From π-calculus to Higher-Order π-calculus — and back

Davide Sangiorgi[1]

Abstract. We compare the first-order and the higher-order paradigms for the representation of mobility in process algebras. The prototypical calculus in the first-order paradigm is the *π-calculus*. By generalising its sort mechanism we derive an ω-order extension, called *Higher-Order π-calculus* (HOπ). We give examples of its use, including the encoding of λ-calculus. Surprisingly, we show that such an extension does not add expressiveness: Higher-order processes can be faithfully represented at first order. We conclude that the first-order paradigm, which enjoys a simpler and more intuitive theory, should be taken as *basic*. Nevertheless, the study of the λ-calculus encodings shows that a higher-order calculus can be very useful for reasoning at a more abstract level.

1 Introduction

A *mobile system* is a system with a dynamically changing communication topology. Examples from operating systems are a resource which has a single owner at any time but whose ownership can be changed as time passes, or process migration, in which tasks or processes can be exchanged among processors to optimise their load balance.

There are two approaches to represent mobility in process algebra. In the *higher-order paradigm* mobility is achieved by allowing agents to be passed as values in a communication; Thomsen's Plain CHOCS [Tho90] and Boudol's γ-calculus [Bou89] belong to this category. In the *first-order paradigm* only ports can be transmitted (we shall use interchangeably the words port, name and channel). The π-calculus is the prototypical first-order calculus. It was introduced by Milner, Parrow and Walker in [MPW92] and later refined by Milner [Mil91] with the addition of *sorts* and of communication of tuples (*polyadic π-calculus*). The choice of the first-order paradigm for π-calculus was motivated — among other reasons — by the belief that reference passing is enough to represent more involved operations like process passing. Our goal here is to validate this claim, showing that no power, only convenience, is gained by moving to higher order. To this end, we introduce a new calculus, called *Higher-Order π-calculus* (HOπ), which enriches the π-calculus with explicit higher-order communications. In the HOπ not only names, but also processes and parametrised processes of arbitrarily high order, can be transmitted. In this sense, if the ordinary π-calculus is of first order and Plain CHOCS is of second order, then HOπ is of ω order. We show that HOπ is *representable* within π-calculus.

But what does it mean that a given source language is representable within a given target language? Typically there are three phases:

(1) Formal definition of the semantics of the two languages;
(2) Definition of the encoding from the source to the target language;
(3) Proof of the correctness of the encoding w.r.t. the semantics given.

The predominant approach to the semantics of concurrent systems is *operational*. The possible evolutions of processes are inductively described in terms of *transition systems* which then are quotien-

---

[1] address: Dep. Comp. Science, Universtity Edinburgh, JCMB, Mayfield road, Edinburgh EH9 3JZ, U.K. Email: sad@dcs.ed.ac.uk Work supported by the ESPRIT BRA project "CONFER".

ted by *equivalence relations* to abstract away from unwanted details. W.r.t. denotational semantics, the operational method necessitates a different approach to translation-correctness, where behaviours rather than meanings are compared. The choice of the behavioural equivalence, besides being "interesting", should be uniform on the calculi. Moreover, we want the encoding to be *fully abstract*, i.e. two source language terms should be equivalent if and only if their translations are equivalent. But since this does not reveal *how* this respectfulness is achieved, the result should be completed with the *operational correspondence* between a term and its translation (i.e. the connection between their transitions).

With the full abstraction demand, we have taken a *strong* point of view on representability. Indeed, while soundness is a necessary property, one might well consider milder forms of completeness, for instance by limiting the testing on target terms to encodings of source contexts. We asked for full abstraction because we wish to use the target terms in *any* contexts; and when two source terms are indistinguishable, their encodings should *always* be interchangeable. In other words, we want to be able to switch freely between the two calculi. In our case, where the source language is HOπ and the target language is π-calculus, this allows us on the one hand to make use of the abstraction power of HOπ, which comes from its ω-order nature. On the other hand, to rely on the more elementary and intuitive theory of π-calculus when reasoning over agents; in virtue of the representability result this theory can be lifted up to HOπ.

This paper is an extract of the core of the author's Ph.D. thesis [San92]. We have tried to keep the presentation rather informal, often preferring examples to meticulous technical details. We review the polyadic π-calculus in Section 2 and we introduce HOπ in Section 3. The behavioural equivalence adopted is defined in Section 4; we explain the reasons for our choice and we outline the problem of defining a natural bisimulation equivalence in a higher-order calculus. In Section 5 we present the compilation from HOπ to π-calculus, whose correctness is examined in Section 6. In Section 7 we look at some uses of the compilation, in particular the study of Milner's encodings of λ-calculus into π-calculus. In Section 8 we survey the related work and finally in Section 9, possible directions for future research.

Acknowledgements. I am most grateful to Robin Milner. This material was developed through a series of discussions with him. I wish to thank Matthew Hennessy, Benjamin Pierce and Gordon Plotkin for insightful comments; and Jean-Jacques Levy for having invited me to INRIA-Rocquencourt, where the paper has been written. This work has been supported by the ESPRIT BRA project "CONFER".

2 The polyadic π-calculus

2.1 Syntax

The letters $a, b, \ldots, x, y, \ldots$ stand for names, and P, Q for processes. We add a tilde to mean a possibly empty tuple. The class of π-calculus processes is built from names using the operators of prefixing, sum, parallel composition, restriction, matching and constant application:

$$P \quad :: \quad \textstyle\sum_{i \in I} \alpha_i.P_i \quad | \quad P_1 \,|\, P_2 \quad | \ \boldsymbol{\nu}\, x\, P \quad | \quad [x = y]P \quad | \quad D\langle \widetilde{x} \rangle$$

α is called *prefix* and can be either an input or an output:

$$\alpha \quad :: \quad x(\widetilde{y}) \quad | \quad \overline{x}\langle \widetilde{y} \rangle$$

Each constant D has a defining equation of the form $D \overset{\mathrm{def}}{=} (\widetilde{x})P$; the expression $(\widetilde{x})P$ is like a procedure, in which $\widetilde{x}$ represents the parameters. Therefore the operators emulate those of CCS [Mil89]; in addition, there is matching to test for equality of names. We refer to [MPW92, Mil91] for the intended interpretation of the operators. Application has the highest precedence; sum and parallel composition the lowest. In the sum, I represents a finite indexing set. When I is empty, we get the inactive process, written as 0; sometimes we abbreviate $\alpha.0$ as α. As usual, $+$ is taken to represent binary sum. As Milner does in [Mil91], our sums are *guarded*, i.e. the outermost operator of the summands is prefixing. Guarded sums simplify the reduction semantics of Section 2.3 and smooth the comparison between higher-order and first-order processes that we shall make in Section 5.

The operators $a(\widetilde{b}).P$, $(\widetilde{b})P$ and $\boldsymbol{\nu}\, b\, P$ bind all free occurrences of the names $\widetilde{b}$ and b in P. These binders give rise in the expected way to the definitions of *free names* of a term. The definitions of substitution and α-conversion are standard too, with renaming possibly involved to avoid capture of free names.

Sometimes *replication* is included in π-calculus in place of constants [Mil91]. The replication $!\,P$ intuitively represents $P \mid P \ldots$, i.e. an unbounded number of copies of P in parallel. It is easy to code it up using constants; and if the number of these is finite, the other way round holds too.

2.2 Sorting

All realistic systems which have been described with the π-calculus seem to obey some discipline in the use of names. The introduction of sorts and sortings into the π-calculus [Mil91] intends to make this name discipline explicit. In the polyadic π-calculus, sorts are also essential to avoid disagreement in the arities of tuples carried by a given name, or to be used by a given constant.

Names are partitioned into a collection of *subject sorts*, each of which contains an infinite number of names. We write $x : s$ to mean that x belongs to the subject sort s; this notation is extended to tuples componentwise. Then *object sorts*, ranged over by S, are just sequences over subject sorts, such as $(s_1, \ldots, s_n)$ or (s). Finally, a *sorting* is a function Ob mapping each subject sort to an object sort. We write $s \mapsto (\tilde{s}) \in Ob$, if Ob assigns the object sort $(\tilde{s})$ to s; in this case we say that $(\tilde{s})$ *appears* in Ob. By assigning the object sort (s_1, s_2) to the subject sort s, one forces the object part of any name in s to be a pair whose first component is a name of s_1 and whose second component is a name of s_2. CCS and the monadic unsorted π-calculus can be derived by imposing the sortings $\{\textsc{Name} \mapsto ()\}$ and $\{\textsc{Name} \mapsto (\textsc{Name})\}$ respectively, in which all names belong to the same subject sort $\textsc{Name}$.

If $a : s \mapsto (s_1, s_2)$, then for $a(\widetilde{x}).P$ and $\overline{a}\langle\widetilde{x}\rangle.P$ to respect Ob, it must be that $\widetilde{x} = x_1, x_2$, for some $x_1 : s_1$ and $x_2 : s_2$. Moreover in a matching $[a = b]$ we require that the tested names a and b belong to the same sort. Fynally, we have to guarantee the correctness of applications. To this end, we assign an object sort to agents: Processes take the sort $()$, whereas if $D \overset{\mathrm{def}}{=} (\widetilde{x})P$ and $\widetilde{x} : \widetilde{s}$, then D, and $(\widetilde{x})P$ take the sort $(\widetilde{s})$. Now, the requirement on $D\langle\widetilde{y}\rangle$ is that $\widetilde{s}$ exists s.t. $\widetilde{y} : \widetilde{s}$ and $D : (\widetilde{s})$. To sum up, a term is *well-sorted* for Ob if all its prefixes, matchings and applications obey the discipline given by Ob, as described above. We call an expression of sort $(\widetilde{s})$, for $\widetilde{s}$ non-empty, *abstraction*. Processes and abstractions are *agents*. We use F, G to range over abstractions and A to range over agents.

2.3 Operational Semantics

Following Milner [Mil90, Mil91], we shall give the operational semantics of the language in terms of a *reduction system* (as opposed to the "traditional" *labelled transition system*). In this technique, inspired by Berry and Boudol's Chemical Abstract Machine [BB90], axioms for a structural congruence relation are introduced prior to the reduction rules, in order to break a rigid, geometrical vision of concurrency and to allow for redexes as subterms. The interpretation of the operators of the language comes out neatly with reduction semantics, due to the compelling naturalness of each structural congruence and reduction rule.

Structural congruence, written $\equiv$, is the smallest congruence over the class of π-calculus agents which satisfies the rules below. (The symbol $\equiv$ should not be confused with $=$, the latter meaning syntactic equality between processes.)

1. $P \equiv Q$ if P is α-convertible to Q;
2. abelian monoid laws for $+$: $P + Q \equiv Q + P$, $P + (Q + R) \equiv (P + Q) + R$, $P + 0 \equiv P$;
3. abelian monoid laws for $\mid$: $P \mid Q \equiv Q \mid P$, $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$, $P \mid 0 \equiv P$;
4. $\boldsymbol{\nu}\, x\, 0 \equiv 0$, $\;\boldsymbol{\nu}\, x\, \boldsymbol{\nu}\, y\, P \equiv \boldsymbol{\nu}\, y\, \boldsymbol{\nu}\, x\, P$, $\;(\boldsymbol{\nu}\, x\, P) \mid Q \equiv \boldsymbol{\nu}\, x\, (P \mid Q)$, if x is not free in Q;
5. $[x = x]P \equiv P$;
6. if $D \overset{\mathrm{def}}{=} (\tilde{x})P$, then $D \equiv (\tilde{x})P$ (or, if instead replication is used, $!\,P \equiv P \mid\, !\,P$).

Now the *reduction rules*, expressing the notion of interaction:

$$\text{COM: } (\cdots + x(\tilde{y}).P) \mid (\cdots + \overline{x}\langle\tilde{z}\rangle.Q) \longrightarrow P\{\tilde{z}/\tilde{y}\} \mid Q$$

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

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

3 Higher Order π-calculus

3.1 Examples

In the π-calculus only object sorts of the form $(\tilde{s})$ are allowed. The sortings so obtained are first order, as indicated by the level of bracket nesting, which is limited to one. The *Higher-Order π-calculus* (HOπ) is essentially derived by dropping this limitation. Thus one may enforce processes to be communicated along a name x by declaring $x : s \mapsto (())$. Then an "executer", which receives a process at x and executes it, can be written as $x(X).X$; when put in parallel with $\overline{x}\langle P\rangle.Q$, it gives rise to the interaction

$$\overline{x}\langle P\rangle.Q \mid x(X).X \longrightarrow Q \mid P$$

Before formally defining the syntax and the semantics, let us look at more interesting examples.

Numerals

In [Mil91], Milner shows how to encode numbers in the π-calculus. If $\overline{y}^n$ represents the sequence $\overline{y}.\cdots.\overline{y}$ of length n, and $y, z : s \mapsto ()$, then the natural number n is encoded as follows:

$$[n] \stackrel{\text{def}}{=} (y,z)\overline{y}^n.\overline{z} : (s,s)$$

We want now to write an agent *Plus* capable of performing the sum of two numbers. Consider the process $\boldsymbol{\nu}\, x\,\big([n]\langle y,x\rangle \,|\, x.[m]\langle y,z\rangle\big)$: If we abstract from possible internal reductions, this behaves exactly like $[n+m]\langle y,z\rangle$. Accordingly, if X and Y are variables of the same sort as numerals, we can define

$$\mathit{Plus} \stackrel{\text{def}}{=} (X,Y,y,z)\big(\boldsymbol{\nu}\, x\,(X\langle y,x\rangle \mid x.Y\langle y,z\rangle)\big) : ((s,s),(s,s),s,s)$$

Plus is a higher-order abstraction, because it abstracts on agent-variables (to be precise *Plus* is a second-order abstraction). This is also indicated by the bracket nesting in the definition of *Plus*, which is greater than one. The machinery can be iterated, for instance by defining abstractions on variables of the same sort as *Plus* and so forth, progressively increasing the order of the resulting agents.

An adder which repeatedly takes two numbers at ports a_1, a_2 and outputs their sum at a_3 can be represented as:

$$\mathit{Add} \stackrel{\text{def}}{=} a_1(X).a_2(Y).\overline{a}_3\big\langle \mathit{Plus}\,\langle X,Y\rangle\big\rangle.\mathit{Add}$$

Encoding of the λ-calculus

The idea common to all various attempts at embedding λ-calculus into a process calculus [Bou89, Bou92, Tho90, Mil90] is to view functional application as a particular parallel combination of two agents, the function and its argument, and β-reduction as a particular case of communication. Our encoding into HOπ makes very transparent this idea. For convenience, a variable x of the λ-calculus is mapped into its upper-case variable X in HOπ. We take for granted the basic concepts of λ-calculus. As evaluation strategy, we adopt the one of Abramsky's *lazy λ-calculus* [Abr89] in which reductions occur only at the extreme left of a term.

$$\begin{aligned}
\mathcal{H}[\![\lambda x.M]\!] &\stackrel{\text{def}}{=} (p)p(X,q).\mathcal{H}[\![M]\!]\langle q\rangle \\
\mathcal{H}[\![x]\!] &\stackrel{\text{def}}{=} X \\
\mathcal{H}[\![MN]\!] &\stackrel{\text{def}}{=} (p)\boldsymbol{\nu}\, q\,\big(\mathcal{H}[\![M]\!]\langle q\rangle \mid \overline{q}\langle \mathcal{H}[\![N]\!], p\rangle\big)
\end{aligned}$$

If s is the sort of the names p, q, then the translation of a λ-term is an abstraction of sort (s). This abstracted name will be the only access to that agent and will be used to interact with the appropriate λ-term. Thus $\mathcal{H}[\![\lambda x.M]\!]\langle p\rangle$ receives at p its λ-argument and the name q which will give access to M. In the translation of application, the restriction on q prevents interferences from other processes.

The higher-order features of HOπ allow us a simpler encoding than Milner's into π-calculus [Mil90]. Indeed, there is a one-to-one correspondence between reductions in λ-terms and in their HOπ counterparts. Therefore, following Boudol's terminology [Bou89], we can claim that *lazy λ-calculus is a subcalculus of HOπ*.

Proposition 3.1 (operational correspondence for $\mathcal{H}$) *Let M and M' be closed λ-terms.*

1. *If $M \longrightarrow M'$, then $\mathcal{H}[\![M]\!]\langle p\rangle \longrightarrow \mathcal{H}[\![M']\!]\langle p\rangle$,*
2. *the converse, i.e. if $\mathcal{H}[\![M]\!]\langle p\rangle \longrightarrow Q$, then $\exists M'$ s.t. $Q \equiv \mathcal{H}[\![M']\!]\langle p\rangle$ and $M \longrightarrow M'$.*

Proof: Induction on the structure of M. $\square$

3.2 The syntax of HOπ

We shall maintain the notation introduced in Section 2. Furthermore, we need a set of *agent-variables*, ranged over by X, Y. There are two modifications to bring into the syntax of the π-calculus. First, variable application should be allowed too, so that an abstraction received as input can be provided with the appropriate arguments. Secondly, tuples in prefixing, applications and abstractions may also contain agents or agent-variables. To simplify the notation, in the grammar we use K to stand for an agent or a name and U to stand for a variable or a name.

$$\begin{array}{lcl} P & :: & \sum_{i\in I}\alpha_i.P_i \quad | \quad P_1 \mid P_2 \quad | \quad \boldsymbol{\nu}\, x\, P \quad | \; [x=y]P \quad | \; D\langle\widetilde{K}\rangle \quad | \quad X\langle\widetilde{K}\rangle \\ \alpha & :: & \overline{x}\langle\widetilde{K}\rangle \quad | \quad x(\widetilde{U}) \end{array}$$

Remember that K may be an agent; hence it may be a process, but also an abstraction of arbitrary high order. An *open* agent is an agent possibly containing free variables. It is worth pointing out that we do not lose expressiveness by having application only with variables and constants. In fact, every well-sorted expression $A\langle\widetilde{K}\rangle$ can be put into this form by "executing" the applications it contains; for instance from $\big((X)Y\langle X\rangle\big)\langle P\rangle$, we get $Y\langle P\rangle$. This makes the definition of substitution more elaborated but facilitates the proofs in the calculus.

3.3 Sorting and operational semantics

In the HOπ the need for sorts is even more compelling than in π-calculus. It is not only now a question of arities, but we have also to avoid any confusion between instantiation to names and to agents as well as instantiation to agents of different order.

W.r.t. the π-calculus sorts, the difference is that the sequences representing object sorts do not have to be made only of subject sorts; but object sorts themselves can appear too. Therefore, using *El* for a subject or an object sort, the grammar for sorts becomes:

$$\begin{array}{lcl} El & :: & s \quad | \quad S \\ S & :: & (\widetilde{El}) \quad | \quad () \end{array}$$

For each object sort S we suppose the existence of an infinite number of variables of sort S. The definition of well-sorted agent is easy and we leave it to the reader. The special case of second-order sorting $\{\text{Name} \mapsto (())\}$ corresponds to Thomsen's Plain CHOCS.

As an aside, let us mention an alternative notation for sorts which seems fairly effective in HOπ. Consider the abstraction $G \stackrel{\text{def}}{=} (X)F$, for $X : S'$, $F : S$. It represents a function which takes an argument of sort S' and gives back an argument of sort S. From a function-theoretic point of view, G has type $S' \longrightarrow S$. Following such intuition, we could explicitly introduce the arrow-sort and say that $G : S' \longrightarrow S$. For instance, for the agent *Plus* in the numeral example, we would have, for $S = s \longrightarrow s \longrightarrow ()$:

$$\mathit{Plus} : S \longrightarrow S \longrightarrow s \longrightarrow s \longrightarrow ()$$

or, using some "uncurrying":

$$\mathit{Plus} : S \times S \longrightarrow s \times s \longrightarrow ()$$

As regards the operational semantics, no modification is necessary in the rules for structural congruence. In the reduction rules only the COM rule is affected; since values exchanged do not have to be only names, it becomes:

$$\text{COM:}\ (\cdots + x(\widetilde{U}).P) \mid (\cdots + \overline{x}\langle\widetilde{K}\rangle) \longrightarrow P\{\widetilde{K}/\widetilde{U}\} \mid Q$$

4 Behavioural equivalence

We concentrate on bisimulation, probably the most studied behavioural equivalence in process algebra. Both in π-calculus and in HOπ we adopt the congruence induced by *barbed bisimulation* [MS92, San92]. There are three main reasons for the interest in barbed bisimulation.

1. It allows us to recover from reduction semantics the well-known bisimulation-based equivalences which are defined on the labelled transition system;
2. It can be defined uniformly in different calculi and thus provides us with a fundamental tool for comparing them, the kind of issue on which this paper is mainly concerned;
3. It allows us to obtain a natural bisimulation equivalence for higher-order calculi.

Some further comment on (3) is worthwhile. The definition of a natural bisimulation equivalence in a higher-order calculus is not straightforward. The habitual definition of bisimulation, based on the labelled transition system, requires that an action be matched by another only if they have *identical* labels. This also works for π-calculus if we concede α-convertibility. But it does *not* in a higher-order calculus. Obvious algebraic laws such as the commutativity of parallel composition are lost: For instance, the processes $\overline{a}\langle P|Q\rangle.0$ and $\overline{a}\langle Q \mid P\rangle.0$ are distinguished since the agents emitted in their respective outputs are syntactically different.

The approach taken by Astesiano, Boudol and Thomsen [AG88, Bou89, Tho90], is to require *bisimilarity* rather than *identity* of the processes emitted in a higher-order output action. But this gives rise to counterintuitive equalities when restriction is a formal binder. The problems are due to the fact that the object part and the continuation are examined separately, thus preventing a satisfactory treatment of the channels private to the two. See [San92] for precise examples.

Barbed bisimulation focuses on the reduction or interaction relation, a concept common to different calculi. It goes a little further though, since the reduction relation by itself is not enough to yield the desired discriminanting power. The choice in [San92] was to introduce, for each name a, an *observation predicate* $\downarrow_a$ which detects the possibility of performing a communication with the environment along a. A simple syntactic condition is enough to know whether $P \downarrow_a$ holds: there must be in P a prefix $a(\widetilde{x})$ or $\overline{a}\langle\widetilde{x}\rangle$ which is not underneath another prefix and not in the scope of a restriction on a. For example, If P is $(\boldsymbol{\nu}\, c)\big(\overline{c}\,.\,b \mid a\,.\,d\big)$, then $P \downarrow_a$, but not $P \downarrow_c$, $P \downarrow_b$ or $P \downarrow_d$.

Definition 4.1 Strong barbed bisimulation, *written* $\stackrel{\bullet}{\sim}$, *is the largest symmetrical relation on the class of processes of the language s.t.* $P \stackrel{\bullet}{\sim} Q$ *implies:*

1. *whenever* $P \longrightarrow P'$ *then* $Q \longrightarrow Q'$ *and* $P' \stackrel{\bullet}{\sim} Q'$*;*
2. *for each channel* a*, if* $P \downarrow_a$ *then* $Q \downarrow_a$*.* $\square$

The weak version of the equivalence, in which one abstracts away from the length of the reductions in two matching actions, is obtained in the standard way: If $\Longrightarrow$ is the reflexive and transitive closure of $\longrightarrow$ and $\Downarrow_a$ is $\Longrightarrow\downarrow_a$ (the composition of the two relations), then *weak barbed bisimulation*, written $\stackrel{\bullet}{\approx}$, is defined by replacing the transition $Q \longrightarrow Q'$ with $Q \Longrightarrow Q'$ and the predicate $Q \downarrow_a$ with $Q \Downarrow_a$.

By itself, barbed bisimulation is rather weak (it is not even preserved by parallel composition). By parametrisation over contexts, we get a finer relation.

Definition 4.2 *Two processes* P *and* Q *are* strong barbed-congruent*, written* $P \sim^{\mathrm{c}} Q$*, if for each context* $C[\cdot]$*, it holds that* $C[P] \stackrel{\bullet}{\sim} C[Q]$ *.* $\square$

To obtain *weak barbed congruence*, written $\approx^{\mathrm{c}}$, replace $\stackrel{\bullet}{\sim}$ with $\stackrel{\bullet}{\approx}$. The reader familiar with process algebra might remark that most of the common weak bisimulations of labelled transition systems are not preserved by *dynamic* operators, i.e. operators like sum and prefix which are discharged when an action is produced. To recover them, one can parametrise barbed bisimulation over the subclass of contexts which are built by composing the hole $[\cdot]$ and the processes by means of only non-dynamic operators. The resulting equivalence is called *barbed equivalence.*

To test the discriminatory power of barbed bisimulation, we have proved in [San92] that in the strong and in the weak case barbed equivalence and congruence coincide in CCS and π-calculus with the ordinary bisimulation-based equivalences. We have also obtained fairly simple direct characterisations of barbed equivalence and congruence in HOπ. In this paper, we shall only deal explicitly with weak barbed congruence; however all results stated hold for weak barbed equivalence too.

5 The compilation from HOπ to π-calculus

We present the compilation into π-calculus (in this and in the next section) on the subclass of the HOπ agents with the following two restrictions: First, *one* only value — a name or an abstraction — can be transmitted, and only *unary* abstractions are permitted. Second, the agent definitions use a *finite* number of constants; this allow us to have replication in place of constants since, as mentioned in Section 2.1, a finite number of constants can be coded up with replication. It is important to stress the different significance of the two simplifications: The former is purely to make more readable the definition of the compilation and of the operational correspondence for it — the generalisation to the calculus with arbitrary arities does not give any problem. By contrast, the latter does intend to exclude some agents, namely those defined with an infinite number of constants; we adopted replication because it is useful in the definition of the compilation and facilitates the reasoning by structural induction in the proofs. We keep '$\stackrel{\mathrm{def}}{=}$' as abbreviation mechanism, to assign names to expressions to which we want to refer later.

We use $P\{m := F\}$ to stand for $\boldsymbol{\nu}\, m\,(P \mid\, !\, m(U).F\langle U\rangle)$, where U is a name or a variable, depending upon the sort. We chose curly brackets for this notation because under a certain condition on the use

$$
\begin{array}{rcl}
\mathcal{C}\llbracket X \rrbracket & \overset{\text{def}}{=} & \begin{cases} \mathcal{C}\llbracket (Y)X\langle Y\rangle \rrbracket & \text{if } X \text{ is a higher-order abstraction} \\ \mathcal{C}\llbracket (a)X\langle a\rangle \rrbracket & \text{otherwise} \end{cases} \\
\mathcal{C}\llbracket \alpha.P \rrbracket & \overset{\text{def}}{=} & \begin{cases} (\overline{a}\langle m\rangle.\mathcal{C}\llbracket P \rrbracket)\,\{m := \mathcal{C}\llbracket F \rrbracket\} & \text{if } \alpha = \overline{a}\langle F\rangle \\ a(x).\mathcal{C}\llbracket P \rrbracket & \text{if } \alpha = a(X) \\ \alpha.\mathcal{C}\llbracket P \rrbracket & \text{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{rclrclrcl}
\mathcal{C}\llbracket X\langle F\rangle \rrbracket & \overset{\text{def}}{=} & (\overline{x}\langle m\rangle.0)\,\{m := \mathcal{C}\llbracket F \rrbracket\} & \mathcal{C}\llbracket X\langle b\rangle \rrbracket & \overset{\text{def}}{=} & \overline{x}\langle b\rangle.0 & \mathcal{C}\llbracket\, !\, P \rrbracket & \overset{\text{def}}{=} & !\,\mathcal{C}\llbracket P \rrbracket \\
\mathcal{C}\llbracket P \mid Q \rrbracket & \overset{\text{def}}{=} & \mathcal{C}\llbracket P \rrbracket \mid \mathcal{C}\llbracket Q \rrbracket & \mathcal{C}\llbracket P + Q \rrbracket & \overset{\text{def}}{=} & \mathcal{C}\llbracket P \rrbracket + \mathcal{C}\llbracket Q \rrbracket & \mathcal{C}\llbracket \boldsymbol{\nu}\, a\, P \rrbracket & \overset{\text{def}}{=} & \boldsymbol{\nu}\, a\, \mathcal{C}\llbracket P \rrbracket \\
\mathcal{C}\llbracket\, [a = b]P\, \rrbracket & \overset{\text{def}}{=} & [a = b]\, \mathcal{C}\llbracket P \rrbracket & \mathcal{C}\llbracket (X)P \rrbracket & \overset{\text{def}}{=} & (x)\mathcal{C}\llbracket P \rrbracket & \mathcal{C}\llbracket (a)P \rrbracket & \overset{\text{def}}{=} & (a)\mathcal{C}\llbracket P \rrbracket
\end{array}
$$

Table 1: The compilation $\mathcal{C}$

of m in P and F, m acts in P as a pointer to F and $\{m := F\}$ as a "local environment" for P. We shall allow ourselves a free use of this abbreviation. Formally, since only guarded sums are admitted in the language, it is not legal to use it in a context like $[\cdot] + Q$. However there are obvious transformations which convert any misuse of this into a correct process expression, and we leave them implicit.

Intuitively, the compilation $\mathcal{C}$ replaces the communication of an agent with the communication of the *access* to that agent. Thus $P_1 \overset{\text{def}}{=} \overline{a}\langle F\rangle.Q$ is replaced by $P_2 \overset{\text{def}}{=} (\overline{a}\langle m\rangle.Q)\,\{m := F\}$. Whereas an agent interacting with P_1 may use F directly with, say, argument b, an agent interacting with P_2 uses m to activate F and provide it with the argument b. The name m is called *name-trigger*.

The compilation has also to modify the sorting Ob. We suppose that Ob is *downward-closed*, that is if $s \mapsto (S) \in Ob$, then s' exists s.t. $s' \mapsto S \in Ob$. If Ob is not already downward-closed, then it can easily be extended to make it so. The *downward-closed* property is used to select a subsorting SOb of Ob from which to draw the name-triggers. There might be different ways of defining SOb: Our sole requirement is that SOb has one and only one subject sort with object sort S, for each object sort S which appears in Ob; we use SOb_S to denote this subject sort. The compilation shall replace agents of sort S with names of sort SOb_S; indeed, if $A : S = (El)$, then a trigger which has to convey the argument for A carries values of sort El, i.e. its object sort is precisely (El). Therefore we have:

$$\mathcal{C}\llbracket Ob \rrbracket = \{s \mapsto (s') \in Ob\} \bigcup \{s \mapsto (SOb_S) \ :\ s \mapsto (S) \in Ob\}.$$

The behaviour of $\mathcal{C}$ on agents is described in Table 1. To respect the definition on the sorting, we assume that if X is a variable of sort S, then its lower case letter x is a name from SOb_S. Moreover, both this name x and the name-trigger m are taken to be fresh, i.e. not occurring in the source agent.

Besides the above sketched treatment of higher-order outputs, the other interesting rules of Table 1 are those for application and for variable. Consider the application $X\langle K\rangle$: When X is instantiated to an agent G, it becomes $G\langle K\rangle$. Translating it, we expect to receive just a name-trigger to G, and we are expected to use it to activate G with its argument K. This is legal when K is a name (and leads to our rule for for first-order application), but it is not when K is an agent, since we cannot pass it at first order. As in the rule for outputs, this is resolved by sending a name-trigger for K. In the rule for higher-order variable (and for uniformity, also in the rule for first-order variable) an η-conversion is employed. This is to make explicit all possible applications and hence introduce all necessary name-triggers; we shall see later, discussing tentative optimisations for $\mathcal{C}$, that the use of full triggered forms is necessary to get soundness. Note that in this rule the distinction between first-order and higher-order variables introduces a dependency from the sorting. Moreover the latter, for which the sort of Y must be "smaller" than the sort of X, guarantees that $\mathcal{C}$ is well-defined. The compilation acts as an homomorphism in all other cases.

Let us illustrate how $\mathcal{C}$ works on reductions. There are two dimensions at which the number of interactions is expanded. One is *horizontal.* If a transmitted agent F is used by its recipient n times, n interactions are required at first-order to activate the copies of F.

Example 5.1 Let $P \stackrel{\mathrm{def}}{=} \overline{a}\langle F\rangle.Q \mid a(Y).(Y\langle b\rangle \mid Y\langle c\rangle)$. Then $P \longrightarrow P' \stackrel{\mathrm{def}}{=} Q \mid F\langle b\rangle \mid F\langle c\rangle$. In $\mathcal{C}[\![P]\!]$ this is simulated using three reductions:

$$\begin{array}{rcl}
\mathcal{C}[\![P]\!] & \stackrel{\mathrm{def}}{=} & (\overline{a}\langle m\rangle.\mathcal{C}[\![Q]\!])\,\{m := \mathcal{C}[\![F]\!]\} \mid a(y).(\overline{y}\langle b\rangle \mid \overline{y}\langle c\rangle) \\
& \longrightarrow & \big(\mathcal{C}[\![Q]\!] \mid \overline{m}\langle b\rangle \mid \overline{m}\langle c\rangle\big)\,\{m := \mathcal{C}[\![F]\!]\} \\
& \longrightarrow\longrightarrow & \big(\mathcal{C}[\![Q]\!] \mid \mathcal{C}[\![F]\!]\langle b\rangle \mid \mathcal{C}[\![F]\!]\langle c\rangle\big)\,\{m := \mathcal{C}[\![F]\!]\} \\
& \sim^{\mathrm{c}} & \mathcal{C}[\![Q]\!] \mid \mathcal{C}[\![F]\!]\langle b\rangle \mid \mathcal{C}[\![F]\!]\langle c\rangle \;=\; \mathcal{C}[\![P']\!]
\end{array}$$

where the last equality holds because m is not free in the body. □

The other way to add interactions is *vertical* and takes its significance from the ω-order nature of HOπ. It arises with higher-order abstractions when, after the abstraction itself, one has also to trigger its arguments. This may give rise to interesting chains of activations. To see a simple case, take $P \stackrel{\mathrm{def}}{=} \overline{a}\langle G\rangle \mid a(Y).Y\langle F\rangle$, where G is a second-order abstraction. We have $P \longrightarrow G\langle F\rangle$. To achieve the same effect $\mathcal{C}[\![P]\!]$ requires two further interactions, one to activate a copy of G and another to activate a copy of F.

6 Correctness of the compilation

Before tackling the question of the semantic correctness of the compilation $\mathcal{C}$, we have to check that its definition is syntactically meaningful by ensuring that it returns first-order agents and that there is agreement between the definitions of $\mathcal{C}$ on sorts and agents. The former is straightforward; the latter holds too because all new names which are introduced (in the rule for application) or whose object part is modified (in the rule for prefixing), respect the sorting $\mathcal{C}[\![Ob]\!]$.

Theorem 6.1 *For each open agent A, it holds that $\mathcal{C}[\![A]\!]$ is a first-order agent and well-sorted for $\mathcal{C}[\![Ob]\!]$.* □

By contrast, the proof that $\mathcal{C}$ is faithful w.r.t. $\approx^{\mathrm{c}}$ is not at all trivial. We limit ourselves to summarising the schema used in [San92], to which we refer for the details. The compilation $\mathcal{C}$ is derived into two steps. The first is a mapping $\mathcal{T}$ which transforms an agent into a *triggered agent.* These are "normalised" HOπ agents in which every agent emitted in an output or "expected" in an input has a very simplified form and the same functionality of name-triggers. Thus higher-order communications have become homogeneous and have lost all their potential richness and variety. This greatly simplifies the reasoning over agents. Triggered agents have a few interesting properties, in particular a quite simple characterisation of barbed congruence.

The agent $\mathcal{T}[\![A]\!]$ already has the same structure as $\mathcal{C}[\![A]\!]$. The real difference is that $\mathcal{T}$ is an *endo*-encoding, that is, it remains within the same calculus. This facilitates the correctness proof and prepares the way for the next step, the mapping $\mathcal{F}$, which leads us down to first order. Syntactically, $\mathcal{F}$ is a fairly simple transformation. But semantically it is more delicate because it modifies the object sort of names. The correctness proofs of $\mathcal{T}$ and $\mathcal{F}$ are obtained using direct characterisations of barbed congruence on labelled transition systems plus the "local environment" properties for $\{m := F\}$.

Theorem 6.2 (full abstraction for $\mathcal{C}$) *For each pair of open agents A_1 and A_2,*

$$A_1 \approx^{c} A_2 \text{ iff } \mathcal{C}[\![A_1]\!] \approx^{c} \mathcal{C}[\![A_2]\!] \qquad \Box$$

The definition of barbed congruence on abstractions and open agents is given in the expected way, by requiring instantiation of variables and of abstracted names with all agents or names of the right sort. Thus $(X)P \approx^{c} (X)Q$ if for each F of the same sort as X, $P\{F/X\} \approx^{c} Q\{F/X\}$.

A few considerations to emphasise the faithfulness of $\mathcal{C}$ are worthwhile. By itself, Theorem 6.2 does not reveal anything about how closely $\mathcal{C}[\![P]\!]$ simulates P; actually, nothing prevents us from obtaining the same result with a very bizarre encoding! First of all, let us show the operational correspondence existing between P and $\mathcal{C}[\![P]\!]$. We only look here at reductions; but a similar result holds for the visible actions of the labelled transition system [San92].

In clause (1) below, an interaction is *first order* (resp. *higher order*) if the transmitted value is a name (resp. an agent). In clause (2) an interaction is *converted* if it comes from a communication along a name whose object sort has been modified by $\mathcal{C}$, i.e. a name which carries agents in Ob. In clauses (b), F represents the abstraction which is exchanged in P and m the trigger which is exchanged in $\mathcal{C}[\![P]\!]$.

Proposition 6.3 (operational correspondence for $\mathcal{C}$) *Suppose m not free in P and $\mathcal{C}[\![P]\!]$:*

1. *(a) If $P \longrightarrow P'$ is a first-order interaction, then $\mathcal{C}[\![P]\!] \longrightarrow \mathcal{C}[\![P']\!]$;*
   (b) If $P \longrightarrow P'$ is a higher-order interaction, then there are $\widetilde{b}, G, F$ s.t.
   $P' \equiv \boldsymbol{\nu}\,\widetilde{b}\,(G\langle F\rangle)$ and $\mathcal{C}[\![P]\!] \longrightarrow \boldsymbol{\nu}\,\widetilde{b}\,\big(\mathcal{C}[\![G]\!]\langle m\rangle\,\{m := \mathcal{C}[\![F]\!]\}\big)$.
2. *the converse of (1), i.e:*
   (a) If $\mathcal{C}[\![P]\!] \longrightarrow P''$ is a non-converted interaction, then P' exists s.t.
   $P \longrightarrow P'$, and $P'' = \mathcal{C}[\![P']\!]$;
   (b) If $\mathcal{C}[\![P]\!] \longrightarrow P''$ is a converted interaction, then there are $\widetilde{b}, G, F$ s.t.
   $P \longrightarrow \boldsymbol{\nu}\,\widetilde{b}\,(G\langle F\rangle)$ and $P'' \equiv \boldsymbol{\nu}\,\widetilde{b}\,\big(\mathcal{C}[\![G]\!]\langle m\rangle\,\{m := \mathcal{C}[\![F]\!]\}\big)$. $\Box$

Secondly, let us point out that by definitions of $\mathcal{C}$, if P is a first-order process then it is not modified by $\mathcal{C}$, i.e.

$$\mathcal{C}[\![P]\!] = P$$

Thirdly, suppose that P is an HOπ process which can only perform first-order actions. This does not imply that P is also a π-calculus process, as *internally* P could perform communications of agents. But if we relax the definition of well-sorted agent, then we can think of comparing directly P with $\mathcal{C}[\![P]\!]$, and we would get

$$\mathcal{C}[\![P]\!] \approx^{c} P$$

Optimisations?

There are critical points in the definition of $\mathcal{C}$ on agents which is worth indicating. We doubt that non-trivial improvements are possible without loosing full abstraction. The first optimisation one might be tempted of, is on the output of agent-variables, defining

$$\mathcal{C}[\![\overline{a}\langle X\rangle.Q]\!] \stackrel{\text{def}}{=} \overline{a}\langle x\rangle.\mathcal{C}[\![Q]\!] \qquad (*)$$

After all, since all communications of higher-order values are transformed by $\mathcal{C}$ into communications of name-triggers, we already know that x will always be instantiated with one of these; then it seems that the original rule, introducing another name-trigger m is just adding a further level of indirection. But rule $(*)$ in general is *not* sound. Consider in fact

$$\begin{array}{rcl} P & \stackrel{\text{def}}{=} & \boldsymbol{\nu}\, a\,\big(\overline{a}\langle F\rangle \mid a(X).\overline{b}\langle X\rangle.\overline{b}\langle X\rangle\big) \\ Q & \stackrel{\text{def}}{=} & \boldsymbol{\nu}\, a\,\big(\overline{a}\langle F\rangle \mid a(X).\overline{b}\langle F\rangle.\overline{b}\langle F\rangle\big) \end{array}$$

Clearly P and Q are equivalent (they are strong barbed-congruent). But adopting rule $(*)$ their translation are not! In fact we have

$$\begin{array}{rcl} \mathcal{C}[\![P]\!] & \stackrel{\text{def}}{=} & \boldsymbol{\nu}\, a\,\big(\overline{a}\langle m\rangle\,\{m := F\} \mid a(x).\overline{b}\langle x\rangle.\overline{b}\langle x\rangle\big) \\ \mathcal{C}[\![Q]\!] & \stackrel{\text{def}}{=} & \boldsymbol{\nu}\, a\,\big(\overline{a}\langle m\rangle\,\{m := F\} \mid a(x).(\overline{b}\langle m_1\rangle.\overline{b}\langle m_2\rangle\,\{m_2 := F\})\,\{m_1 := F\}\big) \end{array}$$

which can be distinguished since after the initial interaction, $\mathcal{C}[\![Q]\!]$ can perform two outputs of private (and hence distinct) names at b, whereas in $\mathcal{C}[\![P]\!]$ the two outputs at b communicate the same name. For similar reasons, the optimisation

$$\mathcal{C}[\![X\langle Y\rangle]\!] \stackrel{\text{def}}{=} \overline{x}\langle y\rangle.0$$

is not sound. For this, take the equivalent HOπ processes $c(X).\boldsymbol{\nu}\, a\,\big(a(Y).(X\langle Y\rangle \mid X\langle Y\rangle) \mid \overline{a}\langle F\rangle\big)$ and $c(X).\boldsymbol{\nu}\, a\,\big(a(Y).(X\langle F\rangle|X\langle F\rangle)|\overline{a}\langle F\rangle\big)$. Their π-calculus translations would be distinguished by reasoning similarly as above.

There are however situations when the above optimisations are indeed sound; we leave for future work more precise answers to this issue.

7 Some uses of the compilation

We have given the faithfulness of $\mathcal{C}$ only w.r.t. barbed congruence. But we believe that $\mathcal{C}$ respects most of the well-known weak equivalences which admit a uniform definition over higher-order and first-order calculi, such as *testing equivalence* [DH84], or *refusal equivalence* [Phi87]. The reason is the close operational correspondence between source and target agents of $\mathcal{C}$.

Indeed $\mathcal{C}$ might even be used to *define* equivalences in HOπ. Take for instance *trace semantics* as defined in [Hoa85], or *causal bisimulation* [DD89]. These were originally proposed for calculi without explicit mobility, but they can easily be adapted to π-calculus. More delicate is their extension to a higher-order calculus; as usual, it is not obvious which condition to impose on higher-order outputs. However, if P and Q are HOπ processes and $\ll\gg$ means weak trace equivalence or weak causal bisimulation, we might simply define:

$$P \ll\gg Q \text{ if } \mathcal{C}[\![P]\!] \ll\gg \mathcal{C}[\![Q]\!]$$

and then look for a direct characterisation (i.e. not mentioning the encoding) of $\ll\gg$.

The study of the λ-calculus encodings

A nice application of Theorem 6.2 comes from the study of the λ-calculus encodings. We presented the encoding into HOπ in Section 3 (for the lazy λ-calculus). By applying compilation $\mathcal{C}$ we can turn this into a π-calculus encoding. The outcome is precisely Milner's [Mil90]. This commutativity strengthens the naturalness of the translations involved. It means also that we can infer for Milner's encoding all results we can prove working with HOπ. We have used this in [San92] to show that both encodings give rise to a λ-model in which a weak form of extensionality holds, and to obtain a direct characterisation of the equivalence induced on the λ-terms by the behavioural equivalence adopted on the process terms.

HOπ has been useful also to understand Milner's encoding of call-by-value λ-calculus into π-calculus. In his original work [Mil90] two encodings were given, and it was not obvious which one should be preferred. When we have tried to see if they factor through $\mathcal{C}$ and an encoding into HOπ, the relationship between the two became clear: We could pass from the first to the second encoding by using the "false" optimisation $(*)$ shown in the previous section. Building on this and on the non-soundness of rule $(*)$, we have been able to prove that β reduction is not valid for the second encoding (i.e. the encodings of a term and of a β-derivative of its are not equivalent), which fairly reduces its importance. The counterexample is rather sophisticated and we doubt we could have obtained it without going through HOπ.

8 Related work

The first attempt at encoding a higher-order process calculus into π-calculus was made by Thomsen. For this, he used Plain CHOCS (PC) which, as mentioned in Section 3.3, corresponds to the subcalculus of HOπ which has the sorting $\{Name \mapsto (())\}$. Thomsen's study acted as stimulus and basis for our work. When applied to PC, our compilation $\mathcal{C}$ coincides with Thomsen's translation and in this sense can be seen as an extension of it. Recently, Thomsen's work has been resumed by Amadio [Ama92], which adopts a different equivalence on PC. Our analysis, however, strengthens and completes both Amadio's and Thomsen's in various aspects. First, PC is a special case of a second-order language, whereas HOπ is of ω order. Second, to establish an operational correspondence between PC processes and their π-calculus encodings, they have to modify the semantics of the calculi; this seems rather arbitrary and obscures the meaning of the results obtained. Thirdly they do not get a full abstraction result, as we did in Theorem 6.2.

9 Directions for future research

Study of the expressiveness of π-calculus

Our study on the translation of HOπ and λ-calculus into π-calculus may be seen as just one aspect of a more general investigation into the expressiveness of π-calculus. For instance, it is not clear to us at what extent the results for compilation $\mathcal{C}$ depend upon the choice of the operators in HOπ and π-calculus. We think that in general we cannot remove the restriction on guardness for the sum. But we do not see this as a strong limitation. Besides yielding a simpler reduction semantics, guarded sums are easier to implement. Furthermore, in process algebras guarded sums are usually necessary to make a number of well-known equivalences, congruences w.r.t. the sum operator. Last but not least, they are justified by practical applications, which show that they give all needed expressiveness.

In general, it seems that the problems for the compilation mainly arise with *dynamic* operators, to which sum belongs (another example of dynamic operator is Lotos's disabling [BB89]).

Adding data to HOπ

The study conducted with the λ-calculus illustrates the usefulness of the abstraction power of HOπ w.r.t. the π-calculus. Such abstraction power could be increased by adding some (simple) form of data, like integers, booleans, or lists. Accordingly, the format of object sorts should be enriched to allow for data communications. Data should be taken into account also in the definition of the equivalences. The interesting thing is that $\mathcal{C}$ is easily generalisable to the extended HOπ, since data can be encoded in the π-calculus ([Mil91, Wal91]). Then the proof that the faithfulness of $\mathcal{C}$ is maintained would give us confidence that what we are developing is sensible.

Semantics of object-oriented languages

Two interesting approaches to the denotational semantics of parallel object-oriented languages are exhibited in [AdBKR89] and [Wal90], using metric spaces and by translation into π-calculus, respectively. In both cases the source language is POOL [Ame89]. Let us point out here their weaknesses. In the former, a heavy mathematical machinery, needed to ensure the well-definedness of the semantics. In the latter, the "flatness": There is no concept of type to give an overall idea of the use and the purpose of the various agents defined; and since π-calculus is "low-level", the protocols implementing interactions among different components sometimes are burdensome.

We would like to see if it is possible to gain some benefit by using the HOπ as target language. Higher-order sorts would play the role of types in [AdBKR89]. The theory developed for the HOπ could be employed to reason on the semantic objects. The representation should be more succinct and readable than the one in [Wal90], even more if data are added to HOπ as suggested above. As for λ-calculus, using $\mathcal{C}$ the two translations could be compared to see if and where they are different.

References

[Abr89] S. Abramsky. The lazy lambda calculus. In D. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1989.

[AdBKR89] P. America, J. de Bakker, J. Kok, and J. Rutten. Denotational semantics of a parallel object-oriented language. *Information and Computation*, 83(2), 1989.

[AG88] E. Astesiano and A. Giovini. Generalized bisimulation in relational specifications. In *STACS '88*, volume 294 of *Lecture Notes in Computer Science*, pages 207–226. Springer Verlag, 1988.

[Ama92] R. Amadio. A uniform presentation of CHOCS and π-calculus. Rapport de recherche 1726, INRIA-Lorraine, Nancy, 1992.

[Ame89] P. America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4):366–411, 1989.

[BB89] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. In P.H.J. van Eijk, C.A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*. North Holland, 1989.

[BB90] G. Berry and G. Boudol. The chemical abstract machine. In *17th POPL*, 1990.

[Bou89] G. Boudol. Towards a lambda calculus for concurrent and communicating systems. In *TAPSOFT '89*, volume 351 of *Lecture Notes in Computer Science*, pages 149–161, 1989.

[Bou92] G. Boudol. Asynchrony and the π-calculus. Submitted for publication, 1992.

[DD89] P. Degano and P. Darondeau. Causal trees. In *15th ICALP*, volume 372 of *Lecture Notes in Computer Science*, pages 234–248. Springer Verlag, 1989.

[DH84] R. De Nicola and R. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mil90] R. Milner. Functions as processes. Research Report 1154, INRIA, Sofia Antipolis, 1990. Final version in *Journal of Mathem. Structures in Computer Science* 2(2):119–141, 1992.

[Mil91] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, LFCS, Dept. of Comp. Sci., Edinburgh Univ., October 1991. To appear in the *Proceedings of the International Summer School on Logic and Algebra of Specification*, Marktoberdorf, August 1991.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, (Parts I and II). *Information and Computation*, 100:1–77, 1992.

[MS92] R. Milner and D. Sangiorgi. Barbed bisimulation. In W. Kuich, editor, *19th ICALP*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695. Springer Verlag, 1992.

[Phi87] I.C.C. Phillips. Refusal testings. *Theoretical Computer Science*, 50:241–284, 1987.

[San92] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, Department of Computer Science, University of Edinburgh, 1992. To appear.

[Tho90] B. Thomsen. *Calculi for Higher Order Communicating Systems*. PhD thesis, Department of Computing, Imperial College, 1990.

[Wal90] D. Walker. π-calculus semantics of object-oriented programming languages. ECS-LFCS-90-122, LFCS, Dept. of Comp. Sci., Edinburgh Univ., 1990. Also in *Proc. Conference on Theoretical Aspects of Computer Software*, Tohoku University, Japan, Sept. 1991.

[Wal91] D. Walker. Some results on the π-calculus. In A. Yonezawa and T. Ito, editors, *Concurrency: Theory, Language, and Architecture*, volume 491 of *Lecture Notes in Computer Science*, pages 21–35. Springer Verlag, 1991.