

Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms

Davide Sangiorgi

Doctor of Philosophy

University of Edinburgh

1992

graduation date: February 1993

Abstract

We study *mobile systems*, i.e. systems with a dynamically changing communication topology, from a process algebras point of view. Mobility can be introduced in process algebras by allowing names or terms to be transmitted. We distinguish these two approaches as *first-order* and *higher-order*. The major target of the thesis is the comparison between them.

The prototypical calculus in the first-order paradigm is the π -calculus. By generalising its sort discipline we derive an ω -order extension called *Higher-Order π -calculus* ($\text{HO}\pi$). We show that such an extension does not add expressiveness to the π -calculus: Higher-order processes can be faithfully compiled down to first-order, and respecting the behavioural equivalence we adopted in the calculi. Such an equivalence is based on the notion of bisimulation, a fundamental concept of process algebras. Unfortunately, the standard definition of bisimulation is unsatisfactory in a higher-order calculus because it is over-discriminating. To overcome the problem, we propose *barbed bisimulation*. Its advantage is that it can be defined *uniformly* in different calculi because it only requires that the calculus possesses an *interaction* or *reduction* relation. As a test for barbed bisimulation, we show that in CCS and π -calculus, it allows us to recover the familiar bisimulation-based equivalences. We also give simpler characterisations of the equivalences utilised in $\text{HO}\pi$. For this we exploit a special kind of agents called *triggers*, with which it is possible to reason fairly efficiently in a higher-order calculus notwithstanding the complexity of its transitions.

Finally, we use the compilation from $\text{HO}\pi$ to π -calculus to investigate Milner's encodings of λ -calculus into π -calculus. We present analogous encodings of λ -calculus into $\text{HO}\pi$. By comparison with those into π -calculus, these are easier to

understand and with a closer correspondence between reduction on λ -terms and on their process counterparts. We show that the two encodings of lazy λ -calculus and the compilation from $\text{HO}\pi$ to π -calculus commute. Thus we can reason in $\text{HO}\pi$ and the results hold for π -calculus as well. In this way we are able to derive a direct characterisation of the equivalence upon λ -terms induced by Milner's encoding and by the behavioural equivalence adopted on the process terms.

From the representability result of $\text{HO}\pi$ in π -calculus 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.

Declaration

This thesis was composed by myself, and the work contained in it is my own unless otherwise stated.

To my parents
and to the memory of my grandmother Rambelli Maria

Acknowledgements

I simply cannot express how much I am indebted to my supervisor Robin Milner. He was always willing to discuss my work and ready to give enlightening suggestions. His precious guidance has deeply marked not only my research, but also my way of doing research; and his enthusiasm has kept me alive throughout the development of the thesis.

I also owe a lot to my former Italian supervisor, Andrea Maggiolo-Schettini: For convincing me to apply to Edinburgh when I was about to join some company and afterwards, for his constant support and friendship. Thanks also to Emanuela Fachini and Rocco De Nicola for their encouragement.

The Computer Science Department in Edinburgh has offered a stimulating and exiting environment for my work. In particular, I would like to thank the past and present members of the Concurrency Club, for the fruitful feedback and many interesting discussions.

Thanks to Stuart Anderson, Glenn Bruns, Faron Moller, Andrea Maggiolo-Schettini, Peter Sewell and David Turner who read (parts of) a draft of this thesis and helped me with my (hopeless) English.

Finally, I am most grateful to all friends who made my stay in Edinburgh enjoyable; in particular Claudio, David, Ian, Marcelo, Roberto, Søren. A special mention to Laurence, for her company and for having helped me to stay serene during the final writing of the thesis.

The work has been supported by a scholarship from the Consiglio Nazionale delle Ricerche, Italy.

★ ★ ★

This is a revised version of my thesis; I have benefited a lot from the detailed criticism of my examiners Matthew Hennessy and Gordon Plotkin.

Main Notations

The following tables summarise the notation used in this thesis. The page numbers refer to the page where the notation is first defined or used.

Process Theory

Symbol	Description	Page
P, Q, R, T	processes	18
A	agents	20
F, G, E	abstractions	20
$C[\cdot]$	context	21
a, b, \dots	names	18
$\mathbf{0}$	inactive process	19
$\sum_{i \in I} \alpha_i.P_i$	process sum	18
$P + Q$	binary sum	19
$P \mid Q$	parallel composition	18
$\prod_{i=1}^n P_i$	multiple parallel composition	22
$\nu a P$	restriction	18
$D \stackrel{def}{=} (\tilde{x})P, D \stackrel{def}{=} (\tilde{U})P$	constant definition	18
$D\langle\tilde{x}\rangle, D\langle\tilde{K}\rangle$	constant application	18
$X\langle\tilde{K}\rangle$	variable application	32
$A\langle\tilde{K}\rangle$	meta application	35
$(x)A, (X)A,$	(typical) abstraction	18
$!P$	replication	22
$[x = y]$	matching	18

$\tau.P$	silent prefix	22
$m \star F$	prefix abbreviation	66
$P \{m := F\}$	special replication-abbreviation	66
α	prefix	18
$x(\tilde{y}), x(\tilde{U}),$	input prefix	18
$\bar{x}(\tilde{y}), \bar{x}(\tilde{K})$	output prefix	18
$\{\tilde{x}/\tilde{y}\}, \{\tilde{K}/\tilde{U}\}$	substitution	20
$fn(A), bn(A)$	free and bound names of A	20
s, S	subject and object sorts	23
El	element sort	34
Pr_π	π -calculus processes	18
$HO\pi, HO\pi^\circ$	closed and open $HO\pi$ agents	33
$THO\pi, THO\pi^\circ$	closed and open triggered agents	73

Process Semantics

μ	action	29
$fn(\mu), bn(\mu), n(\mu)$	free names, bound names and names of μ	29
$\xrightarrow{\mu}$	labelled transition system	28
\longrightarrow	reduction relation	27
\Longrightarrow	reflexive and transitive closure of \longrightarrow	39
$=$	equality up-to α -conversion	21
\equiv	structural congruence	26
\sim_e, \approx_e	strong and weak early bisimulation	42
$\dot{\sim}, \dot{\approx}$	strong and weak barbed bisimulation	48
\sim, \approx	strong and weak barbed equivalence	49
\sim^c, \approx^c	strong and weak barbed congruence	48
$\approx_{Ct}, \approx_{Ct}$	strong and weak context bisimulation	62
$\approx_{Tr}, \approx_{Tr}$	strong and weak triggered bisimulation	83
$\approx_{Nr}, \approx_{Nr}$	strong and weak normal bisimulation	88

Table of Contents

Abstract	ii
Acknowledgements	v
Main Notations	vii
1 Introduction	1
1.1 Background	4
1.2 Representability of the higher-order paradigm at the first-order . .	7
1.3 Bisimulation in higher-order process calculi	9
1.4 Encoding of λ -calculus	11
1.5 Summary of the thesis	12
2 From π-calculus to Higher-Order π-calculus	16
2.1 Syntax of π -calculus	17
2.1.1 The language	17
2.1.2 Sorting	23
2.2 Operational Semantics of π -calculus	25
2.2.1 Reduction semantics	26
2.2.2 Labelled transition semantics	28
2.2.3 Correspondence between the two semantics	30
2.3 Syntax of $\text{HO}\pi$	31
2.4 Operational semantics of $\text{HO}\pi$	37
2.5 Some preliminaries about bisimulations	39
2.5.1 Strong and weak bisimulations	39

2.5.2	Congruences	40
2.5.3	Bisimulations up-to	40
2.6	Bisimulation and congruence for π -calculus	41
3	Barbed Bisimulation	44
3.1	Motivations	44
3.2	From reduction bisimulation to barbed bisimulation and congruence	46
3.3	Discriminating power induced by barbed bisimulation	50
3.3.1	The CCS case	50
3.3.2	The π -calculus case	56
4	Bisimulation in $\mathbf{HO}\pi$	57
4.1	The reduced $\mathbf{HO}\pi$	59
4.2	Strong context bisimulation	62
4.2.1	Definition and preliminaries	62
4.2.2	Congruence properties of context bisimulation	64
4.3	Strong context congruence	65
4.3.1	Distributivity properties for replication	66
4.4	Weak context bisimulation and congruence	68
4.4.1	Triggers	69
4.5	Triggered agents	72
4.5.1	Definition of the class	73
4.5.2	The transformation to triggered agents	76
4.6	Triggered bisimulation	82
4.6.1	Weak triggered and context bisimulations coincide	84
4.7	Uses of triggered agents	88
4.7.1	Normal bisimulation	88
4.7.2	Characterisation in terms of barbed bisimulation	93
4.7.3	Some comments on the bisimulations considered	94
5	The Representability of $\mathbf{HO}\pi$ in π-calculus	96
5.1	The encoding of triggered agents	97

5.2	The compilation of $\text{HO}\pi$ into π -calculus	104
5.3	Related work: Thomsen's encoding and higher-order bisimulation	108
6	An Investigation into Functions as Processes	113
6.1	Introduction	113
6.2	λ -calculus: Preliminaries on its syntax and its encoding	116
6.3	The Lazy λ -Calculus Encoding	117
6.3.1	Applicative Bisimulation	118
6.3.2	Encoding into π -calculus	119
6.3.3	Encoding into $\text{HO}\pi$	120
6.3.4	Correspondence between the two encodings	121
6.4	The Call-by-Value Encoding	122
6.4.1	Encoding into π -calculus	122
6.4.2	Encoding into $\text{HO}\pi$	123
6.4.3	Correspondence among the encodings	124
6.5	A λ -model from process terms	129
6.5.1	λ -models	129
6.5.2	The model	130
6.5.3	Extensionality	133
6.6	Full abstraction	135
6.6.1	The restrictive approach	136
6.6.2	The expansive approach	139
7	Conclusions and Future Work	155
A	Proof of Theorem 3.3.3(2)	161
B	Proof of Proposition 4.2.6	165
C	Proof of Proposition 4.3.2	172
D	Proof of Theorem 4.7.5	184
	Bibliography	195

Chapter 1

Introduction

Concurrent systems are one of the most challenging areas of research in computer science. The notion however, is not exclusive to the “computer world”. For instance, a colony of ants and an aggregate of elementary particles are examples from biology and physics. We view a concurrent system as the composition of independent entities — the *processes* — which may interact and exchange messages.

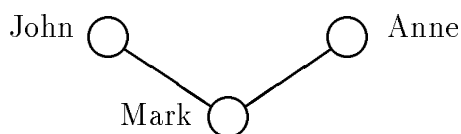
The analysis and description of concurrent systems is a difficult task. The overall behaviour of the system is the result of the interaction of its individual components and their consequent evolution; moreover the occurrence of such interactions may not be deterministically predictable and different interactions may take place at the same time.

Numerous models and methods for reasoning about concurrent systems has been proposed. Among these, *Process Algebra* is one of the most successful. In process algebras both the analysis and the description of a system are carried out in an algebraic setting. Robin Milner’s work on the Calculus of Communicating Systems (CCS) [Mil80] is generally accepted as initiator and landmark in the field. (A more refined version of the theory of CCS is in [Mil89]). Inspired by the λ -calculus, CCS is based upon the selection of a few primitive constructors each embodying a distinct and intuitive idea. But the choice of constructors strikingly differs from λ -calculus. Perhaps this is not too surprising, given the different (and in some sense opposite) objectives of the two calculi. The λ -calculus studies functions and their applicative behaviour; it essentially describes sequential

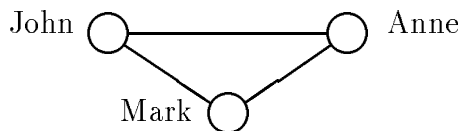
computations. By contrast, CCS aims at parallel computations; interaction and communication are the central idea.

A limitation of CCS (as well as of the other “traditional” process algebras like CSP [Hoa85], ACP [BK84,BK85], and MEIJE [AB84]) is that the set of ports used by a term to perform communications with other terms, is fixed by its syntax. This prevents CCS from effectively describing *mobile* systems, i.e. systems with a dynamically changing communication topology.

To give an example of mobility, consider an abstract view of an electronic mail system with three users John, Mark and Anne in which only John and Anne do not know each other’s address. Pictorially we can view the system as follows:



Suppose that subsequently John and Anne want to communicate with each other. They could do so by going through Mark, but a better way — probably quicker and also safer for their privacy — is to ask Mark for the missing address. This results in the appearance of a connection between John and Anne:



Thus the messages which have been exchanged in the system have affected the communication interface of some of its subcomponents. This is the essence of the notion of mobility.

Mobility is common in operating systems. Take the case of a resource which has a single owner at any time but whose ownership can be changed as time passes; or *process migration* [DO91], where tasks or processes can be exchanged among processors to optimise processor usage. Another example from parallel programming is a procedure which can be called simultaneously by any number of processes; here the problem is to guarantee that each activated instance of the procedure returns its result to the correct calling process. At best this can be

modelled in CCS with a summation indexed over the callers, each of which uses a different port to interact with the procedure. The drawback is that all potential callers must be known in advance; moreover this representation does not capture the mobility which is involved.

The purpose of this thesis is to add to the understanding of the phenomenon of mobility in process algebras. The terms *first-order paradigm* and *higher-order paradigm* identify two approaches to mobility in process algebras. In the higher-order paradigm, mobility is achieved by allowing agents, namely processes or parametrised processes, to be passed as values in a communication; in the first-order paradigm only ports, or names, can be transmitted. The former is closer to λ -calculus, in that a *term* can be bound to a variable; the latter resembles the way in which object oriented programming languages allow objects to refer to one another. In fact, in [Mil90a] the two approaches are called by Milner *function* and *object paradigm*, respectively.

In the next section we give a brief overview of the major attempts to express mobility in models for concurrent systems. Thereafter, we discuss the motivations for the main research directions in the thesis. The comparison between first and higher-order paradigm is a central theme. A considerable amount of effort has been devoted to the development of a higher-order calculus, in particular in its semantics. This because although various higher-order calculi have already appeared, there is not a well-established mathematical treatment of the higher-order paradigm. The situation is different at first-order. The π -calculus is the prototypical calculus and its elegant theory has been explored by Robin Milner, Joachim Parrow and David Walker in [MPW92,MPW91]. Indeed, the higher-order calculus we develop is founded on the π -calculus. Calculi for mobile systems allow elegant representation of the λ -calculus. The investigation into the representation of functions as processes is the topic of the last part of the thesis. The starting point is Milner's work on the encoding of λ -calculus into π -calculus presented in [Mil90a].

1.1 Background

The ancestor of all concurrency models with mobility is probably Hewitt's *actor system* ([Hew77,HB77,Agh86,Agh90]). Actors are active objects which communicate each other via asynchronous messages, themselves thought of as actors. The actor's behaviour, which may be viewed as its local state, specifies the response to an incoming message. An actor can only send messages to those actors of which it has *acquaintance*. The dynamicity of the system derives from the possibility both of creating new actors and of modifying an actor's acquaintance during the computation. The latter because upon receiving a message an actor may add to its acquaintances any name mentioned in the message and because acquaintances may be forgotten during computation. The model has had a considerable success and has given rise to an intensive research on system architecture and object-oriented programming language constructs. However, from a theoretical point of view it has suffered from the lack of a convincing mathematical treatment of its foundational concepts. An example is the mechanism for generating new names for actors and for guaranteeing their unicity. Also the treatment of behavioural equivalences and of system compositionality (according to which a system has a structure and can be analysed in terms of its subcomponent) is not quite satisfactory, especially when compared with their treatment in process algebras. Interesting work developing of a process algebra for actor-like objects is being carried out by Honda and Tokoro (see [HT]).

Graph-grammars [Ehr79] provide an elegant way to model systems with a dynamic behaviour. In [JR89,JR90], Janssens and Rozenberg use graph-grammars to give semantics to a (restricted) version of the actor systems. The nodes of a graph represent actors and the directed edges represent acquaintances; the dynamic evolution of an actor is described in terms of transformations of configuration graphs.

A graph-grammar approach might also be interesting to introduce mobility in *Petri Nets* [Rei85], whose standard definition requires a static, fixed structure. Studies on the connection between Petri Nets and graph-grammars have been carried out among the others, by Kreowsky [Kre80], Pinna and Maggiolo-Schettini [PMS90]; but they do not consider explicitly mobility. Engelfriet, Leih

and Rosenberg take a different direction in [ELR90b,ELR90a]. Their model is a generalisation of Petri Nets aiming at formalising properties of object-based systems. The places of a Petri Net are enriched with extra structure which records the essential state information of the object they represent. For this, three functions are used: One maps each place to the object it represents; another indicates the mode of such object in that state (unborn, alive, dead); the last gives the content of its local memory, possibly including references to other objects. A transition of the Petri Net corresponds to the change of state of one (or more) objects. Creation or destruction of objects and modification of their acquaintance sets are modelled as special instances of such transitions. However, the description of mobility remains rather indirect.

The object paradigm has been investigated in the setting of algebraic parametrised specification by Astesiano and his colleagues in a series of papers ([ARW86, AR87,AG88]). However, to the purposes of this thesis, the most relevant studies dealing with communication of behaviour expressions are [Bou89,Nie89,Tho90].

The γ -calculus introduced by Gerard Boudol in [Bou89] is an extension of λ -calculus with a communication mechanism inspired by CCS. It has two parallel constructs. The first, called interleaving, does not allow communication between agents. The second, called cooperation, forces two agents to communicate on every port and as a consequence, it is non-associative. No restriction or hiding operator is considered. Boudol uses the γ -calculus to represent the λ -calculus. He shows that the latter is a subcalculus of former, in the sense that β -reduction is an instance of the interaction law. The algebraic theory of γ -calculus and its expressiveness in the modelling of concurrent processes remain to be properly investigated.

Flemming Nielson's *Typed Parallel Language* [Nie89] tries to integrate process algebra and the λ -calculus. The focus is mainly on types, as a means of ensuring more reliable programs. Thus the starting point is the *typed λ -calculus* rather than its untyped version. Processes also have types, which record their communication possibilities. Unfortunately the syntax is rather elaborate and the use of types is arduous; for instance it forces a heavy side condition in the definition of parallel

composition. But the attempt towards a notion of type for processes is very interesting and in our view deserves further investigation.

The most conspicuous effort towards the development of an algebra of higher-order processes is by Thomsen [Tho90]. His Calculus of Higher Order Communicating Systems (CHOCS) is an extension of CCS in the sense that the basic operators in the two calculi are the same; the difference is that in CHOCS a communication always causes the exchange of a process. A salient feature of CHOCS is the *dynamic binding* for the restriction operator, as opposed to *static binding* which Thomsen experiments within Plain CHOCS ([Tho90, Chapter 5] and [Tho89]). An example will illustrate the difference. In the notation of this thesis, ' νb ' represents restriction on the name b , ' $\bar{a}\langle P \rangle$ ' the output of the process P , ' $a(X)$ ' input prefixing and ' $|$ ' parallel composition. The following is a legal interaction in CHOCS:

$$a(X).X \mid \nu b (\bar{a}\langle P \rangle.Q) \longrightarrow P \mid \nu b Q$$

The reduction destroys the privacy of the possible b -link between P and Q , since the free occurrences of b in P have evaded the restriction which embraced them. In static binding this is not allowed. Restriction becomes a formal binder like the abstraction operator of λ -calculus and as such, subject to α -conversion. Its peculiarity is that its scope may change because of a communication; thus in Plain CHOCS the above interaction becomes

$$a(X).X \mid \nu b (\bar{a}\langle P \rangle.Q) \longrightarrow \nu b (P \mid Q)$$

The main advantage of dynamic binding is a simple operational semantics. But this does not compensate for the difficulty in the analysis of a program from its text. In this thesis we deal only with static binding.

The merit of being the first to use static binding and to understand how to handle it algebraically goes to Mogens Nielsen and Uffe Engberg [EN86]. Their Extended CCS (ECCS) follows the first-order paradigm, previously attempted by Astesiano and Zucca [AZ84] who used label expressions to emulate parametric channels. The success of ECCS was frustrated by its heavy syntactic definition.

For instance there are three different kinds of variables, ranging over labels, expression behaviours and values, and the interplay between them is often confusing. A simplification of the theory of ECCS was pursued by Robin Milner, Joachim Parrow and David Walker with π -calculus [MPW92]: They achieve an elegant uniform syntactic and semantic framework in which names and agents are the only entities involved. Subsequently, a more refined version of the π -calculus was proposed by Milner [Mil91], where most notably, *sorts* and communication of *tuples* are added. This calculus is at the centre of the attention in this thesis and will be presented in detail in Chapter 2.

We conclude by mentioning firstly Oscar Nierstrasz’s *Object Calculus* [Nie], in which he tries to combine π -calculus and λ -calculus with the purpose of defining a uniform framework for the semantics of concurrent object-oriented languages. Secondly, the languages DyNe [KS85], Nil [SY85], PFL [Hol83], LCCS [Let92] and FACILE [GMP89]. All allow for dynamic linkage reconfiguration, but the emphasis is more on programming language and implementation issues.

1.2 Representability of the higher-order paradigm at the first-order

What makes the π -calculus attractive is the strength of its elementary theory and its reduced complexity. The choice of the first-order paradigm was motivated by the belief that reference-passing is enough to represent more involved operations like process-passing. This thesis validates that claim, showing that no power — only convenience — is gained by moving to the higher-order paradigm.

To this end, we introduce a new calculus, called *Higher-Order π -calculus* ($\text{HO}\pi$), which enriches the π -calculus with explicit higher-order communications. In the $\text{HO}\pi$ 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 Thomsen’s Plain CHOCS is of second-order, then $\text{HO}\pi$ is of ω -order. We show that the $\text{HO}\pi$ is *representable* within the π -calculus. This proves that the first-order paradigm, being by far simpler, should be taken as *basic*.

Such a conclusion takes away the interest in the opposite direction, namely the representability of the π -calculus within a language using purely communications of agents, which in fact has not been investigated in this thesis.

But what does it mean that a given *source* language is representable within a given *target* language? We can identify essentially 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.

As regards (2), it is enough to add that for practical usefulness, the encoding — besides being reasonably “simple” — must be *compositional*, that is its definition on a term should only depend upon the definition on its immediate constituents.

Some further comment on (1) and (3) is worthwhile. There are two major approaches for giving semantics to a programming language. In the *denotational approach* a valuation function maps a program directly to its mathematical meaning or *denotation*. Here the correctness of an encoding can be investigated by considering the relationship between the meaning of a term in the source language and the meaning of its translation into the target language. Denotational semantics has been very successful in modelling many sequential languages; programs are typically viewed as computational functions from the domain of input values to the domain of output values. However, to date there has not been an equally satisfactory denotational treatment of concurrency. The tools employed for sequential programs are inadequate, since an account of the behaviour of a concurrent system must also take into account the *internal* states it can reach.

The predominant approach to the semantics of concurrent systems is *operational*. Following Plotkin [Plo81], the possible evolutions of a process are described in terms of a *transition system* in which the rules are defined inductively on the *structure* of a term. The major drawback of operational semantics is that it is too concrete. Thus, transition systems have to be quotiented by *equivalence* relations which abstract from unwanted details. The operational method necessitates a different approach to translation-correctness, where behaviours rather than meanings are compared. For the purpose of proving the correctness of the encoding,

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 $\text{HO}\pi$ and the target language is π -calculus, this allows us on the one hand to make use of the abstraction power of $\text{HO}\pi$, 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 $\text{HO}\pi$.

1.3 Bisimulation in higher-order process calculi

Behavioural equivalences proposed for process algebras may be classified as *extensional* (or *interleaving*) where concurrency is reduced to sequentiality plus non-determinism, or as *intensional* (or *true concurrent*) where parallel composition is considered a primitive operator. The transport of a well-established equivalence to the higher-order setting is not always straightforward. We focus here on (interleaving) bisimulation equivalence, widely accepted as the finest extensional equivalence one would want to impose. It was originally proposed by Milner and Park [Mil80,Par81] in the early 80’s and since then has become one of the most stable concepts in the theory of concurrency.

Traditionally, a bisimulation is defined on top of a *labelled* transition system,

and requires that an action be matched by another only if they have identical labels. But this is certainly too strong in calculi expressing mobility. For instance it does not respect α -conversion. Now, with name passing the damage is limited to this and can be repaired by adding appropriate side conditions in the definition of bisimulation; but in higher-order calculi the damage goes well beyond. Obvious algebraic laws, such as the commutativity of parallel composition, are lost: For instance take the processes $\bar{a}\langle P \mid Q \rangle.\mathbf{0}$ and $\bar{a}\langle Q \mid P \rangle.\mathbf{0}$ (where $\mathbf{0}$ represents the inactive or null process). They can be distinguished because the first performs an output action which transmits $P \mid Q$, whereas the second transmits $Q \mid P$, which in general is syntactically different from $P \mid Q$. The approach taken by Thomsen in [Tho90], following earlier ideas by Astesiano and Boudol [AG88,Bou89], is to require *bisimilarity* rather than *identity* of the processes emitted in a higher-order output action (*higher-order bisimulation*). This seems a conceivable requirement, but one can object that it is still too strong. To see this, take

$$P \stackrel{def}{=} \bar{a}\langle \mathbf{0} \rangle.\mathbf{0} \qquad Q \stackrel{def}{=} (\nu x)\bar{a}\langle \bar{x}.\mathbf{0} \rangle.\mathbf{0}$$

The processes P and Q differ in the value carried by \bar{a} , which is $\mathbf{0}$ in the former, and $\bar{x}.\mathbf{0}$ in the latter. But since the name x is restricted, process $\bar{x}.\mathbf{0}$ will never find a partner to communicate with. It is therefore a deadlocked process, and as such semantically the same as $\mathbf{0}$. Hence, in any context P and Q give rise to the same interactions, and accordingly, should be considered equivalent. Unfortunately, they are distinguished by higher-order bisimulation, since $\bar{x}.\mathbf{0}$ and $\mathbf{0}$ are not equivalent. One could think of adjusting this example by imposing a different treatment of the outermost restriction in Q , thus comparing $\mathbf{0}$ with $(\nu x)\bar{x}.\mathbf{0}$ rather than $\bar{x}.\mathbf{0}$. But this is certainly wrong and in fact would be disastrous in other situations. For instance, it would equate the processes

$$(\nu x)\bar{a}\langle \bar{x}.R \rangle.x \qquad \text{and} \qquad (\nu x)\bar{a}\langle \mathbf{0} \rangle.x$$

which by contrast may have completely different possibilities of interactions (the former can communicate at x with the recipient of $\langle \bar{x}.R \rangle$ and thus activate a copy of R). This choice would also yield the law

$$(\nu b)\bar{a}\langle P \rangle.Q = \bar{a}\langle \nu b P \rangle.Q$$

Yet in the first process all copies of P activated by its recipient share the name b , whereas in the second one, the name b is private to each copy.

Our aim though, was not merely to solve this specific problem of higher-order process algebras. An important requirement for us was that the solution proposed could be used *uniformly* in different calculi. First of all because this would provide us with a fundamental tool for comparing them, the kind of issue with which this thesis is mainly concerned. Secondly, because it would reveal something of the “essence” of a natural bisimulation and help us to understand whether bisimulations put forward in various process algebras are indeed instances of a more general and common notion. We claim to have achieved this by introducing the notion of *barbed bisimulation*, which focuses on the interaction (or reduction) relation of the calculus. There are other reasons for our interest in barbed bisimulation; for these we refer to Section 3.1.

1.4 Encoding of λ -calculus

In [Mil90a] Milner examines the encoding of the λ -calculus into the π -calculus; more precisely, he shows how the *lazy* and *call-by-value* λ -evaluation strategies [Abr89, Plo75] can be faithfully mimicked.

We have made use of the representability of the $\text{HO}\pi$ in the π -calculus to investigate these encodings. The clue for this is the following. W.r.t. the π -calculus, the higher-order machinery of the $\text{HO}\pi$ gives a simpler description of λ -calculus, with a closer correspondence between reduction on λ -terms and on their process counterparts. Such a $\text{HO}\pi$ description is mapped by our compilation from $\text{HO}\pi$ into π -calculus directly — at least for lazy λ -calculus — onto Milner’s encoding. That is, if \mathcal{P} and \mathcal{H} are respectively, the π -calculus and $\text{HO}\pi$ encoding, and \mathcal{C} is the compilation from $\text{HO}\pi$ to π -calculus, then the following diagram commutes:

$$\begin{array}{ccc}
 \lambda & \xrightarrow{\mathcal{H}} & \text{HO}\pi \\
 \searrow \mathcal{P} & & \downarrow \mathcal{C} \\
 & & \pi
 \end{array}$$

Thus one can reason with \mathcal{H} and infer all results obtained onto \mathcal{P} as well.

Our study of the λ -calculus encodings is not intended to be just an example of the usefulness of the representability of $\text{HO}\pi$ in π -calculus. Virtually all of the different recent generalisations of process calculi with the capability of treating — directly or indirectly — processes as first class objects have put forward attempts at embedding the λ -calculus. A deep comparison between a process calculus and λ -calculus is interesting for several reasons. From the process calculus point of view, besides being a remarkable test of its expressiveness, it usually contributes much towards getting a deeper insight into its theory. From the λ -calculus point of view, it provides the means to study λ -terms in more powerful contexts than the pure (sequential) λ -contexts (namely contexts where sophisticated parallel and non-deterministic operators can be expressed). This is important when considering the integration of functional and concurrent calculi: For example, we might want to know when two functional terms can be exchanged without affecting the behaviour of the process in which they are used. Moreover an encoding of the λ -calculus into a process algebra allows the study of the former using the instruments developed in the latter. For instance, an important behavioural equivalence upon process terms could give rise to a new interesting equivalence on λ -terms. Also, the importance of some λ -calculus evaluation strategy is strengthened if it can be shown to be efficiently encoded in a “successful” process calculus, with the result of intensifying the study of this strategy or even to find new appealing refinements to it.

Other motivations for describing functions as processes are to provide a semantic foundation for languages which combine concurrent and functional programming and to develop parallel implementations of functional languages.

1.5 Summary of the thesis

We summarise here the contents of the different chapters of the thesis.

We start Chapter 2 with the polyadic π -calculus: We review its syntax, its notion of sort and its operational semantics. By extending the sort discipline of π -calculus we derive the *Higher-Order π -calculus* ($\text{HO}\pi$). The syntax and

operational semantics of $\text{HO}\pi$ are natural generalisations of those of the π -calculus. In the last part of the chapter, we present some well-established theory and some conventions related to the notion of bisimulation which will be used throughout the thesis.

In Chapter 3 we seek for a way of defining uniformly bisimulation-based equivalences over different calculi. As discussed in Section 1.3, this is especially important for higher-order process calculi, in which the standard definition of bisimulation induces an over-discriminating relation. The idea is to try to achieve this by equipping a global observer with a *minimal* ability to observe actions and/or process states. We examine first the case in which there are *no* observables and only the *interaction* or *reduction* relation is present. Unfortunately this does not give enough discriminating power. Therefore we add the capability of observing some properties of the states. The predicates which are used for this give — in a process calculus — visibility of the channel at which an action occurs. We call the resulting bisimulation relation *barbed bisimulation*. By parametrising barbed bisimulation over the class of *static* contexts and over the class of all contexts, we define *barbed equivalence* and *barbed congruence*, respectively. We prove that in CCS and π -calculus, barbed equivalence and congruence coincide with the well-known bisimulation-based equivalences.

In Chapter 4 we deepen the analysis of barbed bisimulation and congruence in $\text{HO}\pi$. The target is to derive as simple as possible characterisations for them. A central rôle is played by the *triggers*, intuitively elementary agents whose only functionality is to activate a copy of another agent and provide it with the possible arguments. Using triggers, any subagent of a given agent can be factorised out: This is the content of the *factorisation theorem*. Building on this, we introduce the subclass of *triggered agents*, in which every agent emitted in an output or “expected” in an input is a trigger. Thus higher-order communications have become homogeneous and have lost all their potential richness and variety. This greatly simplifies the reasoning over agents. In particular, on triggered agents barbed equivalence coincides with *triggered bisimulation*. The advantage of the latter is that the clause for higher-order outputs just requires *identity* (modulo α -

conversion) on the labels of two matching actions and the clause for higher-order inputs only contemplates the input of a fresh trigger. We then define a mapping \mathcal{T} which transforms every agent into a triggered agent. By exploiting \mathcal{T} , we are able to prove a simple characterisation of barbed equivalence on the whole class of $\text{HO}\pi$ agents (not necessarily in triggered form), called *normal bisimulation*. This is not as simple as triggered bisimulation, but at least no universal quantifications is present in the definition.

In Chapter 5, we show how the higher-order constructs of the $\text{HO}\pi$ can be effectively compiled down to π -calculus. Formally, the compilation \mathcal{C} is derived in two steps. Firstly the mapping \mathcal{T} introduced in Chapter 4, which maps agents to triggered agents; secondly, a map from triggered agents to first-order agents (and first-order sortings). We show the operational correspondence between an $\text{HO}\pi$ agent and its encoding π -calculus agent and prove that \mathcal{C} respects barbed equivalence and congruence. We conclude the chapter by comparing our work with Bent Thomsen's, which first attempted the translation of a higher-order calculus, namely Plain CHOCS, into the π -calculus.

In Chapter 6 we carry out our investigation into the representation of functions as processes. The starting point is Milner's work on the encoding of lazy and call-by-value λ -calculus into π -calculus. The chapter is ideally divided into two parts. In the first, we present analogous encodings into $\text{HO}\pi$. The higher-order nature of $\text{HO}\pi$ makes them easier to understand and to handle than those into π -calculus; moreover the two can be compared via the compilation \mathcal{C} . On the lazy encodings, this yields a commutative diagram. For call-by-value the situation is less sharp. On the one hand because in his original work [Mil90a], Milner gives two encodings; on the other hand because \mathcal{C} does not return either. Seemingly, to obtain them some code transformation is necessary. The study of these transformations suggests a correction in Milner's encodings, which improves their faithfulness to the call-by-value discipline. The study also reveals a problem in Milner's second encoding, for which β -reduction is not valid. In the second part of the chapter we focus on the lazy λ -calculus encodings, more attractive because of their apparent canonicity. Given the correspondence proved in the first part, we can work with

our $\text{HO}\pi$ encoding and extend the results to Milner's encoding as well. We show that they give rise to a λ -model, in which a weak form of extensionality holds. The model is not fully abstract though. To obtain full abstraction we follow two directions: In the *restrictive* approach the semantic domain of processes is cut down; in the *expansive* approach λ -calculus is enriched with *constants* to obtain a *direct* characterisation of the equivalence induced by the encodings.

In Chapter 7 we comment on the results obtained and we present directions for potential future work.

Chapter 2

From π -calculus to Higher-Order

π -calculus

We start the chapter by reviewing the basic concepts of the (polyadic and sorted) π -calculus. We first present its syntax. Then its operational semantics, both in terms of a reduction relation and of a labeled transition system. The presentation is essentially the same as in [MPW92,Mil91], to which the reader is referred for more details. The major difference is that we allow infinite sums and agents with infinite free names in the syntax.

In the π -calculus only names can be communicated, and hence only “first-order” sortings are employed. However, the sorting discipline naturally lends itself to a higher-order extension, following which we have derived a calculus called the *Higher-Order π -calculus*, briefly $HO\pi$. In the $HO\pi$ not only names, but also processes and abstractions over processes of arbitrarily high order, can be exchanged. As such it is an ω -order calculus; then Thomsen’s Plain CHOCS, in which exclusively processes can be communicated, can be seen as a particular second-order case of $HO\pi$.

The $HO\pi$ is introduced in the second part of the chapter. Its syntax and operational semantics are derived with natural generalisations of those for the π -calculus. Again, we give the operational semantics both in terms of a reduction relation and of a labelled transition system. In the last part of the chapter we present some conventions regarding bisimulations and we review the major

“bisimulation up-to” techniques, used to facilitate the construction of bisimulations. We conclude with the definition of early bisimulation and congruence for the π -calculus.

NOTATION. Let us first set out some general notation for the thesis.

- We use \mathcal{R} to range over relations. We write $(P, Q) \in \mathcal{R}$ also as $P \mathcal{R} Q$.
- \mathcal{R}^{-1} denotes the inverse of \mathcal{R} , i.e. $\mathcal{R}^{-1} = \{(P, Q) : (Q, P) \in \mathcal{R}\}$.
- The binary composition of two relations \mathcal{R} and \mathcal{R}' is written as $\mathcal{R}\mathcal{R}'$. Thus $P\mathcal{R}\mathcal{R}'Q$ means that T exists s.t. $P\mathcal{R}T$ and $T\mathcal{R}'Q$.
- We use a tilde to indicate a *tuple*, that is an ordered sequence of elements. For instance, if x stands for a name, then \tilde{x} is a tuple of names. A tuple can be *empty*, but can also be *infinite*.
- We group brackets. Thus $(x_1) \dots (x_n)$ becomes (x_1, \dots, x_n) and $\langle x_1 \rangle, \dots, \langle x_n \rangle$ becomes $\langle x_1, \dots, x_n \rangle$.
- \emptyset is the empty set.
- \cup, \cap are the familiar union and intersection of sets. Sometimes, if a set H_2 contains a single element h , we abbreviate $H_1 \cup H_2$ as $H_1 \cup h$ (similarly for \cap).
- For sets H_1, H_2 , the set of elements which are in H_1 but are not in H_2 is $H_1 - H_2$.

The polyadic π -calculus

2.1 Syntax of π -calculus

2.1.1 The language

We use $a, b, c, \dots, x, y, z, \dots$ to range over the class of names (we shall use the words ‘name’, ‘port’ and ‘channel’ interchangeably) and P, Q, R, T to range over

the class of processes. The class Pr_π of the π -calculus processes is built using the operators of prefixing, sum, parallel composition, restriction, matching and constant application, with constants represented by D in the grammar below:

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

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

$$\alpha ::= x(\tilde{y}) \quad | \quad \bar{x}(\tilde{y})$$

When a constant application $D\langle \tilde{x} \rangle$ is used, D must have been defined. The defining equations for constants constitute a set and are of the form $D \stackrel{def}{=} (\tilde{x})P$, where the parameters \tilde{x} collect all names which may occur free in P . The latter requirement on \tilde{x} is useful to have simple inductive definitions of substitution and of the free names of an agent. However, for simplicity, in some cases we shall put in the parameters \tilde{x} only those names of P which are supposed to be instantiated (for instance, we might simply write $D \stackrel{def}{=} (x)\bar{a}(x).\mathbf{0}$, that is omitting a in the parameters, if we intend to maintain a in all our uses of D).

There are a few constraints in the above expressions: Firstly, in an input prefix $x(\tilde{y})$ and in a constant definition $D \stackrel{def}{=} (\tilde{x})P$, the tuples \tilde{x} and \tilde{y} are made of all distinct elements (this because they represent binders, as better precised below). Secondly, the brackets $()$ and $\langle \rangle$ are omitted when the name tuple inside is empty. Thirdly, the tuple \tilde{y} in input and output prefixes is finite. By contrast, no limitation is imposed on the tuple \tilde{x} of constant definitions and applications which, therefore, may be infinite. This allows us to have constants which use an infinite number of names, for instance a counter which at each step is able to emit a signal at a different port. Finally, we suppose that it is always possible to α -convert the names used in an expression to “fresh” names and to augment the set of defining equations — under the condition that existing equations are not overwritten.¹ A way of ensuring this is to require that the classes of names and constant symbols

¹To be formal, we could say that an agent comes equipped with a set of defining equations for the constants appearing in it. Then when compositing two agents one should make sure that there is consistency between the two corresponding sets of equations, i.e. a common constant symbol should have the same definition.

are uncountable.

An input-prefixed process $x(\tilde{y}).P$ waits for a tuple \tilde{z} to be transmitted along x ; then the continuation is the process P with the tuple \tilde{y} instantiated by the tuple \tilde{z} . An output-prefixed process $\bar{x}(\tilde{y}).P$ sends the tuple \tilde{y} along x and then behaves like P . The matching $[x = y]P$ is used to test for the equality of the names x and y . The restriction construct $\nu x P$ makes the name x local to P ; thus x becomes a new, unique name, distinct from all those external to P . Sum and parallel composition are the same operators as for CCS, the former to express non-determinism, the latter to run two processes in parallel. In the sum, I represents the countable indexing set (I may be infinite in some proofs in Chapter 3). When I is empty, we get the inactive process, written as $\mathbf{0}$. Sometimes we abbreviate $\alpha.\mathbf{0}$ as α . As usually, $+$ is taken to represent binary sum.

Following Milner [Mil91], in the sums we only admit guarded processes, i.e. processes whose outermost operator is prefixing. There are a number of reasons for preferring guarded sums. For the purpose of this thesis, perhaps the most important is that they smooth the comparison between higher-order and first-order processes that we tackle in Chapter 5. Guarded sums also simplify the reduction semantics of Section 2.2.1, and are easier to implement. Furthermore, usually in process algebras guarded sums are 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. We could have been more permissive and allow restrictions and matchings on the top of the outermost prefix of the summands of $\sum_{i \in I} \alpha_i.P_i$; this would have not affected the validity of the results about π -calculus and Higher-Order π -calculus which we shall present in the following chapters. Our choice makes the syntax of the language simpler. (The constrain is not serious for restriction, since it can just be “pulled out”; for instance, instead of $(\nu x P) + Q$, we can write $(\nu x)(P + Q)$, provided x is not free in Q .)

Constants are employed to represent processes with infinite behaviour, for in the definition $(\tilde{x})P$ of a constant D there might be calls to other constants, including D itself. The expression $(\tilde{x})P$ is like a procedure, in which \tilde{x} represents the

parameters. Of course, then in an *application* $D\langle\tilde{y}\rangle$, tuple \tilde{y} must be of the same length as \tilde{x} ; this will be assured by the use of sorts.

If $D \stackrel{def}{=} (\tilde{x})P$ and \tilde{x} is non-empty, then D and $(\tilde{x})P$ are called *abstractions*. Abstractions and processes are *agents*. We use F, G, E to range over abstractions, and A to range over agents. Milner [Mil91] uses abstractions not only in the definition of constants, but also in the construction of processes, for $a(\tilde{x}).P$ is written as $a.(\tilde{x})P$. Moreover by symmetry, $\bar{a}\langle\tilde{x}\rangle.P$ is replaced by $\bar{a}.(\tilde{x})P$, where $\langle\tilde{x}\rangle P$ is called *concretion*. Milner actually goes well beyond this, by allowing for instance the parallel composition of abstractions and concretions and by conceding (in the structural congruence relation) to push restrictions inside them. A definite advantage of his approach is its elegance. Above all, it makes explicit that the binders of abstractions and input prefixes are inherently the same. With our choice the calculus is reduced to its essential parts, and in the semantics, processes are the only kind of agents we have to deal with. This simplifies the treatment of the calculus (we have fewer structural congruence rules) and the reasoning (for example the proof of bisimulations). Moreover, our separation between action and continuation, as distinct moments in the evolution of a process, better agrees with the forms of transition systems and bisimulations we have adopted, and which will be introduced in the following sections.

The operators $a(\tilde{b}).P$, $\nu \tilde{b} P$ and $(\tilde{b})P$ bind all free occurrences of the names \tilde{b} in P . These binders give rise in the expected way to the definitions of *free* and *bound* names of an agent A , respectively $fn(A)$ and $bn(A)$. Notice in particular that if A is $D\langle\tilde{x}\rangle$, then then $fn(A) = \tilde{x}$ and $bn(A) = \emptyset$. We say that a name x *appears* in A if $x \in fn(A)$ or $x \in bn(A)$. A first-order substitution, or name substitution, is a function from names to names. We write $\{\tilde{y}/\tilde{x}\}$ (where \tilde{x} is a vector of distinct names) for the substitution which maps the x_i -th name in \tilde{x} to the y_i -th name in \tilde{y} , and maps all names not in \tilde{x} to themselves. The definitions of substitution and α -conversion on agents are standard, with renaming possibly involved to avoid capture of free names. We only present the definition of substitution. We write $\sigma(a)$ for the name onto which the substitution σ maps a ; similarly, $\sigma(\tilde{x})$ and $\sigma(fn(A))$ are the vector and the set of names obtained by applying σ to each

element of \tilde{x} and $fn(A)$.

Definition 2.1.1 (first-order substitution) *The effect of the first-order substitution σ on the agent A , written $A\sigma$, is defined inductively on A as below. A substitution does not modify bound names; to avoid that a name free in A become bound in $A\sigma$ we assume that the bound names of A have been previously α -converted to fresh names, so that $bn(A) \cap \sigma(fn(A)) = \emptyset$.*

$$\begin{aligned}
(D\langle\tilde{x}\rangle)\sigma &= D\langle\sigma(\tilde{x})\rangle \\
((\tilde{x})P)\sigma &= (\tilde{x})(P\sigma) \\
(\sum_{i \in I} P_i)\sigma &= \sum_{i \in I} (P_i\sigma) \\
(a(\tilde{x}).P)\sigma &= \sigma(a)(\tilde{x}).(P\sigma) \\
(\bar{a}\langle\tilde{x}\rangle.P)\sigma &= \overline{\sigma(a)}\langle\sigma(\tilde{x})\rangle(P\sigma) \\
(P_1 \mid P_2)\sigma &= (P_1\sigma) \mid (P_2\sigma) \\
(\nu x P)\sigma &= \nu x (P\sigma) \\
([a = b]P)\sigma &= [\sigma(a) = \sigma(b)](P\sigma) \quad \square
\end{aligned}$$

A *context* is a term with a single hole $[\cdot]$ in it. We use $C[\cdot]$ to represent a context; then $C[A]$ is the process obtained by replacing $[\cdot]$ with A .

In the following we shall be working modulo α -conversion and we write $A = A'$ if A and A' are α -convertible. We adopt the following precedence among syntactic forms, in decreasing order:

1. application,
2. substitution,
3. restriction, prefixing, matching, replicator (see below),
4. parallel composition,
5. sum,
6. abstraction.

For instance, $\alpha.D\langle\tilde{x}\rangle|\nu y P$ means $(\alpha.(D\langle\tilde{x}\rangle))|\nu y P$. Occasionally, we use $\prod_{i=1}^n P_i$ as abbreviation for $P_1 | \dots | P_n$. With reference to the previous scale, symbol \prod has precedence (3); thus $\prod_{i=1}^n P_i | Q$ means $(\prod_{i=1}^n P_i) | Q$.

The *monadic* π -calculus is obtained from the polyadic one when only tuples of length one are allowed. We can see a superiority of the polyadic over the monadic. First, when sorts are employed the former is more expressive than the latter [Mil90b] (at least in the present formulation of sorts, as in Section 2.1.2). Secondly, even if without sorts in the monadic calculus a translation from the polyadic to the monadic is given in [Mil91, Section 3.1], it remains unclear whether this truly respects the equivalences in the two versions. Thirdly, the use of tuples arises naturally in many applications. Finally, it seems that the theory for the monadic is always generalisable straightaway to the polyadic.

Other process expressions

There are two other forms of process expressions, semantically derivable in our syntax, which we shall sometimes use. The *replication* $!P$, whose syntax deliberately recalls the “of course” connective of linear logic [Gir87], intuitively represents $P | P \dots$, i.e. an unbounded number of copies of P in parallel. Replication sometimes is included in the π -calculus in place of constants [Mil91]. In fact, it is easy to code it up using constants. And if the definition $(\tilde{x})P$ of each constant has the parameter (\tilde{x}) finite and uses a finite number of constants (i.e. there is no infinite chain D_1, \dots, D_n, \dots s.t. P invokes D_1 and the definition of each D_i invokes a D_j with $j > i$), then also replication can encode constants (see [Mil91, Section 3]). We preferred to take recursive definition as primitive, because we need constants defined in terms of an infinity of constants in some proofs on barbed bisimulation. Moreover, constants seem more fundamental, for their simulation using replication may not be possible in a true concurrency setting [Sanb].

The *silent* prefixing $\tau.P$ represents a process which can evolve to P without requiring communication with the environment. It can be written as $\nu a (\bar{a} | a.P)$, where a is not free in P . We anticipate here the rules describing the formal

behaviour of these operators, which accompany those for the other operators which will be given in Table 2–1:

$$\text{REP: } \frac{P \mid !P \xrightarrow{\mu} P'}{!P \xrightarrow{\mu} P'} \qquad \text{TAU: } \tau.P \xrightarrow{\tau} P$$

2.1.2 Sorting

All realistic systems which have been described with the π -calculus seem to obey some discipline in the use of names. As Milner says in [Mil91], it is much as for the λ -calculus, which is hardly ever used freely, i.e. without an implicit or explicit type discipline. The introduction of sorts and sortings into the π -calculus 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. Let us briefly review now the definition of sorts and sortings.

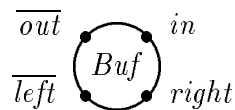
Names are partitioned into a collection of *subject sorts* — possibly infinite many of them — with the condition that that each term of the language leaves unused an infinite number of names for each sort, to allow α -conversion. We write $x : s$ to mean that x belongs to the subject sort s . We also write $x : y$ to mean that names x and y have the same sort. These notations are extended to tuples componentwise. Then *object sorts* are (possibly infinite) sequences over subject sorts, such as (s_1, \dots, s_n) or (s) ; $()$ denotes the empty object sort. We use S to range over object sorts. Finally, a *sorting* is a function Ob mapping each subject sort to a finite object sort. Typically, we write $s \mapsto (\tilde{s}) \in Ob$, (or simply $s \mapsto (\tilde{s})$ if Ob is clear), to mean that Ob assigns the object sort (\tilde{s}) to s . We also use the compact notation $a : s' \mapsto (\tilde{s})$ to mean that $a : s'$ and $s' \mapsto (\tilde{s})$. Intuitively a sorting just describes the sort of the name-vectors which a subject sort can carry. For instance, 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 . Thus CCS and the monadic unsorted π -calculus can be derived by imposing the sortings $\{\text{NAME} \mapsto ()\}$ and $\{\text{NAME} \mapsto (\text{NAME})\}$ respectively, in which all names belong to the same subject

sort NAME.

We say that an agent A *respects* a sorting Ob (or is *well-sorted* for Ob) if all prefixes, matchings and applications in A obey the discipline given by Ob , in the following sense. A prefix $a(\tilde{x}).P$ with $a : s' \mapsto (\tilde{s})$ respects Ob if $\tilde{x} : \tilde{s}$. Similarly, a matching $[x = y]P$ respects Ob if $x : y$. Finally, to say when an application respects Ob , we first assign an object sort to agents: Processes take the sort $()$, whereas if $D \stackrel{def}{=} (\tilde{x})P$ and $\tilde{x} : \tilde{s}$, then D , and $(\tilde{x})P$ take the sort (\tilde{s}) . Now, the requirement is that in an application $D\langle\tilde{y}\rangle$, if $\tilde{y} : \tilde{s}$ then it must be that $D : (\tilde{s})$. Throughout the thesis we suppose that all agents considered respect some sorting Ob .

Similarly to the notation for names, we write $A : A'$ to mean that A and A' are agents with the same sort.

Example 2.1.2 (from [Mil90b]: A cut-out buffer) We want to represent a chain of identical buffers in which any empty buffer can cut itself out by providing its left neighbour with access to its right neighbour. For simplicity we require that in this case the left neighbour must be empty too. Pictorially, a buffer is represented as follows:



Buf receives a single value at in and retransmits it at out ; ports $right$ and $left$ are used for the cut-out operations.

$Buf \stackrel{def}{=}$

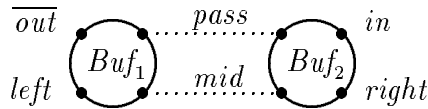
$$(in, out, left, right) \left(in(x).\overline{out}\langle x \rangle.Buf\langle in, out, left, right \rangle + \overline{left}\langle in, right \rangle + right(newin, newright).Buf\langle newin, out, left, newright \rangle \right)$$

The abstraction on the communication ports allows an elegant description of the chaining of two buffers Buf_1 and Buf_2 :

$Buf_1 \frown Buf_2 \stackrel{def}{=}$

$$(in, out, left, right)(\nu pass, mid) \left(Buf_1\langle pass, out, left, mid \rangle \mid Buf_2\langle in, pass, mid, right \rangle \right)$$

which can be visualised as



where the dotted lines indicate a restriction on the corresponding ports. The cut-out of Buf_2 is obtained via an interaction at mid and yields a process which is structurally congruent (as defined in the next section) to $Buf_1\langle in, out, left, right \rangle$.

In this example, the names involved are given quite different tasks. This is reflected in the sorting Ob which can be assigned. Denoting with S the object sort of the name x (we do not have information for S), we have:

$$Ob = \begin{cases} x : s_1 & \mapsto S \\ in, out, pass : s_2 & \mapsto (s_1) \\ left, right, mid : s_3 & \mapsto (s_2, s_3) \end{cases}$$

The sort of the agent identifier Buf is (s_2, s_2, s_3, s_3) . □

2.2 Operational Semantics of π -calculus

Traditionally, the operational semantics of a process algebra is given in terms of a *labelled transition system* describing the possible evolutions of a process. This contrasts with what happens in *term rewriting systems*, as based on an *unlabelled reduction system*. In the λ -calculus, probably the best known term-rewriting system, what makes a reduction system possible is that two terms having to interact are naturally in contiguous positions. This is not the case in process calculi, where interaction is not dependent upon physical contiguity. To put this another way, a *redex* of a λ -term is a subterm, while a “redex” in a process calculus is distributed over the term.

A guideline for the definition of reduction systems in process algebras is offered by Milner [Mil90a, Mil91], inspired by Berry and Boudol’s Chemical Abstract Machine [BB92]. In this technique, axioms for a structural congruence relation are

introduced prior to the reduction system, in order to break a rigid, geometrical vision of concurrency; then reduction rules can easily be presented in which redexes are indeed subterms again.

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. This is not quite the case in the labelled semantics, at least for process algebras expressing mobility: The manipulation of names and the side conditions in the rules are non-trivial and this makes delicate the justification of the choices made. However, if the reduction system is available, the correctness of the labelled transition system can be shown by proving the correspondence between the two systems.

On the other hand, the advantages of labelled semantics appear later, when reasoning with agents. In reduction semantics, the behaviour of a process is understood relatively to a context in which it is contained and with which it interacts. Instead, with labelled semantics every possible communication of a process can be determined in a direct way. This allows us to get simple characterisations of behavioural equivalences. Moreover with labelled semantics the proofs benefit from the possibility of reasoning in a purely structural way.

The respective advantages of the two semantics become even more noticeable in a higher-order calculus, like $\text{HO}\pi$. The conclusion is that both semantics are useful and that they integrate and support each other.

2.2.1 Reduction semantics

Structural congruence, written \equiv , is the smallest congruence over the class of π -calculus processes which satisfies the following rules:

1. $P \equiv Q$ if P is α -convertible to Q ;
2. $\sum_{i \in I} \alpha_i P_i \equiv \sum_{j \in J} \alpha_j P_j$ if J is a permutation of I ;
3. abelian monoid laws for $|$: $P | Q \equiv Q | P$; $P | (Q | R) \equiv (P | Q) | R$; $P | \mathbf{0} \equiv P$;
4. $\nu x \mathbf{0} \equiv \mathbf{0}$; $\nu x \nu y P \equiv \nu y \nu x P$; $(\nu x P) | Q \equiv \nu x (P | Q)$; if $x \notin \text{fn}(Q)$;

5. $[x = x]P \equiv P$;
6. if $D \stackrel{def}{=} (\tilde{x})P$, and $\tilde{x} : \tilde{y}$, then $D(\tilde{y}) \equiv P\{\tilde{y}/\tilde{x}\}$.

We do not have a precise definition of what should determine the set of rules for \equiv . We can however try to indicate some criteria (or general principles) for this:

1. The rules should be valid in any reasonable behavioural equivalence;
2. they should provide some intuition for the operators;
3. they should allow a simple statement of the reduction rules and should be needed when proving the correspondence between reduction and labelled semantics;
4. there should be only a few of them.

On some specific rules however, these principles might not provide a univocal guidance. For instance the rule $\nu x \mathbf{0} \equiv \mathbf{0}$ is not necessary to state the reduction rules and prove correspondence between reduction and labelled semantics and hence might be canceled; yet we have preferred to keep it because it seems to be strongly suggested by criteria (1) and (2) (a similar dispute arises when considering whether to add the rule $\sum_{i \in I} P \equiv P$). The system that we have given follows Milner's in [Mil91] and we think it achieves a good compromise among the above listed principles.

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

$$\begin{array}{c}
 \text{COM: } (\cdots + x(\tilde{y}).P) \mid (\cdots + \bar{x}(\tilde{z}).Q) \longrightarrow P\{\tilde{z}/\tilde{y}\} \mid Q \\
 \text{PAR: } \frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q} \qquad \text{RES: } \frac{P \longrightarrow P'}{\nu x P \longrightarrow \nu x P'} \\
 \text{STRUCT: } \frac{Q \equiv P \quad P \longrightarrow P' \quad P' \equiv Q'}{Q \longrightarrow Q'}
 \end{array}$$

Example 2.2.1 Suppose $x \notin fn(P, Q)$. Then

$$\begin{aligned}
\nu x (\bar{a}\langle x \rangle.P) | R | a(y).Q &\equiv \\
\nu x (\bar{a}\langle x \rangle.P | a(y).Q) | R &\longrightarrow \\
\nu x (P | Q\{x/y\}) | R &\equiv \\
P | R | \nu x (Q\{x/y\}) &\quad \square
\end{aligned}$$

Note that reduction is not allowed underneath a prefix. Thus $\bar{a}\langle x \rangle.P | a(y).Q \longrightarrow P | Q\{x/y\}$ but $\alpha.(\bar{a}\langle x \rangle.P | a(y).Q) \not\longrightarrow \alpha.(P | Q\{x/y\})$. Prefixing is the construct introduced to represent sequentialisation. Hence nothing underneath a prefix should occur before this has fired.

2.2.2 Labelled transition semantics

In the labelled semantics we present, the bound names of an input are instantiated as soon as possible, namely in the rule for input. It is therefore an *early* transition system, as opposed to a *late* transition system – for instance the one used in [MPW92] – where such an instantiation is done later, in the rule for communication. We chose the former because the bisimulation it supports, called *early bisimulation*, has some attractive peculiarities, discussed in Section 2.6, and it is the bisimulation which we shall utilise throughout the thesis.

There are three possible forms for an action. Besides the silent-action τ representing interaction, we have:

$$\begin{aligned}
P \xrightarrow{x\langle \tilde{y} \rangle} P' &\quad \text{input action; } x \text{ is the name at which it occurs,} \\
&\quad \tilde{y} \text{ is the tuple of names which are received} \\
P \xrightarrow{(\nu \tilde{y}')\bar{x}\langle \tilde{y} \rangle} P' &\quad \text{output action; it is the output of the names } \tilde{y} \text{ at } x. \\
&\quad \text{It always holds that } \tilde{y}' \subseteq \tilde{y} - x; \text{ in fact } \tilde{y}' \text{ represents} \\
&\quad \text{private names which are emitted from } P, \text{ i.e. carried} \\
&\quad \text{out from their current scope (} \textit{scope extrusion} \text{)}
\end{aligned}$$

In both cases, x is the *subject* and \tilde{y} is the *object* part of the action. In the input the subject is *positive*, whereas in the output it is *negative*. Note the different brackets in input prefixes and input actions (round in the former, angled in the latter). This is to recall that in input prefix $x\langle \tilde{y} \rangle$, the names \tilde{y} are *binders* (waiting to be instantiated) whereas in an input action $x\langle \tilde{y} \rangle$ they represent *values* (with

which the binders have been instantiated) — in the same way as the names \tilde{y} are values in an output $\bar{x}\langle\tilde{y}\rangle$.

We use μ to represent the label of a generic action (not to be confused with α , which ranges over prefixes). Given an action μ , the bound and free names of μ , respectively written $bn(\mu)$ and $fn(\mu)$, are defined as follows:

μ	$fn(\mu)$	$bn(\mu)$
$x\langle\tilde{y}\rangle$	x, \tilde{y}	\emptyset
$(\nu \tilde{y}')\bar{x}\langle\tilde{y}\rangle$	$x, \tilde{y} - \tilde{y}'$	\tilde{y}'
τ	\emptyset	\emptyset

The names of μ , briefly $n(\mu)$, are $bn(\mu) \cup fn(\mu)$.

The transition system is presented in Table 2–1. We have omitted the symmetric versions of rules SUM, PAR, and COM. Note the presence of the rule ALP: This allows us to define the other transitions up-to redenomination of the bound names; it also allows us to avoid some tedious side conditions when using the transition system. The treatment of restriction is delicate and deserves some explanation.

- In the rule RES the side condition prevents transitions like

$$(\nu b)a(x).P \xrightarrow{a\langle b \rangle} \nu b P\{b/x\}$$

which would violate the static binding assumed for restriction. This does not mean however that name b cannot be received. To allow this, we have just to use some α -conversion:

$$(\nu b)a(x).P = (\nu c)a(x).P\{c/b\} \xrightarrow{a\langle b \rangle} \nu c P\{b/x\}$$

- It would not be enough for the side condition in the RES rule to say $x \notin fn(\mu)$; for, if so, then one could derive the transition

$$\nu y \left((\nu y)\bar{x}\langle y \rangle.P \right) \xrightarrow{(\nu y)\bar{x}\langle y \rangle} \nu y P$$

in which free occurrences of y in P are captured by the wrong binder.

- The OPEN rule is the one which implements scope extrusion.

$$\begin{array}{l}
\text{ALP: } \frac{P' \xrightarrow{\mu} Q \quad P \text{ and } P' \text{ are } \alpha\text{-convertible}}{P \xrightarrow{\mu} Q} \\
\text{OUT: } \overline{x}(\tilde{y}).P \xrightarrow{\overline{x}(\tilde{y})} P \qquad \text{INP: } x(\tilde{y}).P \xrightarrow{x(\tilde{z})} P\{\tilde{z}/\tilde{y}\}, \text{ if } \tilde{z} : \tilde{y} \\
\text{SUM: } \frac{P \xrightarrow{\mu} P'}{P + Q \xrightarrow{\mu} P'} \qquad \text{PAR: } \frac{P \xrightarrow{\mu} P'}{P \mid Q \xrightarrow{\mu} P' \mid Q} \text{ } bn(\mu) \cap fn(Q) = \emptyset \\
\text{COM: } \frac{P \xrightarrow{(\nu \tilde{y}')\overline{x}(\tilde{y})} P' \quad Q \xrightarrow{x(\tilde{y})} Q'}{P \mid Q \xrightarrow{\tau} \nu \tilde{y}'(P' \mid Q')} \tilde{y}' \cap fn(Q) = \emptyset \\
\text{MATCH: } \frac{P \xrightarrow{\mu} P'}{[x = x]P \xrightarrow{\mu} P'} \qquad \text{CONST: } \frac{P\{\tilde{y}/\tilde{x}\} \xrightarrow{\mu} P'}{D(\tilde{y}) \xrightarrow{\mu} P'} \text{ if } D \stackrel{def}{=} (\tilde{x})P \\
\text{RES: } \frac{P \xrightarrow{\mu} P'}{\nu x P \xrightarrow{\mu} \nu x P'} \quad x \notin n(\mu) \quad \text{OPEN: } \frac{P \xrightarrow{(\nu \tilde{y}')\overline{x}(\tilde{y})} P'}{\nu x P \xrightarrow{(\nu x, \tilde{y}')\overline{x}(\tilde{y})} P'} \quad x \neq z, x \in \tilde{y} - \tilde{y}'
\end{array}$$

Table 2–1: π -calculus's labelled transition system

- The side condition in the rule COM prevents interactions like

$$\nu y (\overline{x}(y).P) \mid (a(x).Q \mid \overline{y}.0) \xrightarrow{\tau} \nu y (P \mid (Q\{y/x\} \mid \overline{y}.0))$$

in which the free occurrence of y in $\overline{y}.0$ has become bound after the interaction.

2.2.3 Correspondence between the two semantics

It can be shown that modulo structural congruence, the relation \longrightarrow of reduction semantics is exactly the relation $\xrightarrow{\tau}$ of labelled semantics. Moreover, it is also possible to establish what the observable actions of the latter corresponds to in the former. Roughly speaking, they are represented by those prefixes which can be pulled out (nearly) to the top by means of \equiv . For instance, if $P \xrightarrow{\overline{x}(\tilde{y})} P'$ then one can show that Q, T, R exist s.t. $P \equiv \nu \tilde{z} ((\overline{x}(\tilde{y}).Q + T) \mid R)$, with $\tilde{z} \cap \{x, \tilde{y}\} = \emptyset$

and $P' \equiv \nu \tilde{z}(Q \mid R)$, and with the subterm $+T$ possibly missing. Precise results by Milner and Turner on the topic are in [Mil90a,Tur91], for the monadic case.

Because of this agreement, in the following chapters — where we always work with the labelled transition system — we shall often abbreviate $\xrightarrow{\tau}$ as \longrightarrow .

Higher Order π -calculus

2.3 Syntax of HO π

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* 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 “executor”, which receives a process at x and executes it, can be written as $x(X).X$; when put in parallel with $\bar{x}\langle P \rangle.Q$, it gives rise to the interaction

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

Let us see more interesting examples, involving also higher-order abstractions.

Example 2.3.1 In [Mil91], Milner shows how to encode numbers in the π -calculus.

If \bar{y}^n is taken to mean $\underbrace{\bar{y} \cdot \dots \cdot \bar{y}}_{n \text{ times}}$, and $y, z : s \mapsto ()$, then the natural number n is encoded as follows:

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

We want now to write an agent *Plus* capable of performing the sum of two numbers. Consider the process $\nu x([n]\langle y, x \rangle \mid x.[m]\langle y, z \rangle)$: If we abstract from possible τ -actions, 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

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

Plus is a higher-order abstraction (it cannot be written in π -calculus), because it abstracts from X and Y which are agent-variables; 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.

Now an adder which recursively takes two integers at ports a_1, a_2 and outputs their sum at a_3 can be represented as:

$$Add \stackrel{def}{=} a_1(X).a_2(Y).\bar{a}_3\langle Plus \langle X, Y \rangle \rangle. Add \quad \square$$

Example 2.3.2 In Example 2.1.2, a chaining operator \frown connects two buffers with sort (\tilde{s}) , for $\tilde{s} = s_2, s_2, s_3, s_3$. In $HO\pi$, if $X, Y : (\tilde{s})$, then we can set

$$\frown \stackrel{def}{=} (X, Y)(in, out, left, right)(\nu pass, mid)(X\langle pass, out, left, mid \rangle | Y\langle in, pass, mid, right \rangle)$$

which thus is an agent of sort $((\tilde{s}), (\tilde{s}), \tilde{s})$. \square

At this point, the formal modifications of π -calculus syntax and semantics required to obtain those for $HO\pi$ should start becoming clear. We consider this below. The notation already introduced for the π -calculus will not be repeated.

The language

Let *Var* be a set of agent-variables, ranged over by X, Y . To simplify the notation, we use K to stand for an agent or a name and U to stand for a variable or a name. There are only two modifications to bring into the syntax of the π -calculus. First, tuples over K and U must replace pure name tuples in prefixing, abstractions and applications. Secondly, variable application should be allowed too, so that an abstraction received as input can be provided with the appropriate arguments. In this thesis, we shall only be interested in well-sorted expressions. We shall define well-sorted expressions inductively, using a set of formation rules, after presenting the sorting discipline. For ease of reference, we first give the grammar for (unsorted) processes and agents. The grammar for processes is:

$$\begin{aligned} P &:: \sum_{i \in I} \alpha_i.P_i \quad | \quad P_1 | P_2 \quad | \quad \nu x P \quad | \quad [x = y]P \quad | \quad D\langle \tilde{K} \rangle \quad | \quad X\langle \tilde{K} \rangle \\ \alpha &:: \bar{x}\langle \tilde{K} \rangle \quad | \quad x(\tilde{U}) \end{aligned}$$

where defining equations for constants are of the form $D \stackrel{def}{=} (\tilde{U})P$. Remember that K may be an agent; hence it may be a process, but also an abstraction of arbitrary high order. The grammar for agents is:

$$A ::= (\tilde{U})P \quad | \quad (\tilde{U})X\langle\tilde{K}\rangle \quad | \quad (\tilde{U})D\langle\tilde{K}\rangle$$

(Note also that a variable X and a constant D are agent, corresponding to the cases in which \tilde{U} and \tilde{K} are empty).¹ The same assumptions as for π -calculus apply: Thus, the tuple of binders \tilde{U} is made of distinct elements; and the brackets $()$ and $\langle\rangle$ are intended to be omitted when the tuple inside is empty. All tuples are required to be finite, except those being the parameters of the constants, which may contain an infinite number of names so to express agents which use an infinity of different names (but the number of agent-variables and of agents in these tuples remain finite). A variable X which is not underneath some input prefix $x(\tilde{U})$ or an abstraction (\tilde{U}) with $X \in \tilde{U}$, is *free*. A agent possibly containing free variables is *open*. We denote by $fv(A)$ the set of free variables of the agent A . We use $HO\pi^\circ$ to denote the class of open agents and $HO\pi$ for the class of *closed* agents. Note that every subterm of a closed agent has a finite number of free variables. Indeed, an agent obtained from the given grammar may have infinite free variables, due to the use of infinite sums; but then it could not be made closed because the input and abstractions binders can only bind a finite number of variables.

Well-sorted expressions

In the $HO\pi$ the need for sorts is even more compelling than in π -calculus: Besides the arity question, we also have 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 case of Section 2.1.2, the difference in the syntax for sorts is that the sequences representing object sorts do not have to be made only of subject sorts; rather, object sorts themselves can appear too. We call subject

¹There is a difference in the use of applications, $X\langle\tilde{K}\rangle$ and $D\langle\tilde{K}\rangle$, in the grammar for processes and for agents. In the former case, \tilde{K} is supposed to represent the tuple of *all* arguments for X and D ; in the latter case, \tilde{K} represents an *initial* segment of these arguments. This difference should become clear from the formation rules presented later.

$\frac{X \in \text{Var}(S)}{X : S}$	$\frac{\tilde{U} : \tilde{El} \quad A : ()}{D : (\tilde{El})} \text{ if } D \stackrel{\text{def}}{=} (\tilde{U})A$
$\frac{\tilde{U} : \tilde{El}' \quad A : (\tilde{El})}{(\tilde{U})A : (\tilde{El}', \tilde{El})}$	$\frac{\tilde{K} : \tilde{El} \quad A : () \quad a : s \mapsto (\tilde{El})}{\bar{a}(\tilde{K}).A : ()}$
$\frac{\tilde{U} : \tilde{El} \quad A : () \quad a : s \mapsto (\tilde{El})}{a(\tilde{U}).A : ()}$	$\frac{\forall i \ A_i : ()}{\sum_{i \in I} A_i : ()}$
$\frac{A_1 : () \quad A_2 : ()}{A_1 \mid A_2 : ()}$	$\frac{A : ()}{\nu x \ A : ()}$
$\frac{x : y \quad A : ()}{[x = y]A : ()}$	$\frac{X : (\tilde{El}', \tilde{El}) \quad \tilde{K} : \tilde{El}'}{X(\tilde{K}) : (\tilde{El})}$
$\frac{D : (\tilde{El}', \tilde{El}) \quad \tilde{K} : \tilde{El}'}{D(\tilde{K}) : (\tilde{El})}$	

Table 2–2: Formation rules for HO π

sorts and object sorts *element sorts*; we use El to range over element sorts.

$$\begin{aligned} El &:: s \mid S \\ S &:: (\tilde{El}) \end{aligned}$$

For each object sort S we suppose the existence of an infinite number of variables of sort S , denoted by $\text{Var}(S)$. Notice that the special case of second-order sorting $\{\text{NAME} \mapsto (())\}$ corresponds to Thomsen’s Plain CHOCS.

The set of formation rules for well-sorted agents is presented in Table 2–2 (we remind that $a : s \mapsto (\tilde{El})$ means that the name a belongs to the subject sort s and that the object sort of s is (\tilde{El})). An agent A is *well-sorted* for Ob , if we can infer $A : S$, for some S , from the rules in the table. Moreover, if $S = (\tilde{El})$, for \tilde{El} non-empty, then A is an *abstraction*, whereas if \tilde{El} is empty, then A is a *process*. The well-sortness condition restricts the grammar for agents to those expressions in which all prefixings, matchings and applications conform to the given sorting discipline, in the sense that we explained for π -calculus — the difference being that here we have higher-order sorts. As an aside, let us mention an alternative notation for sorts which seems fairly effective in HO π . Consider the abstraction

$G \stackrel{def}{=} (X)F$, for $X : S'$, $F : S$. It really 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 $G : S' \longrightarrow S$. Thus a sort (El', \widetilde{El}) is equivalent to $El' \longrightarrow (\widetilde{El})$. For instance, for the agent *Plus* in Example 2.3.1, we would have, for $S = s \longrightarrow s \longrightarrow ()$:

$$Plus : S \longrightarrow S \longrightarrow s \longrightarrow s \longrightarrow ()$$

or also, $Plus : S \times S \longrightarrow s \times s \longrightarrow ()$, using some “uncurrying”. In the sequel however, we stick to the original notation.

More on the syntax for well-sorted expressions

Well-sortness allows us to have, in the syntax, only applications in which the term on the left is a constant or a variable: Every expression $A\langle\widetilde{K}\rangle$, if well-sorted, can be rewritten as an expression in our syntax by “executing” the applications it contains; for instance from $((X)Y\langle X\rangle)\langle P\rangle$, we get $Y\langle P\rangle$. This will be made clearer in Lemma 2.3.4, showing the well-definiteness of agent substitutions. The restriction to only constant and variable applications makes the definition of substitution more elaborated but facilitates the proofs in the calculus. However in the thesis we shall often use $A\langle\widetilde{K}\rangle$ as metanotation: For instance, if $F \stackrel{def}{=} (U)P$, then $F\langle K\rangle$ abbreviates $P\{F/U\}$.

We only give the definition of higher-order substitutions, i.e. substitutions among agents: A substitution can always be decomposed into a first-order substitution, i.e. a substitution among names, and into a higher-order substitution; and first-order substitution is defined similarly as for π -calculus. A higher-order substitution $B\{\widetilde{A}/\widetilde{X}\}$ represents the simultaneous replacement in B of the X_i 's with the A_i 's. Since every subterm of a closed HO π agent has a finite number of free variables, it is enough to consider substitutions in which the tuples \widetilde{A} and \widetilde{X} are finite. Such substitutions can be performed sequentially, i.e. replacing one argument at a time, if previously some α -conversion is used to guarantee that the X_i 's do not appear in \widetilde{A} . Therefore we only give the definition of unary sub-

stitutions. In Definition 2.3.3 below we use the possibility of η -converting the substituting agent A into the form $(\tilde{U})P$, that is, an abstraction on a process. The transformation \triangleright_η which does so has the expected definition:

- If $A = (\tilde{U})P$ then $A \triangleright_\eta A$.
- If $A = (\tilde{U}')X\langle\tilde{K}\rangle$ with $X\langle\tilde{K}\rangle : (\tilde{E}l)$ and $\tilde{U} : \tilde{E}l$ and no U_i appears in \tilde{K} , then $A \triangleright_\eta (\tilde{U}', \tilde{U})X\langle\tilde{K}, \tilde{U}\rangle$.
- Similarly, if $A = (\tilde{U}')D\langle\tilde{K}\rangle$ with $D\langle\tilde{K}\rangle : (\tilde{E}l)$ and $\tilde{U} : \tilde{E}l$ and no U_i appears in \tilde{K} , then $A \triangleright_\eta (\tilde{U}', \tilde{U})D\langle\tilde{K}, \tilde{U}\rangle$.

We write $\tilde{K}\sigma$ for the tuple obtained from \tilde{K} by replacing its i -th element K_i with $K_i\sigma$; obviously, $K_i\sigma = K_i$ if K_i is a name and σ a higher-order substitution.

Definition 2.3.3 (higher-order substitution) *Let B and A be well-sorted agents and $\sigma = \{A/X\}$, for $A : X$. Then the effect of the substitution σ on the agent B , written $B\sigma$, is defined inductively on B as below. We suppose that the names and variables which are bound in B do not appear in A and are different from X .*

$$\begin{aligned}
(Y\langle\tilde{K}\rangle)\sigma &= \begin{cases} P\{\tilde{K}\sigma/\tilde{U}\} & \text{if } Y = X \text{ and } A \triangleright_\eta (\tilde{U})P \\ Y\langle\tilde{K}\sigma\rangle & \text{otherwise} \end{cases} \\
(D\langle\tilde{K}\rangle)\sigma &= D\langle\tilde{K}\sigma\rangle \\
((\tilde{U})B')\sigma &= (\tilde{U})(B'\sigma) \\
(\sum_i P_i)\sigma &= \sum_i (P_i\sigma) \\
(a(\tilde{U}).P)\sigma &= a(\tilde{U}).(P\sigma) \\
(\bar{a}\langle\tilde{K}\rangle.P)\sigma &= \bar{a}\langle\tilde{K}\sigma\rangle.(P\sigma) \\
(P_1 \mid P_2)\sigma &= (P_1\sigma) \mid (P_2\sigma) \\
(\nu x P)\sigma &= \nu x (P\sigma) \\
([a = b]P)\sigma &= [a = b](P\sigma) \quad \square
\end{aligned}$$

Lemma 2.3.4 *Higher-order substitution, as given in Definition 2.3.3, is well-defined and terminating.*

PROOF: We proceed by a nested induction. The external induction is on the depth of the sort (\widetilde{El}) of X and A , where the depth of a sort is the maximum level of bracket nesting in the definition of the sort and intuitively, it says how “high-order” the sort is (the depth is finite because the agents of our language may only abstract on a finite number of variables). The internal induction is on the structure of the agent B to which the substitution is applied. The delicate clause is the one for application $Y\langle\widetilde{K}\rangle$, when $Y = X$. We have to ensure that \widetilde{U} and \widetilde{K} have the same sort and that the recursive call of substitutions does not degenerate into an infinite loop. The former is immediate: By hypothesis, X and A have the same sort (\widetilde{El}) ; therefore by definition of well-sorted agents, we have $\widetilde{U} : \widetilde{El}$ and $\widetilde{K} : \widetilde{El}$. For the latter, we distinguish the case in which the depth of A is one or greater than one. If the depth is one, then \widetilde{K} and \widetilde{U} are all names and therefore $P\{\widetilde{K}/\widetilde{U}\}$ terminates in one step. Suppose that the depth of (\widetilde{El}) is $n + 1$. Then each K_i is either a name or a subagent of B whose sort El_i has depth less or equal to n . Therefore, using induction on the structure of B , $\widetilde{K}\sigma$ is terminating, and then, using induction on the depth of the sort, $P\{\widetilde{K}\sigma/\widetilde{U}\}$ is terminating. \square

2.4 Operational semantics of HO π

Reduction semantics

The structural congruence rules and the reduction rules for HO π are the same as for π -calculus. We only have to generalise the structural rule (6) and the rule COM, so that the tuples involved may contain agents; these become:

6. If $D \stackrel{def}{=} (\widetilde{U})P$ and $\widetilde{U} : \widetilde{K}$, then $D\langle\widetilde{K}\rangle \equiv P\{\widetilde{K}/\widetilde{U}\}$.

and

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

$$\begin{array}{l}
\text{ALP: } \frac{P' \xrightarrow{\mu} Q \quad P \text{ and } P' \text{ are } \alpha\text{-convertible}}{P \xrightarrow{\mu} Q} \\
\text{OUT: } \overline{x}(\widetilde{K}).P \xrightarrow{\overline{x}(\widetilde{K})} P \qquad \text{INP: } x(\widetilde{U}).P \xrightarrow{x(\widetilde{K})} P\{\widetilde{K}/\widetilde{U}\}, \text{ if } \widetilde{K} : \widetilde{U} \\
\text{SUM: } \frac{P \xrightarrow{\mu} P'}{P + Q \xrightarrow{\mu} P'} \qquad \text{PAR: } \frac{P \xrightarrow{\mu} P'}{P \mid Q \xrightarrow{\mu} P' \mid Q} \text{ } bn(\mu) \cap fn(Q) = \emptyset \\
\text{COM: } \frac{P \xrightarrow{(\nu \widetilde{y})\overline{x}(\widetilde{K})} P'}{P \mid Q \xrightarrow{\tau} \nu \widetilde{y}(P' \mid Q')} \widetilde{y} \cap fn(Q) = \emptyset \\
\text{MATCH: } \frac{P \xrightarrow{\mu} P'}{[x = x]P \xrightarrow{\mu} P'} \qquad \text{CONST: } \frac{P\{\widetilde{K}/\widetilde{U}\} \xrightarrow{\mu} P'}{D(\widetilde{K}) \xrightarrow{\mu} P'}, \text{ if } D \stackrel{\text{def}}{=} (\widetilde{U})P \\
\text{RES: } \frac{P \xrightarrow{\mu} P'}{\nu x P \xrightarrow{\mu} \nu x P'} \quad x \notin n(\mu) \quad \text{OPEN: } \frac{P \xrightarrow{(\nu \widetilde{y})\overline{x}(\widetilde{K})} P'}{\nu x P \xrightarrow{(\nu x, \widetilde{y})\overline{x}(\widetilde{K})} P'} \quad x \neq z, x \in fn(\widetilde{K}) - \widetilde{y}
\end{array}$$

Table 2–3: HO π 's labelled transition system

Labelled semantics

The generalisation of the labelled transition system requires a little more thought than for the reduction system. The system is represented in Table 2–3. Visible actions become of the form $x(\widetilde{K})$ (input action) or $(\nu \widetilde{y})\overline{x}(\widetilde{K})$ (output action). In the latter, it holds that $\widetilde{y} \subseteq fn(\widetilde{K}) - x$; for instance, if $y \in fn(P)$, we have

$$(\nu y)\overline{x}(P).Q \xrightarrow{(\nu y)\overline{x}(P)} Q$$

Here the free occurrences of y in P force a name extrusion, to respect the static binding on restriction.

The same correspondence between the two semantics mentioned for π -calculus holds for HO π too.

CONVENTION. When performing algebraic manipulations, sometimes we will omit obvious side conditions. For instance, we might write $\nu a P \mid Q \equiv \nu a(P \mid Q)$

without recalling that $a \notin fn(Q)$. \square

2.5 Some preliminaries about bisimulations

This section is devoted to presenting some well-established theory and some conventions related to the notion of bisimulation, which will be used throughout the thesis.

2.5.1 Strong and weak bisimulations

For all bisimulations we consider there is a *strong* and a *weak* version. In the strong case all actions are treated uniformly. In the weak case, one abstracts away from silent actions, because they represent internal behaviour of processes. For this, the *weak transitions* have to be introduced: First the relation \Longrightarrow , the reflexive and transitive closure of $\xrightarrow{\tau}$; then $\xRightarrow{\mu}$ as $\Longrightarrow \xrightarrow{\mu} \Longrightarrow$; finally $\xRightarrow{\hat{\mu}}$ as $\xRightarrow{\mu}$ if $\mu \neq \tau$, and as \Longrightarrow if $\mu = \tau$.

In the thesis, we only define the strong version of a bisimulation; the weak one can be obtained from the former in a completely standard way, which we are going to describe. Let $\langle \rangle$ be the symbol chosen for the strong bisimulation. The definition of $\langle \rangle$ -bisimulation will be, approximately, of the following form:

A relation \mathcal{R} is a $\langle \rangle$ -*simulation* if whenever $(P, Q) \in \mathcal{R}$ and $P \xrightarrow{\mu} P'$, then Q', μ' exist s.t. $Q \xrightarrow{\mu'} Q', \mathcal{B}(\mu, \mu')$ and $\forall C \in \mathfrak{C}$, if $P'' = f(C, \mu, P')$ and $Q'' = f(C, \mu', Q')$, then $P'' \mathcal{R} Q''$. (*)

\mathcal{R} is a $\langle \rangle$ -*bisimulation* if \mathcal{R} and \mathcal{R}^{-1} are $\langle \rangle$ -simulation.

Here, \mathfrak{C} , \mathcal{B} and f are special predefined class, predicate and function, respectively. Often, the quantification on \mathfrak{C} is missing and hence the definitions of the processes P'' and Q'' only depend upon the actions μ and μ' and the processes P' and Q' . When \mathfrak{C} is present, it will represent a class of contexts. Then we define

$$\langle \rangle = \bigcup \{ \mathcal{R} : \mathcal{R} \text{ is a } \langle \rangle\text{-bisimulation} \}$$

Indeed $\langle \rangle$ will always represent the *largest* $\langle \rangle$ -bisimulation. Now the weak version $\langle \langle \rangle \rangle$ of $\langle \rangle$ is obtained from (*) by simply replacing the symbol $\langle \rangle$ with $\langle \langle \rangle \rangle$ and the transition $Q \xrightarrow{\mu'} Q'$ with $Q \xRightarrow{\widehat{\mu}'} Q'$. (In addition to this one might also replace the transition $P \xrightarrow{\mu} P'$ with $P \xRightarrow{\mu} P'$. As usual in bisimulations for process algebras, this would not affect the $\langle \langle \rangle \rangle$ -bisimulation derived, but the definition would be more difficult to verify.)

2.5.2 Congruences

Most of the bisimulations we shall consider are preserved by all operators except prefixing when it involves the input of names. To obtain the full congruence it is enough to require bisimilarity over all first-order substitutions (as done also in [MPW92]). Thus, the congruence $\langle \rangle^c$ of the bisimulation $\langle \rangle$ is defined as follows:

$$A \langle \rangle^c A' \text{ if for all } \tilde{x}, \tilde{y} \text{ with } \tilde{x} : \tilde{y}, A\{\tilde{x}/\tilde{y}\} \langle \rangle A'\{\tilde{x}/\tilde{y}\}.$$

In our symbology for the behavioural equivalences, the appearance of a superscript “c” means that a relation is a congruence over all operators.

2.5.3 Bisimulations up-to

Definition (*) in Section 2.5.1 requires that $P'' \mathcal{R} Q''$. If we want more flexibility, so to reduce the size of the relations to exhibit for proving a $\langle \rangle$ -bisimilarity, we might try to exploit a *bisimulation up-to* technique ([Mil89,MPW92,SM92]). Obviously, before using any of these techniques, its soundness must be guaranteed, by proving that the relations it defines are contained in $\langle \rangle$.

In the most common up-to techniques, the closure of \mathcal{R} is achieved modulo some supporting privileged relation \bowtie . The definition of strong $\langle \rangle$ -bisimulation up-to \bowtie is obtained from (*) by replacing $P'' \mathcal{R} Q''$ with $P'' \bowtie \mathcal{R} \bowtie Q''$. Since $\langle \rangle$ is a strong relation, \bowtie must be strong too; often it is taken to be $\langle \rangle$ itself. When adapting this technique to the weak case, some care is necessary. If \bowtie is a “weak” relation, it is not sound in general simply to replace the transition

$Q \xrightarrow{\mu'} Q'$ with $Q \xrightarrow{\widehat{\mu'}} Q'$ ([SM92]); in addition to this, at least one of the two following requirements is necessary:

- (a) use \bowtie only on the right of \mathcal{R} , i.e. ask $P'' \mathcal{R} \bowtie Q''$,
- (b) replace also the transition $P \xrightarrow{\mu} P'$ with $P \xrightarrow{\mu} P'$.

Another up-to technique, first proposed in [MPW92], is called *bisimulation up-to restriction*. The idea is to achieve the closure of \mathcal{R} modulo possible cancellation of the outermost restrictions. To define $\langle \rangle$ -bisimulation up-to restriction, the modifications of definition (*) involves only the clause for silent actions, which now becomes:

- whenever $P \xrightarrow{\tau} P'$, then Q' exists s.t. $Q \xrightarrow{\tau} Q'$ and for some P'', Q'', \tilde{x} , it holds that $P' = \nu \tilde{x} P''$, $Q' = \nu \tilde{x} Q''$ and $P'' \mathcal{R} Q''$.

For bisimulation up-to restriction the extension to the weak case is smooth: the replacement of the “strong” arrow $Q \xrightarrow{\mu'} Q'$ with the “weak” arrow $Q \xrightarrow{\widehat{\mu'}} Q'$ is enough.

The two up-to techniques considered above can also be mixed together, yielding a *bisimulation up-to \bowtie and up-to restriction*. In this technique, the clauses for visible actions are the same as for bisimulations up-to \bowtie ; the clause for τ -actions is the same as for bisimulation up-to restriction, but with $P'' \bowtie \mathcal{R} \bowtie Q''$ (possibly only $P'' \mathcal{R} \bowtie Q''$ in the weak case) instead of $P'' \mathcal{R} Q''$.

2.6 Bisimulation and congruence for π -calculus

We conclude the chapter with the definition of bisimulation and congruence for π -calculus. Previous work on π -calculus has individuated two different ways of presenting bisimulation, distinguished as *late* and *early*. Intuitively, the discrepancy between them arises in the instantiation of the formal parameter of an input: In the early case the instantiation occurs at the moment of inferring the input action, whereas in the late case at the moment of inferring a communication. In

other words, in the early case the receipt of an object on a port is viewed as one atomic event, whereas in the late case it is viewed as the composition of two more atomic events, namely first the commitment to a port and then the acquirement of the object. The separation of these two events yields a stronger equivalence than the corresponding late one [MPW92,MPW91].

To see the difference between the two with an example, consider the processes (from [MPW92])

$$P \stackrel{\text{def}}{=} a(x).R + a(x).\mathbf{0} \quad Q \stackrel{\text{def}}{=} P + a(x).[x = b]R$$

where R is any non-null process. These processes are early bisimilar, since depending on whether the value y received on a is equal to b or not, Q 's last summand is equal to P 's first summand or to P 's second summand. But P and Q are distinguished in the late bisimulation, where the choice of the value y with which to instantiate the bound name x is done “later” than the choice of the input prefix to consume. Therefore, Q 's commitment to $a(x).[x = b]R$ cannot be matched by P , since different instantiations of x distinguish $[x = b]R$ from R and $\mathbf{0}$.

In the thesis, we shall stick to early bisimulation. There are various reasons for this. Firstly, exploiting the early transition system presented in Section 2.2.2, early bisimulation can be given a simpler definition, in which the clause of bisimilarity is unique for all actions. By contrast, late bisimulation (even using a late transition system) has to distinguish the clause for input actions from the clause for output actions. Secondly, early bisimulation is the one which is recovered with barbed bisimulation, as shown in the next section. Thirdly, we think that the intuition behind early bisimulation — the atomicity requirement on input actions — is more convincingly.

Definition 2.6.1 (early bisimulation) *A relation \mathcal{R} on π -calculus processes is an early simulation if whenever $P \mathcal{R} Q$ and $P \xrightarrow{\mu} P'$ with $\text{bn}(\mu) \cap \text{fn}(P, Q) = \emptyset$, then Q' exists s.t. $Q \xrightarrow{\mu} Q'$ and $P' \mathcal{R} Q'$; \mathcal{R} is an early bisimulation, briefly \sim_e -bisimulation, if \mathcal{R} and \mathcal{R}^{-1} are early simulations. Two process P and Q are early bisimilar, briefly $P \sim_e Q$, if $P \mathcal{R} Q$ for some early bisimulation \mathcal{R} . \square*

Early bisimulation is a congruence over all operators but input prefixing (the same is true for late bisimulation).

Example 2.6.2 Let $x : y$ and $P \stackrel{def}{=} x|\bar{y}$, $Q \stackrel{def}{=} x.\bar{y}+\bar{y}.x$. Then $P \sim_e Q$. However, it holds that $a(x).P \not\sim_e a(x).Q$, since $P \xrightarrow{a(y)} \tau \rightarrow \mathbf{0}$, which $a(x).Q$ cannot match. \square

The full congruence is called *early congruence* and denoted by \sim_e^c . It is defined in the standard way, as described in Section 2.5.2; that is, $P \sim_e^c Q$ if for all first-order substitutions σ , it holds that $P\sigma \sim_e Q\sigma$. The corresponding of \sim_e and \sim_e^c in the weak case are called *weak early bisimulation* and *weak early congruence*, and denoted by \approx_e and \approx_e^c , respectively.

Chapter 3

Barbed Bisimulation

3.1 Motivations

We have discussed in Section 1.3 the problems for the definition of a natural bisimulation in a higher-order process calculus and the importance of finding a notion of bisimulation which can be used uniformly in different calculi.

The idea is to try to achieve this by equipping a global observer with a *minimal* ability to observe actions and/or process states. We then obtain an equivalence, namely indistinguishability under global observations. This in turn induces a congruence over agents, namely equivalence in all contexts. The question is: what minimal power of observation is needed so that the congruences induced in this way coincide with the familiar bisimulation congruences?

It is reasonable to examine first the case in which there are *no* observables. Unfortunately the *reduction congruence* which is so obtained is in general not discriminating enough. Therefore it is necessary to add the capability of observing some properties of the states. It seems natural in concurrency that the extra power provided is in terms of action observability; our choice has been to give the external observer visibility of the channel at which an action occurs. We call the resulting bisimulation relation *barbed bisimulation*, because somehow it adds “barbs” to the reduction congruence (as such resembles what A. Pnueli does in [Pnu85] w.r.t. trace semantics). The purpose of this chapter is to show that barbed bisimulation plus parametrisation over contexts is enough to give the desired discriminating

power.

An important feature of barbed bisimulation is that it can be successfully employed in the setting of reduction semantics. It allows us to recover from such a formulation the well-known bisimulation-based equivalences which are defined the labelled transition system, an important requirement for the adoption of reduction semantics.

Viewing it differently, the “labeled” equivalences, i.e. the ordinary equivalences of the labelled transition systems, since they do not involve quantification over contexts, can be seen as direct characterisations of the “barbed” equivalences. Indeed, the barbed equivalences may act as support for the labeled equivalences (similarly as reduction semantics does for labelled transition semantics). For instance, a barbed equivalence may guide us to the definition of the labeled equivalence and ensure us of its sensefulness. These benefits may be important when tackling the analysis of a new calculus, for which barbed bisimulation immediately suggests a natural congruence. Furthermore, it becomes an excellent test for the calculus and its operators to see whether they can express a direct characterisation of this congruence. All this is exemplified in the study of $\text{HO}\pi$ carried out in Chapter 4.

In the π -calculus the use of barbed bisimulation has other interesting outcomes. Firstly: We have seen in Section 2.6 there are two major philosophies for defining bisimulations in the π -calculus, referred as *early* and *late* and the studies on π -calculus so far appeared in the literature have not clarified yet which one should be preferred. Then the question is: Which of the two – if any – is it captured using the barbed bisimulation machinery? We shall see that the answer is the early version — a good point, we think, in its favour.

Secondly: It is generally recognised that a natural equivalence should be *observational*. That is, two systems are equivalent whenever by interacting with them from the outside world, no difference can be observed. But then one might argue that the ordinary bisimulation for the π -calculus — in the late or early version — is not quite observational: One is supposed to be able to distinguish whether a name received in a communication is bound by a restriction or not, and properly, this is something which cannot be considered as *visible*. These doubts

are cleansed by appealing to the characterisation in terms of barbed bisimulation, since the predicates employed in the definition of the latter have an irrefutable “truly observational” mark.

As an aside let us just point out that the possibility of parametrising barbed bisimulation over a particular class of contexts seems to have other interesting applications. In Chapter 6 it will be used to obtain a fully abstract model for lazy λ -calculus from its encoding into π -calculus and $\text{HO}\pi$. We would also like to apply this parametrisation in the framework of action refinement. Intuitively, since the refinement of an action modifies the communication protocol of a process, only contexts which “respect” such a protocol should be considered when verifying the equivalence between two refined processes.

Although barbed bisimulation will be examined here in the setting of process algebra, the idea is more general and, in fact, can be used in any term rewriting structure $\{Pr, \longrightarrow, \{\downarrow_a\}_a\}$, where Pr is the set of terms, \longrightarrow is the reduction relation and $\{\downarrow_a\}_a$ is a certain set of observation predicates (those producing the bars). Thus, reduction congruence would correspond to the case in which this set of predicates is empty. Moreover, in λ -calculus, using the observation predicates to detect the possibility of convergence of a term, barbed bisimulation induces Abramsky’s applicative bisimulation [Abr89] (see Section 6.3.1).

Structure of the chapter: We start Section 3.2 by introducing reduction bisimulation and congruence; we show that in general these are over-generous; to remedy, we move to barbed bisimulation. In Section 3.3 we prove that in CCS and π -calculus, the congruences induced by barbed bisimulation coincides with the ordinary bisimulation-based equivalences.

3.2 From reduction bisimulation to barbed bisimulation and congruence

In this section, although the examples are in CCS or π -calculus, in the definitions we are not restricted to any specific process calculus. We only require that

- the calculus possesses a reduction relation (which is written as $P \longrightarrow P'$);
- processes use channels for communications, and for each channel a there is an *observation predicate* \downarrow_a detecting the possibility of performing an action along a .

We use Pr to denote the class of processes of the calculus and P, Q to range over Pr . As usual, \Longrightarrow is the reflexive and transitive closure of \longrightarrow ; moreover we use $P \Downarrow_a$ to mean that $P \Longrightarrow P' \downarrow_a$, for some P' .

Example 3.2.1 Let P be the π -calculus process $\bar{a}\langle y \rangle + b(x) + \tau.c$. We have $\{z : P \downarrow_z\} = \{a, b\}$. Notice that $P \not\downarrow_c$, since P cannot perform immediately an action at c . Moreover we have $\{z : P \Downarrow_z\} = \{a, b, c\}$. \square

Reduction bisimulation

From a process calculus point of view, reduction bisimulation represents an attempt to recover familiar bisimulation-based equivalences — at least in the strong case — by focusing only on reduction, the simplest form of action.

Definition 3.2.2 Reduction bisimulation is the largest symmetric relation $\dot{\sim}_{\text{red}} \subseteq Pr \times Pr$ s.t. $P \dot{\sim}_{\text{red}} Q$ and $P \longrightarrow P'$ imply that some Q' exists with $Q \longrightarrow Q'$ and $P' \dot{\sim}_{\text{red}} Q'$. \square

By itself, $\dot{\sim}_{\text{red}}$ is not very interesting. It is rather weak; in general it is even not preserved by parallel composition, as the following example shows:

Example 3.2.3 Let $P \stackrel{\text{def}}{=} a.\mathbf{0}$ and $Q \stackrel{\text{def}}{=} \mathbf{0}$. Then it holds that $P \dot{\sim}_{\text{red}} Q$, but $P \mid \bar{a} \not\dot{\sim}_{\text{red}} Q \mid \bar{a}$. \square

It is natural then to consider the congruence induced by $\dot{\sim}_{\text{red}}$.

Definition 3.2.4 Two processes P and Q are reduction congruent, briefly $P \sim_{\text{red}}^c Q$, if for each context $C[\cdot]$, it holds that $C[P] \dot{\sim}_{\text{red}} C[Q]$. \square

In [MS92], \sim_{red}^c is compared with CCS's strong bisimulation. It is shown that the latter implies the former but that in general the converse fails. To see this, let $P \stackrel{\text{def}}{=} \tau.P$ and $Q \stackrel{\text{def}}{=} \tau.Q + a.P$. They are not strongly bisimilar. However they are reduction congruent (this example is due to Gerard Boudol). Intuitively, since any state that P and Q can reach is *divergent*, i.e. can evolve through an unbounded number of τ -actions, no CCS context can make the distinction between these processes as long as only τ -actions are taken into account. Indeed, in CCS divergency is exactly what makes the two relations different. They coincide on the class of divergence-free processes [MS92].

To conclude, since reduction congruence is not discriminating enough (furthermore in the weak case it corresponds to the universal relation), the power of the observer has to be increased. We do this by adding “barbs” to reduction bisimulation.

Barbed bisimulation

Definition 3.2.5 *A relation $\mathcal{R} \subseteq Pr \times Pr$ is a barbed simulation if $(P, Q) \in \mathcal{R}$ implies:*

1. *whenever $P \longrightarrow P'$ then $Q \longrightarrow Q'$ and $(P', Q') \in \mathcal{R}$;*
2. *for each channel a , if $P \downarrow_a$ then also $Q \downarrow_a$.*

A relation \mathcal{R} is a barbed bisimulation if \mathcal{R} and \mathcal{R}^{-1} are barbed simulations. Two processes P and Q are barbed-bisimilar, briefly $P \dot{\sim} Q$, if $(P, Q) \in \mathcal{R}$, for some barbed bisimulation \mathcal{R} □.

Barbed bisimulation describes a game between two machines which can challenge one another on the reductions they can perform, under the condition that the states reached must have the same observation sets. To obtain *weak barbed bisimulation*, briefly $\dot{\approx}$, just replace in the above definition the transition $Q \longrightarrow Q'$ with $Q \Longrightarrow Q'$ and the convergence predicate $Q \downarrow_a$ with $Q \Downarrow_a$.

As for reduction bisimulation, by itself barbed bisimulation is too weak; we use contexts to get a finer relation. If we use the class of all contexts, we obtain the relation called *barbed congruence*.

Definition 3.2.6 *Two processes P and Q are barbed-congruent, written $P \sim^c Q$, if for each context $C[\cdot]$, it holds that $C[P] \dot{\sim} C[Q]$. \square*

The most studied bisimulations in the weak case usually are not preserved by *dynamic* operators, i.e. operators like prefixing or sum which can be discharged when an action is produced. To recover these equivalences, we have to parametrise $\dot{\sim}$ over a *subclass* of all possible contexts. The obvious representative is the subclass \mathcal{SC} of *static contexts*, that is contexts which are built by composing $[\cdot]$ and processes by means of only static operators. An operator \otimes is called *static* if the rules describing its behaviour have the conclusion of the form $\otimes(P_1 \dots P_n) \xrightarrow{\alpha} \otimes(P'_1 \dots P'_n)$. Formally, the class \mathcal{SC} is defined by the grammar:

$$\mathcal{SC} := [\cdot] \mid P \mid \otimes(\mathcal{SC}_1, \dots, \mathcal{SC}_{r(\otimes)})$$

where P is a process, \otimes is a static operator and $r(\otimes)$ is its arity and with the restriction that the hole $[\cdot]$ occurs at most once in an expression.

Definition 3.2.7 *Two processes P and Q are barbed equivalent, written $P \sim Q$ if for each static context $C[\cdot]$, it holds that $C[P] \dot{\sim} C[Q]$.*

We shall denote by \approx^c , \approx , the corresponding of \sim^c and \sim in the weak case. The extensions of these relations to abstractions and open agents will be considered in Section 4.7.2. We said in Section 2.5.2 that in the thesis we follow the convention that a symbol indexed by a superscript “c” denotes a relation which is a full congruence; similarly undecorated symbols will be used to denote relations which are congruences over static contexts.

A brief discussion regarding the observation predicates employed in the definition of barbed bisimulation is pertinent, for these are not the only predicates one might think of using. The simplest possible predicate is the one adopted in [MS92], which just detects whether *some* observable action can be performed. It seems that in the strong case this indeed suffices to induce the desired congruences; but the answer is still unknown in the weak case.

Another possibility is to partition observable actions into subsets and then only to allow the external observer to discriminate between actions from different subsets. Notice that the predicates in Definition 3.2.5 represent an instance of this, in which the subset is determined by the subject of the action. Some preliminaries studies that we have conducted in this direction suggests that a partition into two subsets, for instance the set of input and the set of output actions, might be enough to recover the CCS ordinary bisimulations.

We prefer to leave for future work a detailed analysis of what is the best balance between power of observer and ease of manipulation. But at least, the choice that we have made here appears to be a reasonable one, and it works perfectly well in all cases we have considered.

3.3 Discriminating power induced by barbed bisimulation

In this section, as a test for barbed bisimulation, we show that in CCS and in the π -calculus it allows us to recover the familiar bisimulations and their congruences. The schema of the proof in all these results is the same. We only give the details for CCS, since it is a simpler case, whereas we have relegated those for the π -calculus in Appendix A.

3.3.1 The CCS case

As mentioned in Section 2.1.2, CCS is derived from the polyadic π -calculus by imposing the sorting $\{Name \mapsto ()\}$. Also the ordinary bisimulation of CCS coincide then with π -calculus's early bisimulation (Definition 2.6.1); we recall the former below for commodity.

Definition 3.3.1 (bisimulation for CCS) *A relation \mathcal{R} on CCS processes is a simulation if whenever $P \mathcal{R} Q$ and $P \xrightarrow{\mu} P'$, then Q' exists s.t. $Q \xrightarrow{\mu} Q'$ and $P' \mathcal{R} Q'$; \mathcal{R} is a bisimulation if \mathcal{R} and \mathcal{R}^{-1} are simulations. Two process P and Q are bisimilar, written $P \sim_{\text{ccs}} Q$ if $P \mathcal{R} Q$ for some bisimulation \mathcal{R} . \square*