An Introduction to the π-Calculus

Chapter to appear in *Handbook of Process Algebra*, ed. Bergstra, Ponse and Smolka, Elsevier

Joachim Parrow*
Dep. Teleinformatics,
Royal Institute of Technology, Stockholm

Abstract

The π-calculus is a process algebra where processes interact by sending communication links to each other. This paper is an overview of and introduction to its basic theory. We explore the syntax, semantics, equivalences and axiomatisations of the most common variants.

*email joachim@it.kth.se

Contents

1 Introduction **3**

2 The π-Calculus **6**
2.1 Basic Definitions 6
2.2 Structural Congruence 9
2.3 Simple Examples 12

3 Variants of the Calculus **15**
3.1 Match and Mismatch 15
3.2 Sum 16
3.3 The Polyadic Calculus 18
3.4 Recursion and Replication 20
3.5 The Asynchronous Calculus 21
3.6 The Higher-Order Calculus 23

4 Operational Semantics **25**

5 Variants of the Semantics **28**
5.1 The Role of Structural Congruence 28
5.2 Symbolic Transitions 30
5.3 The Early Semantics 34
5.4 Reductions 35
5.5 Abstractions and Concretions 36

6 Bisimilarity and Congruence **39**
6.1 Bisimilarity 39
6.2 Congruence 42

7 Variants of Bisimilarity **43**
7.1 Early Bisimulation 43
7.2 Barbed Congruence 46
7.3 Open Bisimulation 47
7.4 Weak Bisimulation 52

8 Algebraic Theory **54**
8.1 Bisimilarity 54
8.2 Congruence 57

9 Variants of the Theory **60**
9.1 Early Bisimilarity and Congruence 60
9.2 Open Bisimilarity 62
9.3 Weak Congruence 62

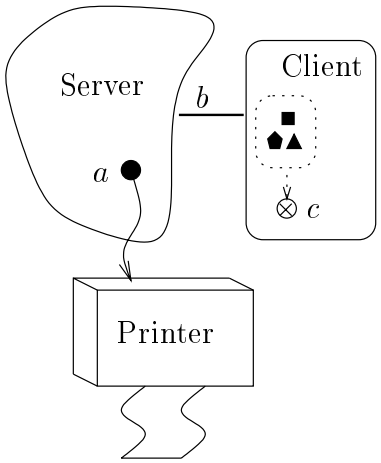
10 Sources **64**

1 Introduction

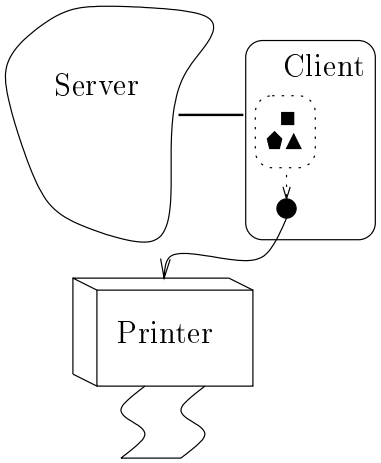
The π-calculus is a mathematical model of processes whose interconnections change as they interact. The basic computational step is the transfer of a communication link between two processes; the recipient can then use the link for further interaction with other parties. This makes the calculus suitable for modelling systems where the accessible resources vary over time. It also provides a significant expressive power since the notions of access and resource underlie much of the theory of concurrent computation, in the same way as the more abstract and mathematically tractable concept of a function underlies functional computation. This introduction to the π-calculus is intended for a theoretically inclined reader who knows a little about the general principles of process algebra and who wishes to learn the fundamentals of the calculus and its most common and stable variants.

Let us first consider an example. Suppose a server controls access to a printer and that a client wishes to use it. In the original state only the server itself has access to the printer, represented by a communication link a. After an interaction with the client along some other link b this access to the printer has been transferred:

Before interaction:



After interaction:



In the π-calculus this is expressed as follows: the server that sends a along b is $\overline{b}a\,.\,S$; the client that receives some link along b and then uses it to send data along it is $b(c)\,.\,\overline{c}d\,.\,P$. The interaction depicted above is formulated

$$\overline{b}a\,.\,S \mid b(c)\,.\,\overline{c}d\,.\,P \quad \xrightarrow{\tau} \quad S \mid \overline{a}d\,.\,P$$

We see here that a plays two different roles. In the interaction between the server and the client it is an object transferred from one to the other. In a further interaction between the client and the printer it is the name of the communication

link. The idea that the names of the links belong to the same category as the transferred objects is one of the cornerstones of the calculus, and is one way in which it is different from other process algebras. In the example a, b, c, d are all just *names* which intuitively represent access rights: a accesses the printer, b accesses the server, d accesses some data, and c is a placeholder for an access to arrive along a. If a is the only way to access the printer then we can say that the printer "moves" to the client, since after the interaction nothing else can access it. For this reason the π-calculus has been called a calculus of "mobile" processes. But the calculus is much more general than that. The printer may have many links that make it do different things, and the server can send these links to different clients to establish different access capabilities to a shared resource.

At first sight it appears as if the π-calculus is just a specialised form of a value-passing process algebra where the values are links. In such a comparison the calculus may be thought rather poor since there are no data types and no functions defined on the names; the transferable entities are simple atomic things without any internal structure. The reason that the π-calculus nevertheless is considered more expressive is that it admits migrating local scopes. This important point deserves an explanation here.

Most process algebras have a way to declare a communication link local to a set of processes. For example in CCS the fact that P and Q share a private port a is symbolised by $(P|Q)\backslash a$, where the operator $\backslash a$ is called *restriction* on a. The significance is that no other process can use the local link a, as if it were a name distinct from all other names in all processes.

In the π-calculus this restriction is written $(\boldsymbol{\nu} a)(P|Q)$. It is similar in that no other process can use a immediately as a link to P or Q. The difference is that the name a is also a transferable object and as such can be sent, by P or Q, to another process which then can use the restricted link. Returning to the example above suppose that a is a local link between the server and the printer. Represent the printer by R, then this is captured by $(\boldsymbol{\nu} a)(\overline{b}a \,.\, S \mid R)$. The server is still free to send a along b to the client. The result would be a private link shared between all three processes, but still distinct from any other name in any other process, and the transition is consequently written

$$(\boldsymbol{\nu} a)(\overline{b}a \,.\, S \mid R) \;\mid\; b(c) \,.\, \overline{c}d \,.\, P \quad \xrightarrow{\tau} \quad (\boldsymbol{\nu} a)(S \mid R \mid \overline{a}d \,.\, P)$$

So, although the transferable objects are simple atomic things they can also be declared local with a defined scope, and in this way the calculus transcends the ordinary value-passing process algebras. This is also the main source of difficulty in the development of the theory because the scope of an object, as represented by the operands of its restriction, must migrate with the object as it is transferred between processes.

The π-calculus is far from a single well defined body of work. The central idea, a process algebraic definition of link-passing, has been developed in several directions to accommodate specific applications or to determine the effects of various

semantics. Proliferation is certainly a healthy sign for any scientific area although it poses problems for those who wish to get a quick overview. Presumably some readers new to the π-calculus will be satisfied with a compact presentation of a single version, while other may be interested in the spectrum of variations.

This paper aims to serve both these needs. In the following, the even-numbered sections develop a single strand of the calculus. Section 2 presents the syntax and give some small examples of how it is used. In Section 4 we proceed to the semantics in its most common form as a labelled transition system. In Section 6 we consider one of the main definitions of bisimulation and the congruence it induces, and in Section 8 we look at their axiomatisations through syntactic equalities of agents. These sections do not depend on the odd-numbered sections and can be considered as a basic course of the calculus. There will be full definitions and formulations of the central results, and sketches that explain the ideas and structure of the proofs.

Each odd-numbered section presents variations on the material in the preceding one. Thus, in Section 3 we explore different versions of the calculus, such as the effect of varying the operators, and the asynchronous, polyadic, and higher-order calculus. Section 5 treats alternative ways to define the semantics, with different versions of labelled and unlabelled transitions. Section 7 defines a few other common bisimulation equivalences (the π-calculus, like any process algebra, boasts a wide variety of equivalences but in this paper we concentrate on the aspects particular to π), and their axiomatisations are treated in Section 9. In these sections we do not always get a full formal account, but hopefully enough explanations that the reader will gain an understanding of the basic ideas. Finally, Section 10 contains references to other work. We give a brief account of how the calculus evolved and mention other overviews and introductory papers. We also indicate sources for the material treated in this paper.

It must be emphasised that there are some aspects of the π-calculus we do not treat at all, such as modal logics, analysis algorithms, implementations, and ways to use the calculus to model concurrent systems and languages. Also, the different variants can be combined in many ways, giving rise to a large variety of calculi. I hope that after this introduction a reader can explore the field with some confidence.

2 The π-Calculus

We begin with a sequence of definitions and conventions. The reader who makes it to Section 2.3 will be rewarded with small but informative examples.

2.1 Basic Definitions

We assume a potentially infinite set of *names* $\mathcal{N}$, ranged over by $a, b, \ldots, z$, which will function as all of communication ports, variables and data values, and a set of *(agent) identifiers* ranged over by A, each with a fixed nonnegative arity. The *agents*, ranged over by $P, Q, \ldots$ are defined Table 1. From that table we see that the agents can be of the following forms:

1. The empty agent $\mathbf{0}$, which cannot perform any actions.
2. An *Output Prefix* $\overline{a}x\,.\,P$. The intuition is that the name x is sent along the name a and thereafter the agent continues as P. So $\overline{a}$ can be thought of as an output port and x as a datum sent out from that port.
3. An *Input Prefix* $a(x)\,.\,P$, meaning that a name is received along a name a, and x is a placeholder for the received name. After the input the agent will continue as P but with the newly received name replacing x. So a can be thought of as an input port and x as a variable which will get its value from the input along a.
4. A *Silent Prefix* $\tau\,.\,P$, which represents an agent that can evolve to P without interaction with the environment. We use α, β to range over $a(x)$, $\overline{a}x$ and τ and call them *Prefixes*, and we say that $\alpha\,.\,P$ is a *Prefix form*, or sometimes just *Prefix* when this cannot cause confusion.
5. A *Sum* $P + Q$ representing an agent that can enact either P or Q.
6. A *Parallel Composition* $P \mid Q$, which represents the combined behaviour of P and Q executing in parallel. The components P and Q can act independently, and may also communicate if one performs an output and the other an input along the same port.
7. A *Match* `if` $x = y$ `then` P. As expected this agent will behave as P if x and y are the same name, otherwise it does nothing.
8. A *Mismatch* `if` $x \neq y$ `then` P. This agent will behave as P if x and y are *not* the same name, otherwise it does nothing.
9. A *Restriction* $(\boldsymbol{\nu} x)P$. This agent behaves as P but the name x is local, meaning it cannot immediately be used as a port for communication between P and its environment. However, it can be used for communication between components within P.

Prefixes	α	::=	$\overline{a}x$	Output
			$a(x)$	Input
			τ	Silent
Agents	P	::=	$\mathbf{0}$	Nil
			$\alpha\,.\,P$	Prefix
			$P + P$	Sum
			$P \mid P$	Parallel
			`if` $x = y$ `then` P	Match
			`if` $x \neq y$ `then` P	Mismatch
			$(\boldsymbol{\nu} x)P$	Restriction
			$A(y_1, \ldots, y_n)$	Identifier
Definitions			$A(x_1, \ldots, x_n) \stackrel{\text{def}}{=} P$	(where $i \neq j \Rightarrow x_i \neq x_j$)

Table 1: The syntax of the π-calculus.

10. An *Identifier* $A(y_1, \ldots, y_n)$ where n is the arity of A. Every Identifier has a *Definition* $A(x_1, \ldots, x_n) \stackrel{\text{def}}{=} P$ where the x_i must be pairwise distinct, and the intuition is that $A(y_1, \ldots, y_n)$ behaves as P with y_i replacing x_i for each i. So a Definition can be thought of as a process declaration, $x_1, \ldots, x_n$ as formal parameters, and the Identifier $A(y_1, \ldots, y_n)$ as an invocation with actual parameters $y_1, \ldots, y_n$.

The operators are familiar from other process algebras so we shall in the following concentrate on some important aspects particular to the π-calculus, trusting the reader to be confident with the more general principles.

The forms Nil, Sum and Parallel have exactly the same meaning and use as in other process algebras, and the Prefix forms are as in the algebras that admit value-passing. The `if` constructs Match and Mismatch may appear limited in comparison with value-passing algebras which usually admit arbitrary Boolean expressions (evaluating to either true or false). But on closer consideration it is apparent that combinations of Match and Mismatch are the only possible tests that can be performed in the π-calculus: the objects transmitted are just names and these have no structure and no operators are defined on them, so the only thing we can do is compare names for equality. We can combine such tests conjunctively by nesting them, for example

$$\texttt{if } x = y \texttt{ then if } u \neq v \texttt{ then } P$$

behaves as P if both $x = y$ and $u \neq v$ hold. We can combine them disjunctively by using Sum, for example

$$\texttt{if } x = y \texttt{ then } P + \texttt{if } u \neq v \texttt{ then } P$$

behaves as P if at least one of $x = y$ and $u \neq v$ hold. Sometimes we shall use a binary conditional

$$\texttt{if } x = y \texttt{ then } P \texttt{ else } Q$$

as an abbreviation for $\texttt{if } x = y \texttt{ then } P + \texttt{if } x \neq y \texttt{ then } Q$.

As in other algebras we say that P is *guarded* in Q if P is a proper subterm of a Prefix form in Q. Also, the input Prefix $a(x).P$ is said to *bind* x in P, and occurrences of x in P are then called *bound*. In contrast the output Prefix $\overline{a}x.P$ does not bind x. These Prefixes are said to have *subject* a and *object* x, where the object is called *free* in the output Prefix and *bound* in the input Prefix. The silent Prefix τ has neither subject nor object.

The Restriction operator $(\boldsymbol{\nu} x)P$ also binds x in P. Its effect is as in other algebras (where it is written $\backslash x$ in CCS and δ_x in ACP) with one significant difference. In ordinary process algebras the things that are restricted are port names and these cannot be transmitted between agents. Therefore the restriction is static in the sense that the scope of a restricted name does not need to change when an agent executes. In the π-calculus there is no difference between "port names" and "values", and a name that represents a port can indeed be transmitted between agents. If that name is restricted the scope of the restriction must change, as we shall see, and indeed almost all of the increased complexity and expressiveness of the π-calculus over value-passing algebras come from the fact that restricted things move around. The reader may also think of $(\boldsymbol{\nu} x)P$ as "new x in P", by analogy with the object-oriented use of the word "new", since this construct can be thought of as declaring a new and hitherto unused name, represented by x for the benefit of P.

In summary, both input Prefix and Restriction bind names, and we can define the *bound names* $\texttt{bn}(P)$ as those with a bound occurrence in P and the *free names* $\texttt{fn}(P)$ as those with a not bound occurrence, and similarly $\texttt{bn}(\alpha)$ and $\texttt{fn}(\alpha)$ for a Prefix α. We sometimes write $\texttt{fn}(P, Q)$ to mean $\texttt{fn}(P) \cup \texttt{fn}(Q)$, and just α for $\texttt{fn}(\alpha) \cup \texttt{bn}(\alpha)$ when it is apparent that it represents a set of names, such as in "$x \in \alpha$". In a Definition $A(x_1, \ldots, x_n) \stackrel{\text{def}}{=} P$ we assume that $\texttt{fn}(P) \subseteq \{x_1, \ldots, x_n\}$. In some examples we shall elide the parameters of Identifiers and Definitions when they are unimportant or can be inferred from context.

A *substitution* is a function from names to names. We write $\{x/y\}$ for the substitution that maps y to x and is identity for all other names, and in general $\{x_1 \ldots x_n / y_1 \ldots y_n\}$, where the y_i are pairwise distinct, for a function that maps each y_i to x_i. We use σ to range over substitutions, and sometimes write $\tilde{x}$ for a sequence of names when the length is unimportant or can be inferred from

context. The agent $P\sigma$ is P where all free names x are replaced by $\sigma(x)$, with alpha-conversion wherever needed to avoid captures. This means that bound names are renamed such that whenever x is replaced by $\sigma(x)$ then the so obtained occurrence of $\sigma(x)$ is free. For example,

$$(a(x)\,.\,(\boldsymbol{\nu} b)\overline{x}b\,.\,\overline{c}y\,.\,\mathbf{0})\{xb/yc\} \quad \text{is} \quad a(z)\,.\,(\boldsymbol{\nu} d)\overline{z}d\,.\,\overline{b}x\,.\,\mathbf{0}$$

A process algebra fan may have noticed that one common operator is not present in the π-calculus: that of relabelling (in CCS written $[a/b]$). The primary use of relabelling is to define instances of agents from other agents, for example, if B is a buffer with ports i and o then $B[i'/i, o'/o]$ is a buffer with ports i' and o'. In the π-calculus we will instead define instances through the parameters of the Identifiers, so for example a buffer with ports i and o is $B(i, o)$, and with ports i' and o' it is $B(i', o')$. For injective relabellings this is just another style of specification which allows us to economise on one operator. (A reader familiar with the CCS relabelling should be warned that it has the same effect as port substitution only if injective. In general they are different.)

Finally some notational conventions: A sum of several agents $P_1 + \cdots + P_n$ is written $\sum_{i=1}^{n} P_i$, or just $\sum_j P_j$ when n is unimportant or obvious, and we here allow the case $n = 0$ when the sum means $\mathbf{0}$. A sequence of distinct Restrictions $(\boldsymbol{\nu} x_1)\cdots(\boldsymbol{\nu} x_n)P$ is often abbreviated to $(\boldsymbol{\nu} x_1 \cdots x_n)P$. In a Prefix we sometimes elide the object if it is not important, so $a\,.\,P$ means $a(x)\,.\,P$ where x is a name that is never used, and similarly for output. And we sometimes elide a trailing $\mathbf{0}$, writing α for the agent $\alpha\,.\,\mathbf{0}$, where this cannot cause confusion. We give the unary operators precedence over the binary and $\mid$ precedence over $+$, so for example $(\boldsymbol{\nu} x)P \mid Q + R$ means $(((\boldsymbol{\nu} x)P) \mid Q) + R$.

2.2 Structural Congruence

The syntax of agents is in one sense too concrete. For example, the agents $a(x)\,.\,\overline{b}x$ and $a(y)\,.\,\overline{b}y$ are syntactically different, although they only differ in the choice of bound name and therefore intuitively represent the same behaviour: an agent that inputs something along a and then sends that along $\overline{b}$. As another example the agents $P|Q$ and $Q|P$ represent the same thing: a parallel composition of the agents P and Q. Our intuition about parallel composition is that it is inherently unordered, and we are forced to syntactically distinguish between $P|Q$ and $Q|P$ only because our language is linear.

We therefore introduce a *structural congruence* to identify the agents which intuitively represent the same thing. It should be emphasised that this has nothing to do with the traditional behavioural equivalences in process algebra which are defined in terms of the behaviour exhibited by an agent under some operational semantics. We have yet to define a semantics, and the structural congruence identifies only agents where it is immediately obvious from their *structure* that they are the same.

The structural congruence $\equiv$ is defined as the smallest congruence satisfying the following laws:

1. If P and Q are variants of alpha-conversion then $P \equiv Q$.
2. The Abelian monoid laws for Parallel: commutativity $P|Q \equiv Q|P$, associativity $(P|Q)|R \equiv P|(Q|R)$, and $\mathbf{0}$ as unit $P|\mathbf{0} \equiv P$; and the same laws for Sum.
3. The unfolding law $A(\tilde{y}) \equiv P\{\tilde{y}/\tilde{x}\}$ if $A(\tilde{x}) \stackrel{\text{def}}{=} P$.
4. The scope extension laws

$$
\begin{array}{lcll}
(\boldsymbol{\nu} x)\mathbf{0} & \equiv & \mathbf{0} & \\
(\boldsymbol{\nu} x)(P \mid Q) & \equiv & P \mid (\boldsymbol{\nu} x)Q & \text{if } x \notin \mathtt{fn}(P) \\
(\boldsymbol{\nu} x)(P + Q) & \equiv & P + (\boldsymbol{\nu} x)Q & \text{if } x \notin \mathtt{fn}(P) \\
(\boldsymbol{\nu} x)\mathtt{if}\ u = v\ \mathtt{then}\ P & \equiv & \mathtt{if}\ u = v\ \mathtt{then}\ (\boldsymbol{\nu} x)P & \text{if } x \neq u \text{ and } x \neq v \\
(\boldsymbol{\nu} x)\mathtt{if}\ u \neq v\ \mathtt{then}\ P & \equiv & \mathtt{if}\ u \neq v\ \mathtt{then}\ (\boldsymbol{\nu} x)P & \text{if } x \neq u \text{ and } x \neq v \\
(\boldsymbol{\nu} x)(\boldsymbol{\nu} y)P & \equiv & (\boldsymbol{\nu} y)(\boldsymbol{\nu} x)P &
\end{array}
$$

Table 2: The definition of structural congruence.

The reader will here correctly object that "represent the same thing" and "immediately obvious" are not formally defined concepts, and indeed several different versions of the structural congruence can be found in the literature; there is no canonical definition and each has different merits. In Section 5.1 we will meet some of them and explore their consequences. Until then we adopt a particular structural congruence. The definition is given in Table 2. We briefly comment on the clauses in the definition.

1. Alpha-conversion, i.e., choice of bound names, identifies agents like $a(x)\,.\,\overline{b}x$ and $a(y)\,.\,\overline{b}y$.
2. The Abelian monoid laws mean that Parallel and Sum are unordered. For example, when we think of a composition of three agents P,Q,R it does not matter if we write it as $(P|Q)|R$ or $(R|Q)|P$. The same holds for Sum. The fact that $\mathbf{0}$ is a unit means that $P|\mathbf{0} \equiv P$ and $P + \mathbf{0} \equiv P$, something which follows from the intuition that $\mathbf{0}$ is empty and therefore contributes nothing to a Parallel composition or Sum.
3. The unfolding just says that an Identifier is the same as its Definition, with the appropriate parameter instantiation.
4. The scope extension laws come from our intuition that $(\boldsymbol{\nu} x)P$ just says that x is a new unique name in P; it can be thought of as marking the occurrences

of x in P with a special colour saying that this is a local name. It then does not really matter where the symbols "$(\boldsymbol{\nu} x)$" are placed as long as they mark the same occurrences. For example, in $\mathbf{0}$ there are no occurrences so the Restriction can be removed at will. In Parallel composition, if all occurrences are in one of the components then it does not matter if the Restriction covers only that component or the whole composition.

Note that we do *not* have that $(\boldsymbol{\nu} x)(P \mid Q) \equiv (\boldsymbol{\nu} x)P \mid (\boldsymbol{\nu} x)Q$. The same occurrences are restricted in both agents, but in $(\boldsymbol{\nu} x)(P \mid Q)$ they are restricted by the *same* binder (or if you will, coloured by the same colour), meaning that P and Q can interact using x, in contrast to the situation in $(\boldsymbol{\nu} x)P \mid (\boldsymbol{\nu} x)Q$.

Through a combination of these laws we get that $(\boldsymbol{\nu} x)P \equiv P$ if $x \notin \mathtt{fn}(P)$:

$$P \equiv P \mid \mathbf{0} \equiv P \mid (\boldsymbol{\nu} x)\mathbf{0} \equiv (\boldsymbol{\nu} x)(P \mid \mathbf{0}) \equiv (\boldsymbol{\nu} x)P$$

So as a special case we get $(\boldsymbol{\nu} x)(\boldsymbol{\nu} x)P \equiv (\boldsymbol{\nu} x)P$ for all P.

Another key fact is that all unguarded Restrictions can be pulled out to the top level of an agent:

Proposition 1 *Let P be an agent where $(\boldsymbol{\nu} x)Q$ is an unguarded subterm. Then P is structurally congruent to an agent $(\boldsymbol{\nu} x')P'$ where P' is obtained from P by replacing $(\boldsymbol{\nu} x)Q$ with $Q\{x'/x\}$, for some name x' not occurring in P.*

The proof is by alpha-converting all bound names so that they become syntactically distinct, and then applying scope extension (from right to left) to move the Restriction to the outermost level. This corresponds to the intuition that instead of declaring something as local it can be given a syntactically distinct name: the effect is the same in that nothing else can access the name.

Our scope extension laws are in fact chosen precisely such that Proposition 1 holds. For example, we have not given any scope extension law for Prefixes and can therefore only pull out unguarded Restrictions. The reader may have expected a law like $(\boldsymbol{\nu} x)\alpha \,.\, P \equiv \alpha \,.\, (\boldsymbol{\nu} x)P$ for $x \notin \alpha$. Indeed such a law would be sound, in the sense that it conforms to intuition and does not disrupt any of the results in this paper, and it will hold for the behavioural equivalences explored later in sections 6 and 7. But it will not be necessary at this point, in particular it is not necessary to prove Proposition 1.

Structural congruence is much stronger, i.e., identifies fewer agents, than any of the behavioural equivalences. The structural congruence is used in the definition of the operational semantics, which in turn is used to define the behavioural equivalences. The main technical reasons for taking this route are that many of the following definitions and explanations become simpler and that we get a uniform treatment for those variants of the calculus that actually require a structural congruence. In Section 5.1 we comment on the possibility to define the calculus without a structural congruence.

2.3 Simple Examples

Although we shall not present the operational semantics just yet (a reader who wishes to look at it now will find it in Section 4) it might be illuminating to see some examples of the scope migration mentioned in Section 1, that Restrictions move with their objects. Formally, scope migration is a consequence of three straightforward postulates. The first is the usual law for inferring interactions between parallel components. This is present in most process algebras and implies that

$$a(x)\,.\,\overline{c}x \mid \overline{a}b \quad \stackrel{\tau}{\longrightarrow} \quad \overline{c}b \mid \mathbf{0}$$

or in general

$$a(x)\,.\,P \mid \overline{a}b\,.\,Q \quad \stackrel{\tau}{\longrightarrow} \quad P\{b/x\} \mid Q$$

The second postulate is that Restrictions do not affect silent transitions. $P \stackrel{\tau}{\longrightarrow} Q$ represents an interaction between the components of P, and a Restriction $(\boldsymbol{\nu}x)P$ only restricts interactions between P and its environment. Therefore $P \stackrel{\tau}{\longrightarrow} Q$ implies $(\boldsymbol{\nu}x)P \stackrel{\tau}{\longrightarrow} (\boldsymbol{\nu}x)Q$. The third postulate is that structurally congruent agents should never be distinguished and thus any semantics must assign them the same behaviour. Now what are the implications for restricted objects? Suppose that b is a restricted name, i.e., that we are considering a composition

$$a(x)\,.\,\overline{c}x \mid (\boldsymbol{\nu}b)\overline{a}b$$

Will there be an interaction between the components and if so what should it be? Structural congruence gives the answer, because b is not free in the left hand component so the agent is by scope extension structurally congruent to

$$(\boldsymbol{\nu}b)(a(x)\,.\,\overline{c}x \mid \overline{a}b)$$

and this agent has a transition between the components: because of

$$a(x)\,.\,\overline{c}x \mid \overline{a}b \quad \stackrel{\tau}{\longrightarrow} \quad \overline{c}b \mid \mathbf{0}$$

we get that

$$(\boldsymbol{\nu}b)(a(x)\,.\,\overline{c}x \mid \overline{a}b) \quad \stackrel{\tau}{\longrightarrow} \quad (\boldsymbol{\nu}b)(\overline{c}b \mid \mathbf{0})$$

and the rightmost $\mathbf{0}$ can be omitted by the monoid laws. So by identifying structurally congruent agents we obtain that

$$a(x)\,.\,\overline{c}x \mid (\boldsymbol{\nu}b)\overline{a}b \quad \stackrel{\tau}{\longrightarrow} \quad (\boldsymbol{\nu}b)\overline{c}b$$

or in general that, provided $b \notin \mathtt{fn}(P)$,

$$a(x)\,.\,P \mid (\boldsymbol{\nu}b)\overline{a}b\,.\,Q \quad \stackrel{\tau}{\longrightarrow} \quad (\boldsymbol{\nu}b)(P\{b/x\} \mid Q)$$

In other words, the scope of $(\boldsymbol{\nu}b)$ "moves" with b from the right hand component to the left. This phenomenon is sometimes called scope extrusion. If $b \in \mathtt{fn}(P)$ a

similar interaction is possible by first alpha-converting the bound b to some name $b' \notin \mathtt{fn}(P)$, and we would get

$$a(x)\,.\,P \mid (\boldsymbol{\nu} b)\overline{a}b\,.\,Q \quad \stackrel{\tau}{\longrightarrow} \quad (\boldsymbol{\nu} b')(P\{b'/x\} \mid Q\{b'/b\})$$

So $P\{b'/x\}$ still contains b free and it is not the same as the received restricted name b'.

For another example consider:

$$((\boldsymbol{\nu} b)a(x)\,.\,P) \mid \overline{a}b\,.\,Q$$

Here the right hand component has a free b which should not be the same as the bound b to the left. Is there an interaction between the components? We cannot immediately extend the scope to the right hand component since it has b free. But we can first alpha-convert the bound b to some new name b' and then extend the scope to obtain

$$(\boldsymbol{\nu} b')(a(x)\,.\,P\{b'/b\} \mid \overline{a}b\,.\,Q)$$

and it is clear that we have a transition

$$(\boldsymbol{\nu} b')(a(x)\,.\,P\{b'/b\} \mid \overline{a}b\,.\,Q) \stackrel{\tau}{\longrightarrow} (\boldsymbol{\nu} b')P\{b'/b\}\{b/x\} \mid Q$$

So the restricted name, now b', will still be local to the left hand component; the attempt to intrude the scope is thwarted by an alpha-conversion. In summary, through alpha-conversion and scope extension we can send restricted names as objects, and Restrictions will always move with the objects and never include free occurrences of that name.

This ability to send scopes along with restricted names is what makes the calculus convenient for modelling exchange of private resources. For example, suppose we have an agent R representing a resource, say a printer, and that it is controlled by a server S which distributes access rights to R. In the simplest case the access right is just to execute R. This can be modelled by introducing a new name e as a trigger, and guarding R by that name, as in

$$(\boldsymbol{\nu} e)(S \mid e\,.\,R)$$

Here R cannot execute until it receives a signal on e. The server can invoke it by performing an action $\overline{e}$, but moreover, the server can send e to a client wishing to use R. For example, suppose that a client Q needs the printer. It asks S along some predetermined channel c for the access key, here e, to R, and only upon receipt of this key can R be executed. We have

$$c(x)\,.\,\overline{x}\,.\,Q \mid (\boldsymbol{\nu} e)(\overline{c}e\,.\,S \mid e\,.\,R) \quad \stackrel{\tau}{\longrightarrow} \quad (\boldsymbol{\nu} e)(\overline{e}\,.\,Q \mid S \mid e\,.\,R) \quad \stackrel{\tau}{\longrightarrow} \quad (\boldsymbol{\nu} e)(Q \mid S \mid R)$$

The first transition means that Q receives an access to R and the second that this access is used. We can informally think of this as if the agent R is transmitted

(represented by its key e) from S to Q, so in a sense this gives us the power of a higher-order communication where the objects are agents and not only names. But our calculus is more general since a server can send e to many clients, meaning that these will share R (rather than receiving separate copies of R). And R can have several keys that make it do different things, for example R can be $e_1 \,.\, R_1 \mid e_2 \,.\, R_2 \cdots$, and the server can send only some of the keys to clients and retain some for itself, or send different keys to different clients representing different access privileges.

A related matter is if S wishes to send two names d and e to a client, and insure that the same client receives both names. If there are several clients then the simple solution of transmitting d and e along predetermined channels may mean that one client receives d and another e. A better solution is to first establish a private channel with a client and then send d and e along that channel. The private channel is simply a restricted name:

$$(\boldsymbol{\nu} p)\overline{c}p \,.\, \overline{p}d \,.\, \overline{p}e \,.\, S$$

A client interacting with C must be prepared to receive a name, and then along that name receive d and e:

$$c(p) \,.\, p(x) \,.\, p(y) \,.\, Q$$

Now, even if we have a composition with several clients and a server, the only possibility is that d and e end up with the same client. This feature is so common that we introduce an abbreviation for it:

$$\begin{array}{lll} \overline{c}\langle e_1 \cdots e_n \rangle \,.\, P & \text{means} & (\boldsymbol{\nu} p)\overline{c}p \,.\, \overline{p}e_1 \,.\, \cdots \,.\, \overline{p}e_n \,.\, P \\ c(x_1 \cdots x_n) \,.\, Q & \text{means} & c(p) \,.\, p(x_1) \,.\, \cdots \,.\, p(x_n) \,.\, Q \end{array}$$

where we choose $p \notin \mathtt{fn}(P, Q)$ and all x_i are pairwise distinct. We will then have

$$\overline{c}\langle e_1 \cdots e_n \rangle \,.\, P \mid c(x_1 \cdots x_n) \,.\, Q \quad \stackrel{\tau}{\longrightarrow} \cdots \stackrel{\tau}{\longrightarrow} \quad P \mid Q\{e_1 \ldots e_n \,/\, x_1 \ldots x_n\}$$

The idea to establish private links in this way has many other uses. Suppose for example that Q wishes to execute P by transmitting on its trigger e, and then also wait until P has completed execution. One way to represent this is to send to P a private name for signalling completion, as in

$$(\boldsymbol{\nu} r)\overline{e}r \,.\, r \,.\, Q \mid e(x) \,.\, P \quad \stackrel{\tau}{\longrightarrow} \quad (\boldsymbol{\nu} r)(r \,.\, Q \;\mid\; P\{r/x\})$$

Here Q must wait until someone signals on r before continuing. This someone can only be P since no other is in the scope of r. This scheme is quite general, for example P can delegate to another agent the task to restart Q, by sending r to it as an object in an interaction.

The π-calculus has been used to succinctly describe many aspects of concurrent and functional programming, and also of high-level system description where mobility plays an important role. We shall not attempt an overview of all applications here. In the rest of this paper we concentrate on some central aspects of the theory of the calculus.