A brief survey of the theory of the Pi-calculus.

Daniel Hirschkoff

Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON
n° 5668

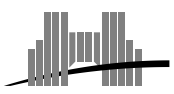
CNRS CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE

A brief survey of the theory of the π-calculus

Daniel Hirschkoff

February 2003

Research Report N° 2003-13



École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`

INRIA

A brief survey of the theory of the π-calculus

Daniel Hirschkoff

February 2003

Abstract

This document collects some important results about the theory of Milner's π-calculus and related formalisms. We present the syntax and semantics of a monadic calculus, and discuss type systems and the most commonly used notions of behavioural equivalences. π-calculus dialects are also briefly introduced, as well as several encodings of the λ-calculus.

Nota: These notes have been used for a course in the winter school on theoretical computer science organised by the Franco-Italian University in Aussois between the 29th of january and the 1st of february, 2003.

Keywords: Process algebra, mobility, name passing, π-calculus, λ-calculus, type system, behavioural equivalence

Résumé

Ce document rassemble quelques résultats essentiels de la théorie du π-calcul de Milner. Nous présentons la syntaxe et la sémantique opérationnelle d'un calcul monadique, et discutons différentes sortes de systèmes de types et d'équivalences comportementales pour le π-calcul. Nous nous intéressons également au problème de l'encodage du λ-calcul, et présentons brièvement différents dialectes du π-calcul.

Note: Ces notes ont servi de support à un cours dans le cadre de l'école d'hiver d'informatique théorique de l'Université Franco-Italienne qui s'est tenue à Aussois entre le 29 janvier et le 1er février 2003.

Mots-clés: Algèbre de processus, mobilité, passage de noms, π-calcul, λ-calcul, système de type, équivalence comportementale

1 π-Calculi

1.1 The monadic π-calculus

1.1.1 Syntax

Our referential calculus will be the *monadic synchronous π-calculus,* which is built upon an infinite set of *names* (also called *channels* or *links*; sometimes also *ports*), over which we range with small letters: $a, b, \ldots, m, n, o, p, q, \ldots, x, y, \ldots$.

Terms of the π-calculus are also called *processes*, and are ranged over with big letters: $A, B, \ldots, P, Q, R, S, \ldots$. They designate a state in the evolution of a system, or rather of a subsystem, in the sense that a π-calculus term is regarded as interacting with a *context*.

$n, m, \ldots$	***names***	P	::=		***processes***
				$\mathbf{0}$	inactive process
			$\mid$	$P_1 \mid P_2$	parallel composition
			$\mid$	$!P$	replication
			$\mid$	$(\boldsymbol{\nu} n)\, P$	restriction
			$\mid$	$n(m).P$	input prefix
			$\mid$	$\overline{n}\langle m\rangle.P$	output prefix

Figure 1: Syntax of the monadic π-calculus

Bound and free names of a process P (written respectively $\mathrm{bn}(P)$ and $\mathrm{fn}(P)$) are defined by saying that name n is bound in $a(n).Q$ and $(\boldsymbol{\nu} n)\, Q$, other kinds of occurrences defining free names.

$\mathbf{0}$ is the terminated process, that does nothing. The parallel composition of processes P_1 and P_2 is written $P_1 \mid P_2$ and lets P_1 and P_2 interact together (and with the context). The operator of *replication* can be thought of as making an unbounded number of copies of a process available: $!P$ stands for $P \mid P \mid \ldots$. *Restriction*, written $\boldsymbol{\nu}$, has the effect of making the usage of a name private to a process; alternatively, $(\boldsymbol{\nu} n)\, P$ can be seen as the process allocating a *fresh* name (different from all names possibly used by any other process in the context) and proceeding as P. Input ($n(m)$) and output ($\overline{n}\langle m\rangle$) prefixes are the elementary constituents of interaction, as expressed by the central reduction rule:

$$\overline{n}\langle a\rangle.P \mid n(b).Q \quad \longrightarrow \quad P \mid Q_{\{b \leftarrow a\}}$$

Here $Q_{\{b \leftarrow a\}}$ stands for the capture-avoiding substitution of name b by name a in Q.

Both kinds of prefix are made of a *subject*, the name over which communication happens (n in the example above), and an *object*, which is the name being transmitted.

Variables. It can be remarked that there is no distinct syntactic notion of variable in the π-calculus: in $n(m).P$, m is a name used as a placeholder for the name to be transmitted upon reception on n. To enhance readability, we shall

$$P \mid \mathbf{0} \equiv P \qquad P \mid Q \equiv Q \mid P \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R$$
$$(\boldsymbol{\nu} n)\, \mathbf{0} \equiv \mathbf{0} \qquad (\boldsymbol{\nu} n)(\boldsymbol{\nu} m)\, P \equiv (\boldsymbol{\nu} m)(\boldsymbol{\nu} n)\, P$$
$$(\boldsymbol{\nu} n)\, (P \mid Q) \equiv P \mid (\boldsymbol{\nu} n)\, Q \quad \text{if } n \notin \mathrm{fn}(P)$$
$$!P \equiv !P \mid P \qquad !\mathbf{0} \equiv \mathbf{0} \qquad !!P \equiv !P \qquad !(P \mid Q) \equiv !P \mid !Q$$

Figure 2: Structural congruence

$$\overline{n}\langle a \rangle.P \mid n(b).Q \longrightarrow P \mid Q_{\{b \leftarrow a\}}$$
$$\frac{P \longrightarrow P'}{(\boldsymbol{\nu} n)\, P \longrightarrow (\boldsymbol{\nu} n)\, P'} \qquad \frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q}$$
$$\frac{P \longrightarrow P' \quad P \equiv Q \quad P' \equiv Q'}{Q \longrightarrow Q'}$$

Figure 3: Reduction

often use names x, y, z to designate the object of receptions, and prefer $n(x).P$ to $n(m).P$ (although this is only a matter of α-conversion). Note also that some authors prefer to explicitely introduce two different sets of names and variables, which turns out to be more tractable in some situations.

Frequently used notations. Some common abbreviations are used when writing π-calculus processes. A trailing inactive process is often left implicit (thus writing e.g. $a(x).\overline{b}\langle x \rangle$ instead of $a(x).\overline{b}\langle x \rangle.\mathbf{0}$). Associativity of parallel composition (see below) will allow us to write processes of the form $P_1 \mid \ldots \mid P_k$ without parentheses. Consecutive restrictions may be grouped using the *tuple* notation: we shall write $\tilde{n} = (n_1, \ldots, n_k)$, and $(\boldsymbol{\nu} \tilde{n})\, P$ stands for $(\boldsymbol{\nu} n_1) \ldots (\boldsymbol{\nu} n_k)\, P$. We shall also write $a(\_).P$ for an input process in which the received value is not used in the continuation P.

1.1.2 Reduction-based operational semantics

The reduction-based presentation of the operational semantics of π originates in Berry and Boudol's *Chemical Abstract Machine*. It is based on the definition of two relations between processes:

- *Structural congruence* is the least congruence relation satisfying the rules of Fig. 2. The important rule in the third line describes *name extrusion*.
- *Reduction* is the least relation satisfying the rules of Fig. 3.

It can be noted that the last three rules for replication in Fig. 2 are not included in all presentations of structural congruence for the π-calculus (in particular not in Milner's original presentation).

1.1.3 Labelled Transition System

We present the labelled transition system for the π-calculus *early semantics* in Fig. 4. A labelled transition system consists of a set of *processes* $\mathcal{P}$, a set of *actions* $\mathcal{A}$, and a *transition relation* included in $\mathcal{P} \times \mathcal{A} \times \mathcal{P}$. Actions label the evolutions that a process can have, either through an interaction with its context or via some internal computation.

Actions.

$$\mu \quad ::= \quad a(n) \mid \overline{a}\langle n\rangle \mid \overline{a}(n)_{\boldsymbol{\nu}} \mid \tau\,.$$

There are four kinds of actions: input, free output, bound output[1] and a special action τ to denote internal computation.

The bound and free names of an action μ (written respectively $\mathrm{bn}(\mu)$ and $\mathrm{fn}(\mu)$) are defined by saying that name n is bound in $\overline{a}(n)_{\boldsymbol{\nu}}$, other occurrences of names in actions defining free names. We also set $\mathrm{n}(\mu) \stackrel{\text{def}}{=} \mathrm{bn}(\mu) \cup \mathrm{fn}(\mu)$.

$$\text{Inp}\ \ a(m).P \xrightarrow{a(n)} P_{\{m \leftarrow n\}} \qquad \text{Out}\ \ \overline{a}\langle n\rangle.P \xrightarrow{\overline{a}\langle n\rangle} P$$

$$\text{Comm}_{\text{l}}\ \frac{P \xrightarrow{a(n)} P' \quad Q \xrightarrow{\overline{a}\langle n\rangle} Q'}{P \,|\, Q \xrightarrow{\tau} P' \,|\, Q'}$$

$$\text{Par}_{\text{l}}\ \frac{P \xrightarrow{\mu} P'}{P \,|\, Q \xrightarrow{\mu} P' \,|\, Q}\ \mathrm{bn}(\mu) \cap \mathrm{fn}(Q) = \emptyset$$

$$\text{Res}\ \frac{P \xrightarrow{\mu} P'}{(\boldsymbol{\nu} n)\, P \xrightarrow{\mu} (\boldsymbol{\nu} n)\, P'}\ n \notin \mathrm{n}(\mu)$$

$$\text{Bang}\ \frac{!P \,|\, P \xrightarrow{\mu} P'}{!P \xrightarrow{\mu} P'}$$

$$\text{Open}\ \frac{P \xrightarrow{\overline{a}\langle n\rangle} P'}{(\boldsymbol{\nu} n)\, P \xrightarrow{\overline{a}(n)_{\boldsymbol{\nu}}} P'}\ n \neq a$$

$$\text{Close}_{\text{l}}\ \frac{P \xrightarrow{a(n)} P' \quad Q \xrightarrow{\overline{a}(n)_{\boldsymbol{\nu}}} Q'}{P \,|\, Q \xrightarrow{\tau} (\boldsymbol{\nu} n)\, (P' \,|\, Q')}\ n \notin \mathrm{fn}(P)$$

Figure 4: Early labelled transition system

Nota: symmetrical versions of rules Comm_{l}, Par_{l} *and* Close_{l} *have been omitted.*

---

[1] The $\boldsymbol{\nu}$ in subscript in a bound output action is rarely found in the literature. We use it here to stress the (slight) typographical difference between $\overline{a}\langle n\rangle$ (free output of n at a) and $\overline{a}(n)$ (bound output).

Proposition. $\longrightarrow \;=\; \xrightarrow{\tau} \equiv$.

This presentation corresponds to an *early* transition semantics because the transmitted name is substituted as soon as possible, namely in rule Inp. We refer to the literature for other labelled transition systems for π, in particular the *late semantics,* where substitution is applied at the point where a τ action (a synchronisation) is derived. This difference has consequences on the behavioural properties of processes (see e.g. [SW01]).

1.2 Related calculi

CCS.

The *Calculus of Communicating Systems* is an important predecessor of the π-calculus. It can be seen as a restriction of π in which only a dummy name w is exchanged in communications, and w cannot be used in subject position. Here is an example:

$$\overline{a}\langle w\rangle.b(w).(\overline{c}\langle w\rangle \,|\, \overline{c'}\langle w\rangle) \mid a(w).\overline{b}\langle w\rangle$$

would be written in CCS:

$$\overline{a}.b.(\overline{c} \,|\, \overline{c'}) \mid a.\overline{b}$$

In CCS, interaction is made of *pure synchronisation* on channels, with no transmission of values (in the context of the polyadic π-calculus – see below – we could say that CCS processes only exchange empty tuples of names). This makes CCS less expressive than π; in particular, *name extrusion* is a distinguishing feature of π w.r.t. CCS. As a consequence, only immutable connection patterns can be modeled in CCS.

Reference: R. Milner, *Communication and Concurrency*, Prentice Hall.

Polyadicity.

In the polyadic π-calculus, communication involves the transmission of *tuples of names*. Tuples are possibly empty, which corresponds to CCS-like synchronisation.

$$P \quad ::= \quad \mathbf{0} \mid P_1 \,|\, P_2 \mid !P \mid (\boldsymbol{\nu} n)\, P \mid \overline{n}\langle \tilde{m}\rangle.P \mid n(\tilde{m}).P$$

In an input $n(\tilde{m}).P$, with $m = (m_1, \ldots, m_k)$, the m_is have to be pairwise distinct names. The definition of the operational semantics of the polyadic π-calculus involves some technicalities. Output actions have the general form $(\boldsymbol{\nu}\tilde{m})\,\overline{a}\langle \tilde{n}\rangle$ with $\tilde{m} \subseteq \tilde{n}$, the tuple of extruded names being possibly empty. It has to be stressed that an overloaded notation is commonly used here: $\tilde{m}$ is treated as having a *set* structure (instead of a tuple structure), which is also necessary when writing the inclusion between *sets* $\tilde{m} \subseteq \tilde{n}$ above.

Because of possible arity conflicts between emission and reception on a given channel, the polyadic π-calculus is intrinsically typed.

Reference: [SW01]

Other operators. Some versions of the π-calculus include additional operators, among which we can mention:

choice $(+)$ A term of the form $P+Q$ may evolve either as P or as Q, depending on the interactions offered by the context. This operator is 'inherited' from CCS, and is most commonly used with prefixed π-calculus processes (see in particular works by Nestmann, Pierce – see also the paragraph about asynchronous π below).

matching The operator of matching allows a process to compare names: a term of the form $[n = m]P$ evolves as P if n and m denote the same name. *Mismatching* as well as an 'if-then-else' version of matching can also be found in the literature.

recursive definitions Infinite behaviours can be introduced in the π-calculus by using recursive definitions instead of the replication operator. These have the form

$$A[\tilde{x}] \stackrel{\text{def}}{=} P\,,$$

where $\tilde{x}$ is a tuple of names, and P may contain occurrences of A (instanciated with the corresponding parameters). Recursive definitions are easier to manipulate when using π as a specification language; on the other hand, they bring more technicalities in the calculus. At the level of expressiveness, replication and recursive definitions are equivalent.

higher-order communications The *Higher-Order π-calculus* ($HO\pi$) allows channels to carry process abstractions (which can be seen as non-located input processes). It has been shown that $HO\pi$ can be faitfully encoded in the π-calculus, thus suggesting that name mobility is enough (at least from a theoretical point of view) to represent code mobility.

Asynchronous π.

The asynchronous π-calculus is an important subcalculus of π, both from practical and theoretical points of view. This calculus is obtained by forbidding continuations after outputs, that thus become asynchronous: an emission is of the form $\overline{a}\langle b\rangle$, and is interpreted as dropping a message 'in the soup'.

$$P \quad ::= \quad P_1 \,|\, P_2 \mid\, !P \mid\, (\boldsymbol{\nu} n)\, P \mid\, \overline{n}\langle m\rangle \mid\, n(m).P$$

We may remark the absence of the terminal $\mathbf{0}$ process, which is encodable (up to any reasonable behavioural equivalence) as $\boldsymbol{\nu} n.\overline{n}\langle n\rangle$.

Adopting asynchronous output can be justified from an implementation point of view (indeed, programming language implementations based on π such as PICT or JoCaml are based on an asynchronous language). Moreover, asynchronous contexts make several distinctions between various notions of bisimulation collapse, and give rise to simple and useful proof techniques.

The problem of implementing the operator of choice (mentioned above) has led to a series of work about the separation between full π and its asynchronous version (see works by Palamidessi).

Reference: [SW01]

Localised π.

The *localised π-calculus* (which is closely related to the Join calculus) is defined by imposing a commonly used discipline in the usage of names[2]: in a term of the form

$$a(n).P\,,$$

the name n can only be used in output in the continuation P, that is P can either make an emission on channel n or transmit n on another channel, but cannot make an input on n. In other words, n should not occur free in input position in P. The 'localised discipline' can be ensured using a type system, but

[2]In particular, do not get misled by this terminology: *localised* π does not introduce an explicit notion of locality to be manipulated at the level of syntax, but rather enforces some kind of *local usage* of names.

is such a common idiom that it is worth studying a calculus where this discipline is wired in the syntax.

In localised π, every subterm listening on some channel is local to the process that created this channel. This kind of information can be relevant when writing a distributed implementation of π, in order to find where messages sent on a given channel should be routed to.

An interesting analogy is with object-oriented systems: intuitively, the discipline of localised π is suited for the representation of objects, in the sense that communicating the name of an object lets an agent interact (by invoking some methods) with it, but precludes the creation of a different object having the same name (in this approach, the interface of an object would be represented by a process listening on a corresponding channel).
Reference: [SW01]

The Join calculus.

The Join calculus serves as the basis of a programming language for mobile and distributed systems called *JoCaml*. It uses a 'local π' discipline for accessing channels (only output capability is transmitted), and is based on the notion of *join definition*, which concentrates in one construct π's operators for input, restriction and replication. Communications are asynchronous in Join.

A typical join definition looks like

$$a(x) \mid b(y) \mid c(z) \ \triangleright \ P$$

and entails the introduction of:

- names a, b and c;
- the (persistent) definition associating the pattern $a(x) \mid b(y) \mid c(z)$ with process P (in which x, y and z can only be used in output).

Join allows for the simultaneous introduction of several definitions, that use the same input names according to different synchronisation patterns. This programming idiom turns out to be very convenient to specify concurrent behaviours, and is somehow reminiscent of Petri Nets.

The join calculus has also been enriched with a notion of *locality* and of *movement* of localities, to support distributed programming.
Reference: `http://join.inria.fr`

Distributed π.

Although the π-calculus has enough expressiveness to describe mobile computation, it does not have an explicit notion of locality, which can be useful when reasoning about distributed forms of computation. The *distributed π-calculus* ($D\pi$) is an extension of the π-calculus in which processes are located at *sites*. More precisely, the 'toplevel' consists in a parallel composition of sites where threads (given as π-calculus processes) are running. The language is also enriched with a `goto` primitive to let code migrate between sites. Type systems to address some properties of mobile and distributed computation have been introduced for $D\pi$.
Reference: Work by Hennessy and Yoshida. See also Nomadic π (by Sewell et al.), that is based on a richer notion of distribution.

Mobile Ambients.

The calculi of Mobile Ambients really form a tribe, starting from Cardelli and Gordon's original calculus. They are *nominal calculi* (in the sense, roughly, that terms are built out of names), like π, but the primitive notion of interaction is *movement* of locations (as opposed to name passing). In Ambients, computation consists in the reconfiguration of a spatial structure, which is given by a tree of locations. A location is an Ambient: $n[\,P\,]$ represents the process P running at location n (or in Ambient n).

The core of Cardelli and Gordon's calculus is *Pure Mobile Ambients:*

$$P \quad ::= \quad \mathbf{0} \mid P_1 \mid P_2 \mid\, !P \mid n[\,P\,] \mid \mathsf{in}\, n.P \mid \mathsf{out}\, n.P \mid \mathsf{open}\, n.P$$

The first four constructs are used to describe a spatial configuration, while the in, out and open *capabilities* are used to make a configuration evolve, accoding to the following rules:

$$\begin{array}{rcl} n[\,\mathsf{in}\, m.P \mid Q\,] \mid m[\,R\,] & \longrightarrow & m[\,n[\,P \mid Q\,] \mid R\,] \\ m[\,n[\,\mathsf{out}\, m.P \mid Q\,] \mid R\,] & \longrightarrow & n[\,P \mid Q\,] \mid m[\,R\,] \\ \mathsf{open}\, n.P \mid n[\,Q\,] & \longrightarrow & P \mid Q \end{array}$$

Movements are *subjective* in Ambients: in the rules for in and out, a process *inside* an ambient takes the control over it to let it move. The presence of the open capability is quite controversial, but pure movement capabilities do not seem to be enough to 'program' in Ambients. Pure Mobile Ambients can be enriched with a form of communication, which is local to an ambient.

Variations. As it is, the calculus of Mobile Ambients is 'very permissive': it has powerful primitives, that can be used in a very distributed and asynchronous fashion. Variants of the calculus have attempted to provide a better control on Ambients, in order to make their behaviour more predictable, and programmable. These include most notably Safe Ambients (where *cocapabilities* impose a form of synchronisation to trigger movements) and Boxed Ambients (where the open capability is dropped in favour of more complex communication patterns).

Reference: For Mobile Ambients: Cardelli and Gordon's work – Safe Ambients: Levi and Sangiorgi – Boxed Ambients: Bugliesi, Castagna, Crafa et al.

Internal mobility and fusions.

The π-calculus has two binders: reception (input), and restriction. While an input-bound name is instanciated upon communication with the transmitted value, a restricted name is never substituted. Combining output with restriction, we get a construction which is quite similar to input. Consider the synchronisation:

$$a(q).P \mid (\boldsymbol{\nu} p)\, \overline{a}\langle p \rangle.Q \quad \longrightarrow \quad (\boldsymbol{\nu} p)\, (P_{\{q \leftarrow p\}} \mid Q)\,.$$

In this transition, the participants can agree, using α-conversion, on a name n *before* synchronisation, which illustrates the symmetry. We thus write:

$$a(n).\underbrace{P_{\{q \leftarrow n\}}}_{P_1} \mid (\boldsymbol{\nu} n)\, \overline{a}\langle n \rangle.\underbrace{Q_{\{p \leftarrow n\}}}_{Q_1} \quad \longrightarrow \quad (\boldsymbol{\nu} n)\, (P_1 \mid Q_1)\,.$$

This idea is related to the definition of the *fusion calculus,* where fusions denote the identifications between names that result from interactions. Fusions have notably been used to encode constraints in the π-calculus, and also to study implementation patterns for π-calculus-based languages.

Reference: Works by Parrow, Victor, Gardner, Wischik – see
`http://www.wischik.com/lu/research/fusions.html`

See also πI (the π-calculus where only emission of fresh names is allowed, which corresponds to *internal mobility*), the Chi calculus (Fu), Solos (Laneve, Victor).

Spi.

The Spi calculus has been introduced to specify and analyze cryptographic protocols, by writing protocols as processes. It is an enrichment of the π-calculus with some constructs to represent cryptographic primitives such as various forms of encryption and decryption. The main difference between π and Spi is in the shape of messages that are communicated: a value like $\langle \tilde{n} \rangle_k$ represents the encryption of tuple $\tilde{n}$ with name k (interpreted as *key* k). Creation of nounces (which are ubiquitous in cryptographic protocols) is represented in Spi using the restriction operator.

The theory of the π-calculus has been adapted to Spi, in order to express and prove security properties using bisimulation-based equivalences or type systems.
Reference: Papers by Abadi and Gordon.

Another related reference is the study of cryptographc protocols based on multiset rewriting (see works by Mitchell, Cervesato).

Applied π.

The mechanism of *name passing* turns out to bring considerable expressive power. However, working in the pure π-calculus can quickly become tedious when manipulating even simple datastructures or writing elementary (sequential) operations – like writing programs in λ with Church's integers. The *applied π-calculus* offers a framework to study enriched versions of π, where programming languages primitives can be 'plugged' into the calculus. Complex proofs about cryptographic systems have been conducted within applied π.
Reference: Work by Abadi and Fournet.

2 Typed π-calculus

We present here type systems for the polyadic π-calculus. The difference with types for monadic π is minor. Moreover, historically, types for the π-calculus have been introduced in a polyadic context (the polyadic π-calculus is hardly manipulated in an untyped context).

Types are a distinguishing feature of the π-calculus w.r.t. CCS. They are used to check statically some properties of processes, and also to help in establishing behavioural properties of processes (the latter aspect being out of the scope of these notes).

2.1 Simple types

Simple types are ranged over using T, U and are defined by the following grammar:

$$T \quad ::= \quad \#\tilde{T}\,,$$

where $\tilde{T}$ is a notation for (possibly empty) tuples of types. Note that in this presentation, the only base type is given by taking an empty tuple in $\tilde{T}$ (and corresponds to the type of CCS-like channels). Basic datatypes like booleans or integers can smoothly be integrated in this system, as well as programming

languages constructions like variant types and records (see [SW01]). The syntax of terms is modified by associating explicit type annotations with the creation of new names: we shall write $(\boldsymbol{\nu} n : T)\, P$. Note that input-bound names (or name tuples) do not require such annotations, since the type of the expected parameters can be deduced from the type of the subject of the input. Technically, working in a typed language also requires the redefinition of the operational semantics, which amounts to make some simple changes, that we shall not describe here. We adopt in what follows a reduction-based presentation of the operational semantics.

Typing contexts, ranged over with Γ, are lists of hypotheses of the form $n : T$, where n is a name and T is a type. For n a name and T a type, $\Gamma, n : T$ stands for the context obtained by adding $n : T$ to Γ, hiding a possible previous hypothesis on n in Γ. We write $\Gamma(n)$ for the type associated to name n in Γ. If $\tilde{n} = (n_1, \ldots, n_k)$, and $\tilde{T} = (T_1, \ldots, T_k)$, we write $\Gamma, \tilde{n} : \tilde{T}$ for $\Gamma, n_1 : T_1, \ldots, n_k : T_k$.

The typing rules for simple types are given in Fig. 5. They define two type judgments, one of the form $\Gamma \vdash \tilde{n} : \tilde{T}$, to assign type tuples to tuples of names, and one of the form $\Gamma \vdash P$, to assert that a process is well-typed.

$$\frac{\Gamma(n) = T}{\Gamma \vdash n : T} \qquad \frac{\Gamma \vdash n_i : T_i}{\Gamma \vdash (n_1, \ldots, n_k) : \langle T_1, \ldots, T_k \rangle}$$

$$\frac{\Gamma \vdash P \quad \Gamma \vdash n : \#\tilde{T} \quad \Gamma \vdash \tilde{m} : \tilde{T}}{\Gamma \vdash \overline{n}\langle \tilde{m} \rangle.P} \qquad \frac{\Gamma, \tilde{x} : \tilde{T} \vdash P \quad \Gamma \vdash n : \#\tilde{T}}{\Gamma \vdash n(\tilde{x}).P}$$

$$\frac{\Gamma, x : T \vdash P}{\Gamma \vdash (\boldsymbol{\nu} x : T)\, P} \qquad \frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \,|\, Q} \qquad \frac{\Gamma \vdash P}{\Gamma \vdash\, !P} \qquad \Gamma \vdash \mathbf{0}$$

Figure 5: Simple types for the polyadic π-calculus

Properties of the type system.

Lemma [Strengthening and Weakening]. *For any P, n, T and Γ,*

- *if $\Gamma, n : T \vdash P$ and n does not occur free in P, then $\Gamma \vdash P$;*
- *if $\Gamma \vdash P$, then $\Gamma, n : T \vdash P$ if Γ does not contain a typing hypothesis on n.*

Lemma [Substitution]. *For any P, n, m, T and Γ, if $\Gamma \vdash P$, $\Gamma \vdash n : S$, and $\Gamma \vdash m : S$, then $\Gamma \vdash P_{\{n \leftarrow m\}}$.*

We now say that simple types are aimed at controlling the arity of channels.

Theorem [Correctness of arity checking]. *For any P and Γ, if $\Gamma \vdash P$ and $P \equiv (\boldsymbol{\nu} \tilde{n}) \left(\overline{a}\langle m_1, \ldots, m_k \rangle.P_1 \mid a(x_1, \ldots, x_l).P_2 \mid P_3\right)$, then $k = l$.*

Correctness of the type system follows from the following key result:

Theorem [Subject reduction]. *For any P, P' and Γ, $\Gamma \vdash P$ and $P \longrightarrow P'$ implies $\Gamma \vdash P'$.*

2.2 Types and capabilities: I/O types and subtyping

In the system of Input-Output types (I/O types), a channel type can be decomposed into *capabilities,* for input and output. The grammar for types becomes:

$$T \quad ::= \quad \mathsf{i}\tilde{T} \mid \mathsf{o}\tilde{T} \mid \#\tilde{T}\,.$$

Capabilities naturally give rise to a notion of subtyping. The subtyping relation, written $T \leq U$, is defined on Fig. 6, that also gives the typing rules that are different from those of Fig. 5, as well as the additional *subsumption rule*.

$$\mathrm{T} \leq \mathrm{T} \qquad \frac{T \leq T' \quad T' \leq T''}{T \leq T''} \qquad \frac{T_1 \leq U_1 \quad \cdots \quad T_k \leq U_k}{(T_1, \ldots, T_k) \leq (U_1, \ldots, U_k)}$$

$$\#T \leq \mathsf{o}\mathsf{T} \qquad \frac{T \leq U}{\mathsf{i}T \leq \mathsf{i}U} \qquad \frac{T \leq U}{\mathsf{o}U \leq \mathsf{o}T} \qquad \frac{T \leq U \quad U \leq T}{\#T \leq \#U}$$

$$\#T \leq \mathsf{i}\mathsf{T}$$

$$\frac{\Gamma \vdash n : T \quad T \leq U}{\Gamma \vdash n : U}$$

$$\frac{\Gamma \vdash P \quad \Gamma \vdash n : \mathsf{o}\tilde{T} \quad \Gamma \vdash \tilde{m} : \tilde{T}}{\Gamma \vdash \overline{n}\langle\tilde{m}\rangle.P} \qquad \frac{\Gamma, \tilde{x} : \tilde{T} \vdash P \quad \Gamma \vdash n : \mathsf{i}\tilde{T}}{\Gamma \vdash n(\tilde{x}).P}$$

Figure 6: Input-Output types: subtyping and modified typing rules

The π-calculus with Input-Output types enjoys basically the same properties as the simply typed system.

Other type systems. Several other kinds of type systems have been proposed to analyse π-calculus terms. *Linear types* can guarantee that some channels are used at most once (in input and in output); this kind of information validates some optimisations in implementations of π (like the PICT programming language).

Similarly, *receptive* channels are also frequently used in π-calculus programming, and can be detected using *receptive types*. They intuitively correspond to the location of resources, that are always available at toplevel.

Polymorphism is also present in the π-calculus, like e.g. in the classical following process:

$$a(v, r).\overline{v}\langle r\rangle\,.$$

The polymorphic π-calculus is also at work in PICT, together with a type inference procedure.

Other type systems, involving *session types* or various forms of *graph types*, have not been integrated in an implementation of π. Work has also been done on types to insure termination of subclasses of processes. Some proposals also use CCS terms to type π-calculus processes.

3 Reasoning about the behaviour of processes

We sketch here (a small part of) the theory of contextual equivalence in the π-calculus. Some of the notions presented below correspond to a direct adaptation of the corresponding concepts in CCS, the results and proof techniques being sometimes more complex due to the greater expressive power of π. Useful references for this Section include Milner's book on CCS (*Communication and Concurrency*, Prentice Hall) and [SW01].

3.1 Reduction-based contextual equivalence

We first introduce a behavioural equivalence based on observability predicates that we call *barbs*.

Definition [Barb]. *Given a name n and a process P, P exhibits barb n (resp. $\overline{n}$), written $P \downarrow_n$ (resp. $P \downarrow_{\overline{n}}$) iff $P \equiv (\boldsymbol{\nu}\tilde{m})\,(n(x).P_1 \mid P_2)$ (resp. $P \equiv (\boldsymbol{\nu}\tilde{m})\,(\overline{n}\langle v\rangle.P_1 \mid P_2)$) with $n \notin \tilde{m}$.*

Definition [Barbed bisimilarity, barbed equivalence]. *A* symmetric *relation $\mathcal{R}$ between processes is a* strong barbed bisimulation *iff for any P, Q, whenever $P\,\mathcal{R}\,Q$:*

- *$P \downarrow_\eta$ implies $Q \downarrow_\eta$ (η being of the form n or $\overline{n}$);*
- *if $P \longrightarrow P'$ then there exists Q' such that $Q \longrightarrow Q'$ and $P'\,\mathcal{R}\,Q'$.*

Strong barbed bisimilarity, *written $\dot{\sim}$, is the greatest strong barbed bisimulation.*

Two processes P and Q are strong barbed equivalent, *written $P \simeq Q$, iff for any R, $P \,|\, R \dot{\sim} Q \,|\, R$.*

It has to be noted that the pattern we have followed to introduce barbed bisimilarity pertains to a very general approach: it is only based on a notion of reduction (internal computation) and a notion of observable (here, barbs). Barbed equivalence has then been obtained from barbed bisimilarity by closing w.r.t. a simple form of contexts. *Barbed congruence* can also be defined, by taking all contexts into account (see also a discussion on π-calculus contexts below).

3.2 Bisimulation and bisimilarity

Definition [(Strong) Bisimulation, bisimilarity]. *Given a labelled transition system written $\xrightarrow{\mu}$, a* symmetric *relation $\mathcal{R}$ between processes is a* bisimulation *iff for any P, Q, P' and μ, whenever $P\,\mathcal{R}\,Q$ and $P \xrightarrow{\mu} P'$, there exists Q' such that $Q \xrightarrow{\mu} Q'$ and $P'\,\mathcal{R}\,Q'$.*

Bisimilarity is the greatest bisimulation.

In the π-calculus, the labelled transition system of Fig. 4 gives rise to the notion of *strong early bisimilarity* (in short *strong bisimilarity*, or even simply bisimilarity) written $\sim$.

A *context* (ranged over with C) is a π-calculus term containing an occurrence of a special term, $[\,]$, called the *hole*. If C is a context and P is a process, $C[P]$ denotes the process obtained by replacing the hole in C with P.

A relation $\mathcal{R}$ is a congruence if it is preserved by all contexts, i.e. if for any P, Q, $P\,\mathcal{R}\,Q$ implies that for any C, $C[P]\,\mathcal{R}\,C[Q]$. In the π-calculus, a *non-input congruence* is a relation that is preserved by all contexts in which the hole does not occur under an input construct.

Theorem. *In the π-calculus, strong bisimilarity is a non-input congruence.*

$\sim$ is not a 'full' congruence in the π-calculus. The classical counterexample is written in the π-calculus enriched with the operator of choice: take $P \stackrel{\text{def}}{=} a(x).\overline{b}\langle v\rangle + \overline{b}\langle v\rangle.a(x)$ and $Q \stackrel{\text{def}}{=} a(x) \mid \overline{b}\langle v\rangle$, and consider the context $C \stackrel{\text{def}}{=} c(a).[\,]$.

Theorem [Characterisation of $\simeq$]. *For any P and Q, $P \sim Q$ iff $P \simeq Q$.*

This result shows that $\sim$ can be seen as a proof technique for $\simeq$. Historically, labelled transition systems for the π-calculus have been proposed before the notion of barbed equivalence was invented.

Weak case. The notion of weak (early) bisimulation is obtained by adapting these definitions, taking the point of view that internal computations (τ steps) are unobservable.

Definition [Weak transitions, weak bisimulation]. *Take $\stackrel{\mu}{\longrightarrow}$ to be the labelled transition system defined in Fig. 4. We let $\Longrightarrow$ be the reflexive transitive closure of $\stackrel{\tau}{\longrightarrow}$, and define $\stackrel{\hat{\mu}}{\Longrightarrow} \stackrel{\text{def}}{=} \Longrightarrow\stackrel{\mu}{\longrightarrow}\Longrightarrow$ if $\mu \neq \tau$, and $\stackrel{\hat{\tau}}{\Longrightarrow} \stackrel{\text{def}}{=} \Longrightarrow$.*

A symmetric relation $\mathcal{R}$ between processes is a weak bisimulation *iff for any P, Q, P' and μ, whenever $P\,\mathcal{R}\,Q$ and $P \stackrel{\mu}{\longrightarrow} P'$, there exists Q' such that $Q \stackrel{\hat{\mu}}{\Longrightarrow} Q'$ and $P'\,\mathcal{R}\,Q'$.*

Weak bisimilarity, written $\approx$, is the greatest weak bisimulation. Quite a few other different notions of equivalences and preorders have been introduced to reason about π-calculus processes (late and open bisimulation, expansion, coupled bisimulation, ...). There also exist weak versions of the barbed equivalences described above.

3.3 Proof techniques

Bisimulation is the most commonly used technique to establish bisimilarity laws. This method can be further enhanced through the use of *up-to proof techniques,* that can be introduced with the following definition:

Definition [Up-to bisimulation]. *We consider a function $\mathcal{F}$ from relations (between processes) to relations and a labelled transition system $\stackrel{\mu}{\longrightarrow}$. A symmetric relation $\mathcal{R}$ is a* bisimulation up to $\mathcal{F}$ *iff for any P, Q, P' and μ, whenever $P\,\mathcal{R}\,Q$ and $P \stackrel{\mu}{\longrightarrow} P'$, there exists Q' such that $Q \stackrel{\mu}{\longrightarrow} Q'$ and $P'\,\mathcal{F}(\mathcal{R})\,Q'$.*

Of course, not any function $\mathcal{F}$ can be used. A function $\mathcal{F}$ on relations induces a sound proof technique if ($\mathcal{R}$ is a bisimulation up to $\mathcal{F}$) imples ($\mathcal{R} \subseteq \sim$).

Examples of correct up-to proof techniques for $\sim$ in the π-calculus are:

- up to bisimilarity: $\mathcal{F}(\mathcal{R}) \stackrel{\text{def}}{=} \sim \mathcal{R} \sim$ (we use here relation composition);
- up to parallel composition: $\mathcal{F}(\mathcal{R}) \stackrel{\text{def}}{=} \{(P \mid R, Q \mid R).\, P\,\mathcal{R}\,Q\}$;
- up to restriction: $\mathcal{F}(\mathcal{R}) \stackrel{\text{def}}{=} \{\big((\boldsymbol{\nu} n)\,P, (\boldsymbol{\nu} n)\,Q\big).\, P\,\mathcal{R}\,Q\}$;

- up to injective substitution:

$$\mathcal{F}(\mathcal{R}) \stackrel{\text{def}}{=} \{(P\sigma, Q\sigma).\, P\,\mathcal{R}\,Q \text{ and } \sigma \text{ an injective substitution}\}\,.$$

In the weak case, it should be stressed that weak bisimulation up to strong bisimilarity is sound, but weak bisimulation up to weak bisimilarity is not.

4 Encoding λ in π

We define encodings of the call-by-name and call-by-value λ-calculus in the polyadic synchronous π-calculus. We only give here the definitions of the encodings as an illustration of how functions can be programmed in the π-calculus. We do not enter the details of the correspondence results that validate these encodings (see [SW01]).

A λ-term M is translated into a process $[\![M]\!]_p$, where p is a name, that is the encoding is parametric on some port that represents the channel where the 'value' corresponding to M will be made available.

---

abstraction

$$[\![\lambda x.M]\!]_p \stackrel{\text{def}}{=} (\boldsymbol{\nu} v)\,\overline{p}\langle v\rangle.!v(x,q).[\![M]\!]_q$$

variable

$$[\![x]\!]_p \stackrel{\text{def}}{=} \overline{x}\langle p\rangle$$

application: call-by-name

$$[\![M\,N]\!]_p \stackrel{\text{def}}{=} (\boldsymbol{\nu} q)\,\big([\![M]\!]_q \mid q(v).(\boldsymbol{\nu} x)\,\overline{v}\langle x,p\rangle.!x(r).[\![N]\!]_r\big)$$

application: call-by-value

$$[\![M\,N]\!]_p \stackrel{\text{def}}{=} (\boldsymbol{\nu} q)\,\big([\![M]\!]_q \mid q(v).(\boldsymbol{\nu} r)\,([\![N]\!]_r \mid r(w).(\boldsymbol{\nu} x)\,\overline{v}\langle x,p\rangle.!x(r').\overline{r'}\langle w\rangle)\big)$$

Figure 7: π-calculus encodings of the λ-calculus

The definition of the encodings is given on Fig. 7. The names introduced in every clause (restricted or bound names) are supposed to be fresh. The encodings use two kinds of names, that are ranged over using different letters: p, q, r, r' denote a *location* where the result of an evaluation is made available, while x, y are names that are used to access values.

In the π-calculus, terms cannot be passed in a communication. Therefore, the translations of λ in π make use of an indirection, and an evaluation returns *a pointer* to its result. This discipline suggests an analogy with continuation passing style (CPS) transformations in the λ-calculus. [SW01] gives a systematic description of π-calculus encodings of λ via a compilation in three steps: first a CPS translation, then a translation from the CPS version of a term into the Higher-Order π-calculus, and finally a compilation from $HO\pi$ into π.

Theorem [Adequacy]. *Given a λ-term M, M converges to a value (for a strategy, be it call-by-name or call-by-value) iff for any p, $[\![M]\!]_p \longrightarrow^* \downarrow_\eta$ for some η.*

The correspondence between λ-terms and their encodings can be made more precise than the statement above, in the sense that the encodings validate the λ-theory. These results are based on notions of *typed behavioural equivalence,* which we will not discuss here.

Encodings of the typed λ-calculus into the I/O π-calculus have also been studied. The translation of arrow types is rendered quite naturally in terms of I/O types. As mentioned above, the latter are also needed to prove certain behavioural properties of processes resulting from the translation of a function, to enforce the programming discipline associated to the encoding.

Call-by-name is the strategy that is most naturally rendered in the π-calculus. An accurate study of the equality on λ-terms induced by the π-calculus encoding of call-by-name is made in [SW01]. It is shown that this relation coincides with the equality of the corresponding Lévy-Longo trees. Moreover, this relation does not correspond exactly to *applicative bisimilarity*, a reason for this being that the π-calculus is not confluent. Full abstraction can be obtained by enriching the λ-calculus with certain non-confluent operators.

5 Sources

The references mentioned along the text are rather rough. However, most of the works should be easy to find on the WWW. Here is a list of valuable starting points to look for information on the subjects mentioned above.

Books. There exist two books on the π-calculus, both published by Cambridge University Press. The one by Milner *(Communicating and Mobile Systems: the π-calculus)* is aimed at a rather broad audience, and gives an introduction to the subject. Sangiorgi and Walker's book *(The π-calculus: a theory of mobile processes)* is a comprehensive exposition of the theory of π, sometimes going into rather technical considerations. Milner's book on CCS (*Communication and Concurrency*, Prentice Hall) is also a useful reference, especially on bisimulation.

Survey papers.

J. Parrow's chapter on the π-calculus in the *Handbook of Process Algebra* (Elsevier, 2001) gives a good overview on the theory of π. An alternative is P. Sewell's *Applied π – A brief tutorial*. S. Gay's survey on type systems (*Some type systems for the π-calculus*) is also a good starting point on this particular topic (and has been helpful in preparing these notes).

Web pages and related sources.

Some web pages collect links to researchers and projects in the field. The most relevant are:

- mobility `http://lamp.epfl.ch/mobility`
- ambients `http://xdguan.freezope.org/wiki/AmbientCalculiOnline`

There are also mailing lists:

- *moca* (mobile calculi)
  `http://www-sop.inria.fr/mimosa/personnel/Davide.Sangiorgi/moca.html`
- *concurrency* `http://www.cwi.nl/~bertl/concurrency/concurrency.html`

Reference

[SW01] D. Sangiorgi and D. Walker, *The π-calculus, a theory of mobile processes*, Cambridge University Press, 2001.