriction $P\backslash a$ behaves as P except that it can offer neither a nor $\overline{a}$ to its environment. The names a and $\overline{a}$ in P are said to be *bound* in $P\backslash a$. The *bound names* of P, $bn(P)$, are those with a bound occurrence in P, and the *free names* of P, $fn(P)$, are those with a not bound occurrence in P. Finally, $P \parallel Q$ represents parallelism; either P or Q may perform an action, or they can also synchronize when performing complementary actions.

The above description is made precise by the operational semantics in Table 1. A transition $P \xrightarrow{\alpha} Q$ says that P can perform α and evolve into Q.

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

2.1 Parametric Definitions: $\text{CCS}_{\mathtt{p}}$

A common way of specifying infinite behavior is by using parametric definitions [10]. 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* $A(x_1, \ldots, x_n) \stackrel{\mathtt{def}}{=} P_A$ where the x_i's are pairwise distinct, and the intuition is that $A(y_1, \ldots, y_n)$ behaves as its *body* P_A with each y_i replacing the *formal parameter* x_i. We denote by $\mathcal{D}$ the set of all definitions. We often use the notation $\vec{x}$ as an abbreviation of $x_1, x_2, \ldots, x_n$.

Convention 1 (Finitary $\mathcal{D}$) *Similar to [13], we shall require any process to depend only on finitely many definitions. Below we formalize this requirement.*

Given $A(\vec{x}) \stackrel{\texttt{def}}{=} P_A$ and $B(\vec{y}) \stackrel{\texttt{def}}{=} P_B$ in $\mathcal{D}$, we say that A (directly) *depends* on B, written $A \rightsquigarrow B$, if there is an invocation $B(\vec{z})$ in P_A. The above requirement can be then formalized by requiring the strict order induced by $\rightsquigarrow^*$ (the reflexive and transitive closure of $\rightsquigarrow$)[1] to be well-founded. We also stipulate the following requirement.

Convention 2 *For each $A(x_1, \ldots, x_n) \stackrel{\texttt{def}}{=} P_A$, we require $fn(P_A) \subseteq \{x_1, \ldots, x_n\}$.*

We shall use CCS_p to denote the calculus with parametric definitions with the above syntactic restrictions. The rules for CCS_p are those in Table 1 plus the rule:

$$\text{CALL}\ \frac{P_A[y_1, \ldots, y_n/x_1, \ldots, x_n] \stackrel{\alpha}{\longrightarrow} P'}{A(y_1, \ldots, y_n) \stackrel{\alpha}{\longrightarrow} P'}\ \text{if}\ A(x_1, \ldots, x_n) \stackrel{\texttt{def}}{=} P_A \qquad (3)$$

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., performing *name α-conversion*, wherever needed to avoid capture. It follows from [10] that in CCS_p we can identify process expressions obtained by renaming bound names (so $P\backslash a$ is the same as $P[b/a]\backslash b$). We then say that CCS_p satisfies *name α-equivalence* .

2.2 Constant Definitions: CCS_k

We now consider the alternative for infinite behavior given in CCS [9]. We refer to identifiers with arity zero and their corresponding definitions as *constants* and *constant* (or *parameterless*) *definitions*, respectively. We omit the "( )" in $A(\,)$.

Given $A \stackrel{\texttt{def}}{=} P$, requiring all names in $fn(P)$ to be formal parameters, as we did for CCS_p (Convention 2), would be too restrictive—P would have no visible actions. Consequently, let us drop the requirement in Convention 2 to consider a fragment allowing *only* constant definitions but *with possible occurrence of free names in their bodies*. The rules for this fragment, which we call CCS_k, are simply those of CCS_p. In this case Rule CALL (which for CCS_k we prefer to call CONS) takes the form

$$\text{CONS}\ \frac{P_A \stackrel{\alpha}{\longrightarrow} P'}{A \stackrel{\alpha}{\longrightarrow} P'}\ \text{if}\ A \stackrel{\texttt{def}}{=} P_A \qquad (4)$$

i.e., 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.

Relabelings. The reader familiar with process algebras may have noticed that CCS_k is basically CCS except for the absence of *relabeling*. A relabeling $f : \underline{Act} \rightarrow Act$ is the identity for all but finitely many actions. Furthermore, f satisfies $\overline{f(a)} = f(\overline{a})$, $f(a) \neq \tau$ and $f(\tau) = \tau$. For each action α performed by P, the relabeled process $P(f)$ executes $f(\alpha)$. More precisely:

$$\text{REL}\ \frac{P \stackrel{\alpha}{\longrightarrow} P'}{P(f) \stackrel{f(\alpha)}{\longrightarrow} P'(f)}$$

[1] The relation $\rightsquigarrow^*$ is a preorder. By induced strict order we mean the strict component of $\rightsquigarrow^*$ modulo the equivalence relation obtained by taking the symmetric closure of $\rightsquigarrow^*$.

Remark 1. It is well known that the behavior specified by any process involving only *injective* relabelings can be equivalently specified (up to strong bisimilarity) by a relabeling-free process with the help of parametric definitions [12]. This is important since, as pointed out in [9], one usually works with injective relabelings. □

2.3 Recursion Expressions: CCS$_\mu$

Hitherto we have seen process expressions whose recursive behaviors are specified by 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 our set of finite processes (Equation 1) with such recursive expressions:

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

Here $\mu X.P$ binds the occurrences of the *process variable* X in P. As for bound and free names, we define 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 not 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:

$$\text{REC}\ \frac{P[\mu X.P/X] \xrightarrow{\alpha} P'}{\mu X.P \xrightarrow{\alpha} P'} \tag{6}$$

We call CCS$_\mu$ the resulting calculus. From [5] it follows that in CCS$_\mu$ we can identify processes up to name α-equivalence. Furthermore, we make a typical assumption on CCS$_\mu$ process variables; they need to be guarded. We say that an expression is *guarded* in P iff it lies within some sub-expression of P of the form $\alpha.Q$.

Convention 3 (Guarded Recursion) *We shall confine ourselves to CCS$_\mu$ processes where all variables are guarded.*

Static and Dynamic Scope. An interesting issue regarding expression $P[\mu X.P/X]$ (cf. rule REC) is whether *bound names* in P should be renamed to avoid captures (i.e., *name α-conversion*). Such a requirement seems necessary should we want to identify processes up to *α-equivalence*. In fact, the requirement gives CCS$_\mu$ *static* scoping of names. Let us illustrate this with an example.

Example 1. Consider $\mu X.P$ with $P = (a \parallel (\overline{a}.b \parallel X)\backslash a)$. First, let us assume we perform name α-conversions to avoid captures. So, $[\mu X.P/X]$ in P renames the bound a by a fresh name, say c, thus avoiding the capture of $P's$ free a in the replacement: I.e,

$$P[\mu X.P/X] = (a \parallel (\bar{c}.b \parallel \mu X.P)\backslash c) = (a \parallel (\bar{c}.b \parallel \mu X.(a \parallel (\overline{a}.b \parallel X)\backslash a))\backslash c)$$

The reader may care to verify (using the rules in Table 1 plus Rule REC) that b will not be performed; i.e., there is no $\mu X.P \xrightarrow{\alpha_1} P_1 \xrightarrow{\alpha_2} \ldots$ s.t. $\alpha_i = b$.

Now let us assume that the substitution makes no name α-conversion. This causes a free occurrence of a in P (indicated by the dashed circle) to get bound, *dynamically*, *in the scope* of the outermost restriction: I.e.,

$$P[\mu X.P/X] = (a \parallel (\bar{a}.b \parallel \mu X.P)\backslash a) = (a \parallel (\bar{a}.b \parallel \mu X.(\dot{a} \parallel (\overline{a}.b \parallel X)\backslash a))\backslash a).$$

The reader can verify that, in this case, b may eventually be performed. Such an execution of b cannot be performed by $\mu X.Q$ where Q is $(a \parallel (\overline{c}.b \parallel X)\backslash c)$ i.e, P with the binding and bound occurrence of a syntactically replaced with c. This shows that name α*-equivalence does not hold when dynamic scoping is used*. □

Remark 2. 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, [9] 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 1 one can verify that, in CCS_k, A exhibits exactly the same dynamic scoping behavior illustrated by the example. So, *name α-equivalence does not hold in CCS* (exposing yet another semantic difference between CCS and the π-calculus as the latter uses static scoping and satisfies α-equivalence). □

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

One simple way of expressing infinite behavior is by using replication. Although mostly found in calculi for mobility, replication has also been studied in the context of CCS [3, 2]. In this case the syntax of finite processes (Equation 1) is extended with:

$$P, Q, \ldots := \ldots \mid !P \tag{7}$$

Intuitively $!P$ behaves as $P \parallel P \parallel \ldots \parallel P \parallel !P$; as many copies of P as you wish. We call $\text{CCS}_!$ the calculus that results from the above syntax. The operational rules for $\text{CCS}_!$ are those in Table 1 plus the following rule:

$$\text{REP}\ \frac{P \parallel !P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'} \tag{8}$$

From [10] we know that $\text{CCS}_!$ processes can be identified under α-equivalence.

2.5 Summary of Calculi

We described several calculi based on the literature of CCS. We have CCS_p the calculus with parametric definitions and CCS_k the calculus with constant (or parameterless) definitions. We also have CCS_μ the statically scoped calculus with recursive expressions—the dynamically scoped version instead coincides with CCS_k. Finally, we have the calculus with replication, $\text{CCS}_!$.

Convention 4 *Henceforth, we use Σ to denote the signature $\{\mathtt{p}, \mathtt{k}, \mu, !\}$ of our calculi sub-indexes. We shall use $\sigma, \sigma', \ldots$ to range over Σ. In the following sections, we shall index sets and relations with the appropriate symbol from Σ to make explicit the calculus under consideration. For example, $\xrightarrow{\alpha}_\sigma$ represents a transition of CCS_σ. Similarly, we shall use $Proc_\sigma$ to denote the set of CCS_σ processes. However, we may omit the indexes when these are unimportant or clear from the context.*

3 Expressiveness and Classification Criteria

Here we introduce the means we shall use to compare and classify the various calculi.

Comparing Calculi: Bisimilarity. We wish to compare the behavior of two given processes P and Q w.r.t. the standard notion of (weak) bisimilarity [9]. However, P and Q may belong to two different calculi, say CCS_σ and $CCS_{\sigma'}$. We then find it convenient to state the standard notion as below. First, recall that the *converse* of a binary relation $\mathcal{S}$ is $\mathcal{S}^{-1} = \{(e', e) \mid (e, e') \in \mathcal{S}\}$

Definition 1 (Bisimilarity). *A relation $\mathcal{S} \subseteq Proc_\sigma \times Proc_{\sigma'}$, with $\sigma, \sigma' \in \Sigma$, is said to be a* (strong) simulation *iff for all $(P, Q) \in \mathcal{S}$:*

$$\textit{whenever } P \xrightarrow{\alpha}_\sigma P' \textit{ then, for some } Q', Q \xrightarrow{\alpha}_{\sigma'} Q' \textit{ and } (P', Q') \in \mathcal{S}.$$

The relation $\mathcal{S}$ is called a (strong) bisimulation *if both $\mathcal{S}$ and its converse are simulations. Furthermore, we say that $P \in Proc_\sigma$ and $Q \in Proc_{\sigma'}$ are strongly bisimilar (w.r.t., σ and σ'), written $P \sim^{\sigma'}_{\sigma} Q$ (or simply $P \sim Q$), iff there exists a bisimulation $\mathcal{S} \subseteq Proc_\sigma \times Proc_{\sigma'}$, such that $(P, Q) \in \mathcal{S}$. The relation $\sim$ is called (strong) bisimilarity.* □

Let us now recall the weaker notion of bisimilarity which abstracts away from silent (i.e., τ) actions. We need some little notation. Define $\stackrel{s}{\Longrightarrow}$,with $s = \alpha_1.\alpha_2.\ldots. \in \mathcal{L}^*$, as $(\xrightarrow{\tau})^* \xrightarrow{\alpha_1} (\xrightarrow{\tau})^* \ldots (\xrightarrow{\tau})^* \xrightarrow{\alpha_n} (\xrightarrow{\tau})^*$. The notions of *weak (bi)simulation* and *weak bisimilarity* can be derived from the strong versions by replacing in Definition 1 $\xrightarrow{\alpha}$ and $\sim$ with $\stackrel{s}{\Longrightarrow}$ and $\approx$, respectively (cf. [9, §7.1]). We can now make precise our criterion for expressiveness.

Definition 2. *We say that CCS_σ is* as expressive as *$CCS_{\sigma'}$ iff for every $P \in Proc_\sigma$, there exists $Q \in Proc_{\sigma'}$ such that P and Q are weakly bisimilar (w.r.t. σ and σ').* □

To prove equivalence on expressiveness, we shall provide *(weak) bisimulation preserving* mappings $[\![\cdot]\!]$, which we call *encodings*, from the processes of one calculus into the processes of another. Some encodings will be chosen to preserve one further property: *divergence*. It should be noticed that unlike strong bisimulation, weak bisimulation identifies some divergent processes with non-divergent ones. Let us formalize the notion of divergence.

Definition 3. *We say that P is* divergent *(or that it diverges) iff $P(\xrightarrow{\tau})^\omega$, i.e., there exists an infinite sequence $P = P_0 \xrightarrow{\tau} P_1 \xrightarrow{\tau} \ldots$.* □

Classifying Calculi: Decidability of Divergence. We shall classify the various calculi according to whether divergence is *decidable* for the calculus. By divergence being decidable for CCS_σ, we mean that there exists an algorithm which can fully determine, given $P \in \mathit{Proc}_\sigma$, whether P is divergent.

4 Encodings

In this section we give the various encodings. Furthermore, in order to classify the calculi w.r.t. to the decidability of divergence, we shall also prove the relevant encodings to be divergence-preserving and computable.

4.1 Encoding CCS_p into CCS_k

Here we give an encoding $[\![\cdot]\!] : \text{CCS}_\text{p} \to \text{CCS}_\text{k}$. For the sake of presentation, we consider only unary parametric definitions. The encoding can be easily generalized to the n-ary case by extending our concepts and definitions from names to vector of names.

For simplicity and w.l.o.g we assume there is a definition of the form $M^P(x) \stackrel{\text{def}}{=} P \in D_P$ with M^P not occurring in P and D_P being the *finite* set of definitions arising from the identifiers in P—think of M^P as the "main" procedure of P. Formally, D_P is the set of definitions for the identifiers in the closure under $\rightsquigarrow$ of $\{M^P\}$ (See Convention 1).

For the encoding we would like to associate to each process P in CCS_p a process in CCS_k substituting B_y for each invocation $B(y)$ in P. How many invocations of this form should be considered? Given that CCS_p satisfies α-equivalence, there is potentially an infinite number of such invocations—which means that a careful choice of names y is needed if we want to obtain a finite number of constant definitions. To complicate things further, rule CALL may force an α-conversion anywhere in the execution of a CCS_p process.

Instead of presenting the encoding mapping right away, we proceed in a stepwise fashion. We start with the set of all CCS_k processes (because of α-conversions) that may be associated to a single CCS_p process. In Def. 5, we identify sufficient conditions for subsets of those processes to define a good encoding into CCS_k. Finally, we show a procedure to effectively construct such an encoding.

Definition 4. *The function* $\widehat{\cdot}$: $\mathit{CCS}_\text{p} \to \mathcal{P}(\mathit{CCS}_\text{k})$ *is inductively defined over the structure of its parameter:*

$$\widehat{P} = \begin{cases} \{\mathbf{0}\} & \text{if } P = \mathbf{0} \\ \{\alpha.Q \mid Q \in \widehat{P'}\} & \text{if } P = \alpha.P' \\ \{Q_1 \parallel Q_2 \mid Q_i \in \widehat{P}_i \ , \ i = 1,2\} & \text{if } P = P_1 \parallel P_2 \\ \{\Sigma_{i\in I}\alpha_i.Q_i \mid Q_i \in \widehat{P}_i \ , \ i \in I\} & \text{if } P = \Sigma_{i\in I}\alpha_i.P_i \\ \{Q\backslash\beta \mid \exists P' \in \mathit{Proc}_p \ . \ P \equiv_\alpha P'\backslash\beta \wedge Q \in \widehat{P'}\} & \text{if } P = P''\backslash\alpha \\ \{A_y\} & \text{if } P = A(y). \end{cases}$$

Example 2. If $P = a.\overline{b}.\mathbf{0} + B(b)$ then $\widehat{P}$ is the singleton $\{a.\overline{b}.\mathbf{0} + B_b\}$. □

Example 3. If $P = (z.x.\mathbf{0} \parallel \overline{x}.\mathbf{0} \parallel A(z))\backslash z$ then $\widehat{P}$ contains (among many others) the elements $(z.x.\mathbf{0} \parallel \overline{x}.\mathbf{0} \parallel A_z)\backslash z$ and $(y.x.\mathbf{0} \parallel \overline{x}.\mathbf{0} \parallel A_y)\backslash y$. □

Remark 3. The definition of $\widehat{\cdot}$ is invariant under α-conversions. More generally, it can be shown that $P \equiv_\alpha Q$ iff $\widehat{P} = \widehat{Q}$. □

We now define $[\![P]\!]$ which requires specifying the set $[\![D_P]\!]$ of (constant) definitions induced by $[\![P]\!]$.

Definition 5. *Given a process $P \in CCS_{\mathrm{p}}$ with associated definition set D_P, an* encoding *of P in CCS_{k} is defined as the CCS_{k} constant M_x^P (called $[\![P]\!]$) together with an underlying set of definitions $[\![D_P]\!]$, satisfying the following two conditions:*

(I) $[\![D_P]\!]$ *contains a definition* $(M_x^P \stackrel{\mathrm{def}}{=} P_0)$ *for some* $P_0 \in \widehat{P}$.

(II) If $(A_y \stackrel{\mathrm{def}}{=} Q_A) \in [\![D_P]\!]$, B_z *occurs in* Q_A *and* $(B(x) \stackrel{\mathrm{def}}{=} P_B) \in D_P$ *, then there is* $Q_B \in \widehat{P_B[z/x]}$ *s.t.* $(B_z \stackrel{\mathrm{def}}{=} Q_B) \in [\![D_P]\!]$.

We understand a set of definitions to contain at most one definition per process constant. A set of definitions satisfying conditions (I) and (II) is called an encoding set. □

Observe that, according to the definition, there are (infinitely) many encodings for a given process P. Not only can an encoding be extended with definitions and still remain an encoding, but also condition (II) allows for many different definitions for constant B_z. If, say, $Q_B, Q'_B \in \widehat{P_B[z/x]}$, then an encoding $[\![D_P]\!]$ may contain either the definition $B_z \stackrel{\mathrm{def}}{=} Q_B$ or the definition $B_z \stackrel{\mathrm{def}}{=} Q'_B$ (but not both).

The following lemma[2] characterizes the shape of minimal encoding sets.

Lemma 1. *Given an encoding set $[\![D_P]\!]$, the set* $\mathbf{D} = \{(A_y \stackrel{\mathrm{def}}{=} Q_A) \in [\![D_P]\!] \mid Q_A \in \widehat{P_A[y/x]}\}$*, is an encoding set (included in $[\![D_P]\!]$).* □

Recall that D_P contains finitely many definitions. We shall show that an encoding can be effectively constructed (so that the resulting set of definitions $[\![D_P]\!]$ is also finite). First let us illustrate the construction with the following example.

Example 4. Let $P = A(x)$ with $D_P = \{A(x) \stackrel{\mathrm{def}}{=} (z.x.\mathbf{0} \parallel \overline{x}.\mathbf{0} \parallel A(z))\backslash z\}$. We proceed to define an encoding by constructing a set $[\![D_P]\!]$ so that it satisfies conditions (I) and (II). To satisfy condition (I), let $M_x^P \stackrel{\mathrm{def}}{=} (z.x.\mathbf{0} \parallel \overline{x}.\mathbf{0} \parallel A_z)\backslash z \in [\![D_P]\!]$. Then, condition (II) requires a definition such as: $A_z \stackrel{\mathrm{def}}{=} (z_1.z.\mathbf{0} \parallel \overline{z}.\mathbf{0} \parallel A_{z_1})\backslash z_1 \in [\![D_P]\!]$. Notice that due to α-conversion in equation A_z we have obtained a new name z_1 and hence we have to give a new definition for A_{z_1}. Of course because of the α-conversion we could have chosen another fresh name z_2, but that would only lead to a different but equally useful encoding. Using condition (II) again: $A_{z_1} \stackrel{\mathrm{def}}{=} (z.z_1.\mathbf{0} \parallel \overline{z_1}.\mathbf{0} \parallel A_z)\backslash z \in [\![D_P]\!]$, and we are done; no other definition needs to be added to $[\![D_P]\!]$. It is easy to check that the resulting set satisfies conditions (I) and (II), and therefore constitutes an encoding of P in $\mathrm{CCS}_{\mathrm{k}}$. □

[2] See [6] for the proof of the lemmas in this paper.

We now show that for any P, one can compute an encoding set $[\![D_P]\!]$.

Theorem 1. *For any $P \in CCS_{\mathrm{p}}$ with a finite set D_P of associated definitions, one can effectively construct an encoding set $[\![D_P]\!]$.*

Proof. Let $Var(D_P)$ be the set of all the names occurring in D_P. For each $A(x) \stackrel{\text{def}}{=} P_A \in D_P$ and each $y \in Var(D_P)$, choose a P_A^y so that $P_A^y \in \widehat{P_A[y/x]}$. Define $S = \{A_y \stackrel{\text{def}}{=} P_A^y \mid (A(x) \stackrel{\text{def}}{=} P_A) \in D_P \wedge y \in Var(D_P)\}$. Notice that S is a finite set. Proceed by defining $\mathcal{F} = \{z \mid \exists \text{ constant } B_z.\ B_z \text{ occurs in } S \wedge B_z \text{ is not defined in } S\}$, and notice that $\mathcal{F}$ is a finite set too. Observe that, for each definition $A(x) \stackrel{\text{def}}{=} P_A \in D_P$ and for each $y \in \mathcal{F}$, the substitution $P_A[y/x]$ requires no alpha-conversion. Consequently it is possible to choose $P_A^y \in \widehat{P_A[y/x]}$ so that for each constant B_z occurring in P_A^y, $z \in (Var(D_P) \cup \mathcal{F})$. We have now a candidate Σ_{D_P} for the set of definitions in the encoding of P. It is simply defined as $\Sigma_{D_P} = \{A_y \stackrel{\text{def}}{=} P_A^y \mid (A(x) \stackrel{\text{def}}{=} P_A) \in D_P \wedge y \in (Var(D_P) \cup \mathcal{F})\}$. Since $(M_x^P \stackrel{\text{def}}{=} P_0) \in S \subseteq \Sigma_{D_P}$, with $P_0 \in \widehat{P}$, our candidate set satisfies condition (I) in Def. 5. It remains to be shown that Σ_{D_P} also satisfies condition (II). Assume now that $(A_y \stackrel{\text{def}}{=} Q_A) \in \Sigma_{D_P}$, that B_z occurs in Q_A and that $(B(x) \stackrel{\text{def}}{=} P_B) \in D_P$. By construction, $z \in (Var(D_P) \cup \mathcal{F})$, and therefore $(B_z \stackrel{\text{def}}{=} P_B^z) \in \Sigma_{D_P}$. This shows that Σ_{D_P} satisfies condition (II). Therefore, our effectively constructed candidate Σ_{D_P} is indeed an encoding $[\![D_P]\!]$. □

We now state the correctness of the encoding up to (strong) bisimilarity. The theorem actually says that parametric definitions are not more expressive than constant definitions.

Theorem 2. *Given a process $P \in CCS_{\mathrm{p}}$ with associated set of definitions D_P, any encoding $[\![P]\!]$ with definition set $[\![D_P]\!]$ satisfies $P \sim_p^k [\![P]\!]$.* □

Remark 4. It follows from Remark 1 and the above theorem that injective relabelings are redundant in CCS (up to strong bisimilarity).

Now, [3] shows that divergence is undecidable for $\mathrm{CCS_p}$. Furthermore, we also showed that the above encoding is computable. Since divergence is invariant under strong bisimilarity, we can then conclude the following result.

Theorem 3. *The divergence problem is undecidable for CCS_{k}.* □

4.2 Encoding $\mathrm{CCS_k}$ into $\mathrm{CCS_p}$

Intuitively, if the free names are treated dynamically, then they could equivalently be passed as parameters. Thus, we can define the encoding as follows:

Definition 6. *Given $P \in CCS_{\mathrm{k}}$ with a set of associated constant definitions of the form $A \stackrel{\text{def}}{=} P_A$ and given a strict total order over names, the encoding of P into CCS_{p} is a process $[\![P]\!]$ with associated set of definitions*

$$\left\{ A(x_1, \ldots, x_n) \stackrel{\text{def}}{=} [\![P_A]\!] \mid (A \stackrel{\text{def}}{=} P_A) \in Dp \wedge fn(P_A) = \{x_1, \ldots, x_n\} \right\}.$$

The encoding function $[\![\cdot]\!]\colon Proc_k \to Proc_p$*, which is an homomorphism over all other operators, satisfies* $[\![A]\!] = A(x_1, \ldots, x_n)$ *where* $fn(P_A) = \{x_1, \ldots, x_n\}$*. Both in definitions and in invocations, all lists of argument names are assumed sorted.* □

(By homomorphism we mean that $[\![P \parallel Q]\!] = [\![P]\!] \parallel [\![Q]\!]$ and similarly for the other operators.)

The following theorem states that constant definitions with dynamic scoping are not more expressive than parametric definitions with static scoping.

Theorem 4. *For every process* P *in* CCS_{k}, $[\![P]\!] \sim_k^p P$. □

4.3 Encoding CCS_μ into $CCS_!$

The main idea behind this encoding is to associate a replicated process $!x.P'$ to each occurrence of the recursion operator, $\mu X.P$. In the past a similar approach has been used to show that, in the π-calculus, recursion can be expressed using replication [13]. While in [13] each π-calculus process and its encoding happen to be strongly bisimilar, this is not the case for CCS_μ. Although in general a CCS_μ process is only weakly bisimilar to its encoding, we show that divergence properties are always preserved.

Our definition assumes that process variables are indexed by I, i.e. $\{X_i \mid i \in I\}$:

Definition 7. *Let* $[\![\cdot]\!]\colon Proc_\mu \to Proc_!$ *be the encoding function that is homomorphic over all operators in the sub-calculus defining finite behavior and is otherwise defined as follows:*

$$\begin{aligned} [\![X_i]\!] &= \overline{x_i}.\mathbf{0} \\ [\![\mu X_i.P]\!] &= (!x_i.[\![P]\!] \parallel \overline{x_i}.\mathbf{0})\backslash x_i \end{aligned}$$

where the names $\{x_i \mid i \in I\}$ *are fresh.* □

The freshness condition on the variables x_i is meant to guarantee that every time we apply $[\![P]\!]$, P mentions none of them.

Remark 5. The above encoding would not work had we adopted dynamic scoping in the Rule REC for CCS_μ (see Remark 2). The $\mu X.P$ in Example 1 actually gives us a counter-example. □

The following example illustrates why a CCS_μ process may not be strongly bisimilar to its encoding.

Example 5. Consider the CCS_μ process $P = \mu X.a.X$ with corresponding encoding $[\![P]\!] = (!x.a.\bar{x} \parallel \bar{x})\backslash x$. They are clearly not strongly bisimilar, as P has the single trace $\mu X.a.X \xrightarrow{a}_\mu \mu X.a.X \xrightarrow{a}_\mu \mu X.a.X \ldots$ while $[\![P]\!]$ only produces $(!x.a.\bar{x} \parallel \bar{x})\backslash x \xrightarrow{\tau}_\mu (!x.a.\bar{x} \parallel a.\bar{x})\backslash x \xrightarrow{a}_\mu (!x.a.\bar{x} \parallel \bar{x})\backslash x \xrightarrow{\tau}_\mu \ldots$ Observe that each transition in the first trace uses rule REC, and that every other step in the second one reflects explicitly, as an internal transition, each recursive call. □

In comparing CCS_μ and $CCS_!$, we find it convenient to consider yet another variant calculus, as an intermediate step, which we call CCS_τ: Its syntax agrees entirely with

CCS$_\mu$'s (i.e. $Proc_\tau = Proc_\mu$), and its semantics differs from CCS$_\mu$'s only by a replacement of REC with a rule in which the unfolding performs a τ action—hence the name CCS$_\tau$:

$$\text{REC'} \quad \frac{}{\mu X.P \xrightarrow{\tau}_\tau P[\mu X.P/X]}$$

Example 6. Consider process P as given in Example 5 but this time within CCS$_\tau$ (which is possible thanks to $Proc_\tau = Proc_\mu$). The only trace exhibited by P is: $\mu X.a.X \xrightarrow{\tau}_\tau a.(\mu X.a.X) \xrightarrow{a}_\tau \mu X.a.X \xrightarrow{\tau}_\tau \ldots$ and therefore $P \sim^!_\tau [\![P]\!]$. □

In fact, the property illustrated by the previous example holds in general, as stated in the following theorem. The proof is essentially an adaptation of the one given by Sangiorgi and Walker in [13].

Theorem 5. *If $P \in CCS_\tau$, then $P \sim^!_\tau [\![P]\!]$.* □

Because strong bisimilarity is known to preserve expressiveness and divergence, the above theorem lets us reduce the problem of studying the encoding to investigating the relation between CCS$_\tau$ and CCS$_\mu$.

We define a binary relation $\mathcal{R} \in (Proc_\mu \times Proc_\tau)$ as follows: $P \mathcal{R} Q$ iff there exist $n \geq 0$ such that $P = Q_0 \xrightarrow{\tau}_\tau Q_1 \xrightarrow{\tau}_\tau \ldots Q_n = Q$, where each derivation $Q_i \xrightarrow{\tau}_\tau Q_{i+1}$ involves the application of rule REC'.

We show that besides being a weak bisimulation relation, $\mathcal{R}$ also relates processes with equal divergence properties. As a first step, notice that each $\xrightarrow{\alpha}_\mu$ transition can be mimicked by $\mathcal{R}$-related processes in CCS$_\tau$ after possibly some τ transitions (which correspond to recursive invocations involving rule REC').

Lemma 2. *If $P \mathcal{R} Q$ and $P \xrightarrow{\alpha}_\mu P'$ then there exists Q' such that $Q(\xrightarrow{\tau}_\tau)^* \xrightarrow{\alpha}_\tau Q'$ and $P' \mathcal{R} Q'$.* □

Remark 6. Notice that we have restricted our attention to processes where all variables are guarded. Without this assumption divergence would not be preserved by our encoding. For example, $\mu X.X$ diverges in CCS$_\tau$ but deadlocks in CCS$_\mu$. □

Lemma 3. *If $P \mathcal{R} Q$ and there is a derivation of $Q \xrightarrow{\alpha}_\tau Q'$ which does not involve the application of rule REC', then there exists P' s.t. $P \xrightarrow{\alpha}_\mu P'$ and $P' \mathcal{R} Q'$.* □

To show that two identical processes, interpreted in CCS$_\mu$ and resp. CCS$_\tau$, are weakly bisimilar we need to show two simulations: One is provided by Lemma 2 and the other follows by a combination of Lemma 3 and the definition of $\mathcal{R}$ (to cover the case in which $Q \xrightarrow{\alpha}_\tau Q'$ does use rule REC'). The result is summarized by our next theorem.

Theorem 6. *Given a process P in CCS$_\mu$, $P \approx^\tau_\mu P$.* □

Observe that this is still not enough to show that $\mathcal{R}$ relates processes with the same divergence properties. If $P \mathcal{R} Q$ and Q diverges, Lemma 3 is not strong enough to show that P may execute a single τ transition. However, it turns out that Q cannot diverge by executing only recursive calls (again, a result of our assumptions on guarded summation

and guarded recursion; see Remark 6 and [6]). So, if after some finite execution trace, Q performs a τ transition that does not involve REC', we can apply Lemma 3 to deduce that P may also perform a τ transition. Since this process can be repeated endlessly it must be concluded that divergence in CCS_τ forces divergence in CCS_μ. The converse is an easy consequence of Lemma 2. That is, we have shown:

Proposition 1. *For $P \in CCS_\mu$, $P(\xrightarrow{\tau}_\mu)^\omega$ iff $P(\xrightarrow{\tau}_\tau)^\omega$.* □

Our journey from CCS_μ to $\text{CCS}_!$ through CCS_τ has rendered the following result.

Corollary 1. *For $P \in Proc_\mu$, $P \approx^!_\mu [\![P]\!]$. Moreover, P diverges iff $[\![P]\!]$ diverges.* □

From the above corollary, the fact that the encoding is computable, and the result of [3] showing that divergence is decidable for $\text{CCS}_!$ we conclude the following:

Theorem 7. *The divergence problem is decidable for CCS_μ.* □

4.4 Encoding $\text{CCS}_!$ into CCS_μ

Except for the syntax and our restriction to guarded recursion, this encoding is essentially that given in [13] for the π-calculus.

Definition 8. *Let $[\![\cdot]\!]\colon Proc_! \to Proc_\mu$ be the encoding function that is homomorphic over all operators in the sub-calculus defining finite behavior and is otherwise defined as follows: $[\![!P]\!] = \mu X.([\![P]\!] \parallel \tau.X)$.* □

In fact, the proof of the following theorem follows that in [13].

Theorem 8. *For $P \in Proc_!$, $P \approx^\mu_! [\![P]\!]$.* □

Observe that, because of our restriction to guarded recursion, the encoding does not preserve divergence. For instance, if $P = !\mathbf{0}$ then P is deadlocked in $\text{CCS}_!$; but

$$[\![P]\!] = \mu X.(\mathbf{0} \parallel \tau.X) \xrightarrow{\tau}_\mu \mathbf{0} \parallel \mu X.(\mathbf{0} \parallel \tau.X) \xrightarrow{\tau}_\mu \mathbf{0} \parallel \mathbf{0} \parallel \mu X.(\mathbf{0} \parallel \tau.X) \xrightarrow{\tau}_\mu \ldots .$$

5 Concluding Remarks

We studied the relative expressiveness (w.r.t. weak bisimilarity) and the decidability of divergence for some CCS-like calculi. The calculi differ on the constructs used to express infinite behavior and on the treatment of scoping of channel names; the finite core being the same. We showed that parameters can be removed from recursive definitions without loss of expressiveness provided dynamic name scoping is applied. We also showed that the expressiveness of recursive expressions with static scoping corresponds precisely to that of replication. We partitioned the calculi into two groups: For one, divergence is undecidable (i.e., constant and parametric definitions), whereas it is decidable for the other (i.e., replication and recursive expressions with static scoping). Figure 1, in the Introduction, illustrates these results.

As a consequence of our results, we proved that a substantial family of relabelings, the injective ones, is redundant in CCS (see Remark 4). We also showed that a slightly different interpretation of Rule REC, namely performing also name α-conversions in substitutions, can render decidable (w.r.t. divergence) an otherwise undecidable calculus (see Remark 2). We illustrated that CCS exhibits dynamic name scoping and that it does not preserve α-equivalence.

Related Work. Most of the related work was already discussed in the Introduction. The most closely related work is [3] which shows the (un)decidability of divergence for $\mathrm{CCS}_{\mathtt{p}}$ and $\mathrm{CCS}_{!}$. Here we extend these results to the corresponding equally expressive calculi. The work on ECCS [5], perhaps the most immediate predecessor of the π-calculus, advocates static scoping of names. In contrast, the work on CHOCS [14] advocates dynamic name scoping in the context of higher-order CCS. Furthermore, the CCS variant in [10] uses statically scoped parametric definitions while the Edinburgh Concurrency Workbench tool [4] uses dynamic scoping for parametric definitions.

The work in [1] shows that that in CCS, non-injective relabelings lead to a sensible different treatment of asynchrony w.r.t the injective ones. We believe that it would be interesting to investigate more qualitative distinctions for these two kinds of relabelings.

Acknowledgments. We are indebted to Maurizio Gabbrielli, Jean-Jacques Lévy, Sergio Maffeis, Catuscia Palamidessi, Joachim Parrow, Rosario Pugliese and Davide Sangiorgi, for insightful discussions on the topics of this paper.

References

1. M. Boreale, R. De Nicola, and R. Pugliese. Trace and testing equivalence on asynchronous processes. *Information and Computation*, 172(2):139–164, 2002.
2. N. Busi, M. Gabbrielli, and G. Zavattaro. The expressive power of replication in CCS. Draft, 2003.
3. N. Busi, M. Gabbrielli, and G. Zavattaro. Replication vs. recursive definitions in channel based calculi. In *ICALP'03*, volume 2719 of *LNCS*, pages 133–144. Springer Verlag, 2003.
4. R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A semantics based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, 1993.
5. U. Engberg and M. Nielsen. A calculus of communicating systems with label-passing. Technical report, University of Aarhus, 1986.
6. P. Giambiagi, G. Schneider, and F.D. Valencia. On the expressiveness of CCS-like calculi. Technical report, Uppsala University, 2004. Postscript available from `http://www.sics.se/fdt/publications/GSV-Expr-TR04.ps`.
7. S. Maffeis and I. Phillips. On the computational strength of pure ambient calculi. In *EXPRESS'03*, 2003.
8. R. Milner. Calculi for synchrony and asynchrony. Technical Report CSR-104-82, University of Edinburgh, 1982.
9. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
10. R. Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.
11. C. Palamidessi. Comparing the expressive power of the synchronous and the asynchronous π-calculus. In ACM Press, editor, *POPL'97*, pages 256–265, 1997.
12. J. Parrow. An introduction to the π-calculus. In *Handbook of Process Algebra*, pages 479–543. Elsevier, 2001.
13. D. Sangiorgi and D. Walker. *The $\pi-$calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.
14. B. Thomsen. A calculus of higher order communicating systems. In *POPL'89*, ACM, pages 143–154, 1989.