The π-calculus

A Thesis
Presented to
The Division of Mathematics and Natural Sciences
Reed College

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Arts

William Henderson

May 2008

Approved for the Division
(Mathematics)

James Fix

A Word About The Format of This Thesis

One of the pedagogical aims of this thesis is to be clear and reader-assistive during its presentation. To that end, I have taken several liberties in the formatting of this thesis. Figures, equations, definitions, theorems and examples all share one numbering scheme in hopes that it will make them easier to locate. Where important *definitions* appear, they are clad in italics and sit in a box on the margin to make them easier to find. When definitions or equations are referred to later in the text, an assistive link will appear in the margin to avoid index-fingering. In addition, the wired reader will find a searchable pdf version of the text at the following url:

definitions

definitions
↪ page i

http://wcrawford.org/thesis

All references in this pdf are hyper-textual (clickable).

Acknowledgments

I want to thank my hamsters, Boris Becker, and this bottle of Merlot.

Table of Contents

Chapter 1: Introduction **1**
1.1 Distributed Systems 2
1.2 Process Algebra 3
1.3 The π-calculus: An Introduction 4
1.4 An Outline of This Thesis 6

Chapter 2: The Asynchronous π-Calculus **7**
2.1 Syntax 7
2.2 Structural Equivalence 10
2.2.1 Identifier Substitution and α-Equivalence 10
2.2.2 Contexts and Equivalence 12
2.3 Reduction Semantics 13
2.4 Action Semantics 15
2.5 Extended Example: Memory Cells 19

Chapter 3: The Synchronous π-Calculus **24**
3.1 The Synchronous π-Calculus 25
3.2 Computation in The Synchronous π-Calculus 26
3.3 Extended Example: Leader Elections 28

Chapter 4: Synchronicity and Distributed Systems **31**
4.1 Separation Results 31
4.2 Encoding Choice 33
4.3 A 'Bakery' Algorithm 35

Chapter 5: Conclusions **38**

List of Figures **40**

References **40**

List of Figures

1.1.1 *A distributed mobile system* 2
1.3.1 *Simplified mobile phone network in the π-calculus* 5

2.1.1 *Terms in the asynchronous π-calculus* 7

3.1.1 *Terms in the synchronous π-calculus* 25
3.1.2 *Structural equivalence axioms in the synchronous π-calculus* 26
3.2.1 *Reduction rules for the synchronous π-calculus* 27
3.2.3 *Action rules for the synchronous π-calculus* 28
3.3.1 *Rules for applying σ* 29

Abstract

In this thesis I will blow your mind, and possibly (hopefully?) my own in the process.

1
Introduction

Computer scientists have long been interested in algebraic models that are capable of describing computation by formulating a 'program' as an algebraic expression along with a set of rules for reducing such expressions — i.e. 'running' them. The advantage of such algebras is that they can be studied using formal reasoning techniques. This means that we can rigorously prove things about them and derive useful observations 'programming' in them without actually ever having to implement them on a system. Of course, these algebras often are used in real languages whether faithfully or in part, but this is usually after they have been studied in detail for some time and we can be sure of their value.

In the 1930's, Alonzo Church and Stephen Kleene introduced one such algebra, known as the λ-calculus. In the λ-calculus, we can write programs as expressions that are a series of nested functions, and we can run these programs by invoking the functions within. A function is indicated by the λ symbol, followed by a single variable representing its input. For example, the following is a very simple 'successor' program that takes in a number x and adds 1 to it, returning the result:

$$(\lambda x.x + 1)$$

To run this program, we need to actually apply it to something — that is, we need to give it input. We express this by placing the input to the right of the function, as in:

$$(\lambda x.x + 1)4$$

The above program first substitutes 4 in for the input x, yielding 4+1. The program then returns the result, 5. Since we can nest functions, a slightly more complicated version adds 1 to the input 4 and then multiplies the result by 3. We give this program with each computational 'step' below:

$$\begin{aligned}(\lambda y.y * 3)((\lambda x.x + 1)4)\\(\lambda y.y * 3)(4 + 1)\\(\lambda y.y * 3)5\\5 * 3\\15\end{aligned}$$

You are probably already getting a sense of the expressive power of the λ-calculus. In fact, it is capable of expressing *any* computer algorithm, even without the numbers

and arithmetic operators we have implicitly included above! Besides allowing us to study the nature of computation in the abstract, the λ-calculus went on to inspire many programming languages like Lisp, ML and Haskell, and even some functionality in languages like Smalltalk, Ruby and Python.

1.1 Distributed Systems

distributed systems

The discovery of the λ-calculus did not end the search for new algebras. More recently, new kinds of programs and demands have emerged with the computational platform of *distributed systems*. These systems consist of loose networks of machines capable of exchanging messages and information. We say this shared information and computational power belongs to 'the system' in that any program running on the machines of the system can freely access these resources, coordinating efforts to produce a single complex behavior.

Jim: wasn't sure exactly how to phrase this. ok like this?

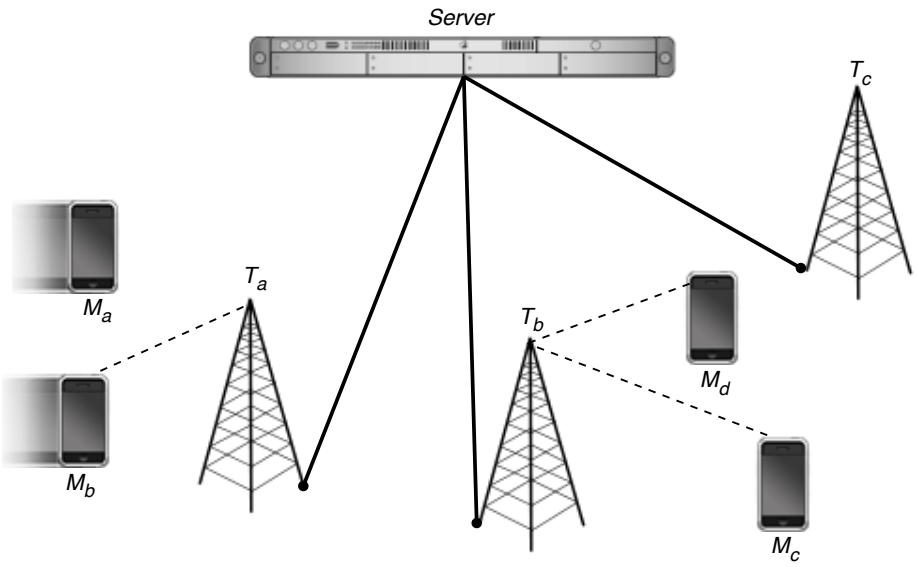

Figure 1.1.1: *A distributed mobile system*

For example, consider the familiar system that powers your mobile phone network. There may be one or more connected central servers, all of which are connected to the various towers that provide service. Towers and servers may have different capabilities and responsibilities, but the important thing is that the entire system needs to behave as a single unit with a bunch of shared information and computational power. A call in progress, for example, uses capabilities at various places in the system — by a tower to handle the call, by a server, perhaps, to handle the billing and routing of that call.

Mobile clients are also connected and a part of this system. When a user places a call, for example, he may use some capabilities on the phone to input the number, which is transmitted by the phone to the tower and then sent to the server to be

routed. This entire 'program' for making a call is one seamless operation that is happening across several machines.

The demands of a distributed system strain the expressive power of the λ-calculus. Since the λ-calculus models all computation via functions, our only means of modeling a resource is as a function. Functions can only be accessed directly – input can be passed to a function only by directly applying the function to that input. But if we want access to a resource on a system, we might not know where it is or what it is doing. Furthermore, our system of many machines could be doing many different things at once: routing calls, handling other calls in session, calculating a bill. Yet the λ-calculus has no means of easily expressing this concurrency which is so basic to many distributed systems.

Consider the phones on our system. They are *mobile* — in the sense that their connections to the system can be added and removed at any time. In Figure 1.1.1 above, client M_b is wirelessly connected to tower T_a while M_c, M_d are connected to T_b. Client M_a is currently disconnected and T_c has no clients. All the towers maintain a hard-wired link to the *server*. We refer to the connections in a distributed system as its communication *topology*.

mobility

topology

Furthermore, M_a and M_b are in physical movement and their connections may change soon. M_b might, for example, go out of range and disconnect from T_a and connect to T_b instead. Such changing communication topologies present even more difficulties to the λ-calculus. How can we abstract function invocation such that a function can be called from one place at one time, and another place later? Even if we could somehow invoke a function indirectly, how could we deal with the fact that a function (say a client's ability to receive a call) is not currently available?

1.2 Process Algebra

Clearly we need an algebraic model for computation that eases the difficulty of modeling such systems. Such a model might consider computation via its natural distributed unit — the *process*. A process is just a computational task, without reference to where it might be run nor with what input. For example, we assign a process to a conversation on a phone somewhere in our network. Since concurrency of processes is such a basic operation, we make it a part of our algebra. A system, then, is simply a group of processes which are executing concurrently. An important thing about processes is that they maintain computation state independently of one another. Instead of a single program state where functions interact via invocation, processes run independently and interact via *message passing* – sending data back-and-forth via named channels. These channels can be shared between some or all processes in the system, but a single instance of communication is always between a pair of processes.

process

message passing

Because these channels can be shared among processes and used an arbitrary number of times, channels are not a concrete invocation system for a chunk of computation the way a function call is — processes simply send values to channels, assuming the receiver (if there is one) will do something useful with it. As with functions in the λ-calculus, processes are the basic unit of a program in the π-calculus. Any bit of

functionality can be referred to as a process, with no specification of the granularity.

In 1978 C.A.R. Hoare introduced an early process algebra called CSP[Hoa78]. Another major step came in the 1980's when Robin Milner introduced his Calculus of Communicating Systems (CCS)[Mil82]. The CCS modeled systems as groups of communicating processes interacting via shared channels, and drastically eased the difficulty of modeling indirectly invoked concurrent operations. However, the CCS still would have had trouble with our mobile phone network, because it did not provide a way for processes to gain and lose their communication channels.

Although it can be defined in other ways, one of the ways of giving a process's *location* is in terms of the communication channels which can be used to access it. Since processes are the units populating the space of a CCS system, it is natural to think of a process's location in terms of the processes which are 'near' it — those it can connect to. Since communicate happens via channels, a connection between processes just means that they share at least one channel. In this way, changing the communication channel topology of a system can be seen as changing the locations of its component processes.

In the CCS, channel topology is static — it does not allow new connections to be made or old ones to be removed. Not long after CCS's birth, Robin Milner, Joachim Parrow and David Walker created an improved algebra called the π-calculus. The π-calculus allows communication channels to be dynamically established and relinquished between processes. Since channels are what define location, dynamically created and destroyed channels give a kind of *mobility* of processes, which vastly expands the capabilities of interaction in a system and finally allows us to give an account of our mobile phone system.

1.3 The π-calculus: An Introduction

As an example of how naturally the π-calculus models distributed systems, let us again consider our mobile phone network[1]. Our system will be simplified a bit: only two towers and one phone, with the only system functionality being talking on the phone or switching from one tower to another.

First we give a process describing the behavior of a mobile phone:

$$Mobile(talk, switch) \Leftarrow talk!\langle\rangle.Mobile\langle talk, switch\rangle + switch?(t, s).Mobile\langle t, s\rangle$$

Above, we use $\Leftarrow$ to mean that the term to the right of the $\Leftarrow$ is given the shorthand $Mobile$ and that it uses the channels $talk$ and $switch$ given elsewhere. When the shorthand is used, which occurs with angle brackets $\langle\rangle$, the channels given in the brackets are simply substituted for the terms in the parenthesis. Hence, for example, $Mobile\langle t, s\rangle$ would be the right-hand term with the channels t and s substituted for $talk$ and $switch$. We call $Mobile(talk, switch)$ the *interface* of the right-hand term.

interface

[1] This presentation is adapted from a version first given in [Mil99]

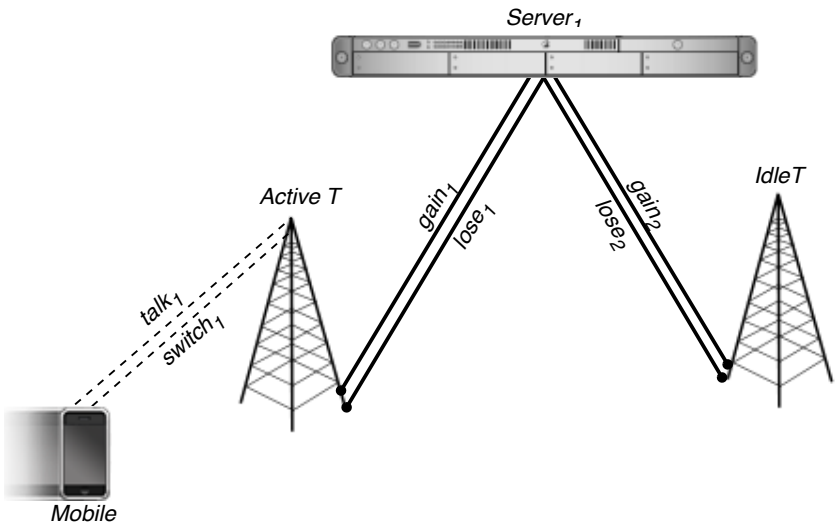

Figure 1.3.1: *Simplified mobile phone network in the π-calculus*

The notation $talk!\langle\rangle$ means that we are sending an 'empty' (since there are no input values given in the brackets) signal on channel $talk$, while $switch?(t, s)$ means we are listening on $switch$ and receive the input t, s. After a signal is sent or received on a channel, execution then continues with the term immediately to the right of the term, using the input provided (if any).

Thus, $switch?(t, s).Mobile\langle t, s\rangle$ indicates that we should listen on $switch$ for the input t, s and then use t, s to 'spawn' a new computation of $Mobile$ with these new channels. The term $talk!\langle\rangle.Mobile\langle talk, switch\rangle$ means send a signal on $talk$ before spawning a computation of $Mobile$ with the same $talk, switch$ channels we are using. Finally, the + operator denotes that we should choose to compute one or the other of the operands (but not both). In sum then, $Mobile$ has the ability to either send along $talk$ and stay in its current location, or it can receive on $switch$ and 'move' to the location where it has new talking and switching channels t, s. This last capability expresses the mobile nature of our processes with surprising elegance: here we have just expressed that channels $talk, switch$ are dropped and new channels t, s are established.

Next we consider the behavior of a tower:

$$\begin{aligned} ActiveT(talk, switch, gain, lose) \Leftarrow\ & talk?().ActiveT\angle talk, switch, gain, lose\rangle \\ & + lose?(t, s).switch!\langle t, s\rangle.IdleT\langle gain, lose\rangle \\ IdleT(gain, lose) \Leftarrow\ & gain?(t, s).ActiveT\langle t, s, gain, lose\rangle \end{aligned}$$

Here, the are two 'states' our tower can be in. It can be active in talking with a mobile phone, or it can be disconnected and idle. If it is active, it can either receive on $talk$ (from the phone) and continue to be active, or it can receive on $lose$ (from the server, as we shall see below). In the latter case it then sends t, s on $switch$ before becoming idle. That is, it will receive some new channels t, s on $lose$ and then send them to

the phone on *switch* before becoming idle. An idle tower has only one capability: to receive on *gain* and then become active again on the new channels t, s.

Finally we give the behavior of the server:

$$\begin{aligned} Server_1 &\Leftarrow lose_1!\langle talk_2, switch_2\rangle.gain_2!\langle talk_2, switch_2\rangle.Server_2 \\ Server_2 &\Leftarrow lose_2!\langle talk_1, switch_1\rangle.gain_1!\langle talk_1, switch_1\rangle.Server_1 \end{aligned}$$

Above, the server has two states: it can be controlling tower 1 or it can be controlling tower 2. In either case, the only capability is to lose one tower and gain the other before going into the opposite state. Hence, in $Server_1$ we can send to the first tower on $lose_1$ and then to the second tower on $gain_2$ before entering state $Server_2$.

We have now completely described the components of our system, but we still haven't put them all together. To do so, we'll need a new operator. This operator denotes that its operands run concurrently in parallel and is denoted $|$. Thus, our system is simply:

$$Mobile\langle talk_1, switch_1\rangle | ActiveT_1\langle talk_1, switch_1, gain_1, lose_1\rangle | IdleT\langle gain_2, lose_2\rangle | Server_1$$

1.4 An Outline of This Thesis

Now that we have gotten a taste of the power of the π-calculus, we are ready to explore it in more detail. However, our first pass at the π-calculus will not include some of the operators included in our mobile phone network example. For one, it will not include the + operator. We will also use a version of sending that is a little simpler than the one we've seen so far. More specifically, our first calculus will be *asynchronous*, meaning that processes will not continue on with any computation after sending on a channel. That is, in a process like

asynchronous

$$c!\langle\rangle.P,$$

we will disallow the presence of P – the process will simply terminate after sending on c. In Chapter 2, we will give a rigorous presentation of the semantics of this asynchronous π-calculus and the rules describing they way computation happens.

The two simplifications mentioned above will make our calculus somewhat easier to grasp, but there is another reason to consider an asynchronous calculus: it is far easier to implement on a real distributed system than the full synchronous version. At the base level, communication between machines on a distributed system is asynchronous, so implementing synchronous calculi requires finding a way to simulate synchronous message passing using only asynchronous primitives. In Chapter 3, we will present the *synchronous* π-calculus(which has the full expressive power used in the mobile phone network example) and its computational rules.

synchronous

Chapter 4 looks at the problem of modeling the synchronous π-calculus using the asynchronous version. It then takes a more practical-minded presentation of the way that a synchronous π-calculus-like language might be implemented using asynchronous primitive π-calculus constructs.

2

The Asynchronous π-Calculus

In this chapter, we will give the syntax of the *asynchronous* π-calculus and a discussion of its features. We will loosely follow the style of a recent presentation given in [Hen07]. Following this, we will introduce a notion of equivalence of terms in the language. There are two viewpoints from which the computation behavior of a process can be characterized. The first is a set of semantic reduction rules, given in Section 2.3, that focus on the way a process behaves in and of itself through internal evolution. The second viewpoint is more general and addresses how a process evolves in the context of a larger system. We give this important system in Section 2.4. Finally, we conclude the chapter with an extended example that provides an interesting π-calculus implementation while demonstrating many of its systems and rules.

2.1 Syntax

identifiers

The terms of the π-calculus operate on a space of *identifiers* which consists of names $a, b, c, ..., n, m, o$ for communication channels, and variables v, w, x which can refer to channels, and recursive variables p, q, r, explained in more detail below. In general, we will use capital letter to denote a process.

Process terms

$R :=$	$R_1 \mid R_2$	Composition
	$n!\langle\overline{V}\rangle$	Send
	$n?(\overline{X}).R$	Receive
	$\text{new}(n).R$	Restriction
	if $v_1 = v_2$ then R_1 else R_2	Matching
	rec $p.R$	Recursion
	stop	Termination

System

$\text{new}(c_1, ..., c_n).R_1 \mid ... \mid R_m$	$n, m >= 0$

Figure 2.1.1: *Terms in the asynchronous π-calculus*

Given two or more processes, we can compose them using the $\mid$ operator, which means that the composed processes will be executed concurrently.

TODO: Get index up to snuff by making sure more terms are margin defined.

arity

handshake

guarded

patterns

scope

scope extrusion

We denote the sending of a message $\overline{V}$ over a channel n by $n!\langle\overline{V}\rangle$. Here $\overline{V}$ is a tuple of identifiers in the form $\overline{V} = (v_1, ..., v_k) : k \geq 0$. We say that $\overline{V}$ has *arity* k. In the case $k = 0$ nothing is being transmitted; the communication acts as a *handshake* or signal. We will denote this case by $n!\langle\rangle$. When $k = 1$, only a single value v_1 is being transmitted, in which case we write $n!\langle v_1\rangle$. Because our calculus is asynchronous, sending is not a *guarded* operation – that is, a send operation does not continue to execute any process after sending its value, but simply terminates after sending the value. We will show in Example 2.1.9 that synchronous behavior can still be modeled in our language.

The term $n?(\overline{X}).R$ describes a process waiting to receive a tuple along n before continuing with R. Here $\overline{X}$ is a *pattern* – a tuple of variables of arity k – which can be used anywhere in R. Patterns allow us to decompose the transmitted tuple into its component values by naming them with $x_1, ..., x_k$, which can be referred to in R. Thus, in the term

$$c!\langle n_1, n_2, n_3\rangle \mid c?(x_1, x_2, x_3).R, \tag{2.1.2}$$

the names n_1, n_2, n_3 are received via c and correspond to the variables x_1, x_2, x_3 in the pattern. Hence, the variables x_1, x_2, x_3 can be used anywhere within the process R to mean n_1, n_2, n_3.

Similarly to sending, the case of arity 0 is denoted $n?().R$ – here R will not happen until a handshake is received on n. Notice that in contrast to the case of sending, receiving is a guarded operation – that is, the process R will execute after $\overline{X}$ has been received via n. For example, the term

$$c?().stop \tag{2.1.3}$$

represents a 'listener' process that simply consumes the value waiting on input channel c. The term *stop* describes a terminating process: one that simply halts.

The term $\text{new}(n).R$ describes a process in which a new channel name n is created and limited to being expressed in the process R (we say the *scope* of n is *restricted* to R). We shall see that scope plays a very big role in the way processes can establish new connections. Essentially, for a process P to have a connection to another process R really means that they both 'know' about a common channel c. In that case, c is scoped P and R. Hence, $\text{new}(n).R$ really means that we have created a channel n that – for the time being – only process R knows about. It is important to note that n *can* be used outside of R if it is sent and then received by some outside process. This feature is known as *scope extrusion*, and is the underpinning for the dynamic communication topologies introduced in the π-calculus. For example, in the term

$$\text{new}(n).(c!\langle n\rangle) \mid c?(x).x!\langle\rangle, \tag{2.1.4}$$

we have a channel n scoped to the left hand process which is sending n over c. The right hand process then receives n over c (referred to by x) and then sends an empty signal over n. At the time of its creation, n's scope is just the left half of the term. However, after n is received in the right hand process it will be able to be referred to

outside of its initial scope. We will give a precise account of how this happens in our reduction rules and Example 2.3.2 below.

We will sometimes abbreviate terms that use multiple channel restrictions by writing $\text{new}(n, m).R$ to denote the term $\text{new}(n).(\text{new}(m).R)$.

Simple conditional execution based on value equality comparison is available through the use of if $v_1 = v_2$ then R_1 else R_2. For example, in the following term, the value received on c is checked; if it matches a then a handshake is sent along a (which is referred to by x), otherwise the process terminates.

$$c?(x).(\text{if } x = a \text{ then } x!\langle\rangle \text{ else } \mathit{stop}) \tag{2.1.5}$$

Recursion is built into the language using the syntax $\text{rec } p.R$. The process $\text{rec } p.R$ itself is referred to by to the variable p, which is used somewhere in R to express a recursive call. For example, consider the recursive responder term below, which receives a channel x_1 via c. It then sends a handshake response on x_1, while another responder process is run in parallel.

$$\text{rec } p.c?(x_1).(x_1!\langle\rangle \mid p) \tag{2.1.6}$$

The variable p's scope is restricted R, just as $\overline{X}$ is restricted to R in $c?(\overline{X}).R$ and n is restricted to R in $\text{new}(n).R$.

The composition operator $\mid$ should be seen as having higher ordering priority than the other operators. Thus, for example, $\text{new}(n).R \mid Q$ should be read as $(\text{new}(n).R) \mid Q$. Operators that join using the dot (.) notation should always be taken as being grouped together. Thus, $\text{new}(c).c?(x).a!\langle b\rangle$ should be read as $\text{new}(c).(x?(x).a!\langle b\rangle)$ and not $(\text{new}(c).x?(x).)(a!\langle b\rangle)$.

We call a collection of parallel processes communicating over shared channels a *system*. Note that a system is itself a process. Though not strictly a primitive of the π-calculus, it is nonetheless a helpful construct for understanding the way that behavior can be modeled using processes as atoms.

Example 2.1.7 Many presentations (including our own in the next chapter) of the π-calculus involve a choice or summation operator as in $P+Q$ for two processes terms P and Q. The meaning is essentially that either P or Q can be non-deterministically executed, and the other process will not be. However, we can model the same behavior without defining a choice operator:

$$c!\langle\rangle \mid c?().P \mid c?().Q \tag{2.1.8}$$

In the above process, either Q or P (but not both) could be executed. This is due to the asynchronous behavior of our language – sending an empty signal along c, we cannot control which process consumes the signal (or even if it will be consumed at all). Thus, even without the choice operator our calculus is non-deterministic. ■

Example 2.1.9 Perhaps we are not happy with this asynchronous transmission behavior. Surely we'd like to have blocking sends sometimes, or be able to guarantee that a value is received. Our asynchronous calculus can model synchronous sending

by using a private channel that acknowledges when a value is received. To see how, consider the following system:

$$F_1(c) \Leftarrow \text{rec } z.(\text{new}(ack).(c!\langle ack\rangle \mid ack?().(R \mid z))) \tag{2.1.10}$$
$$F_2(c) \Leftarrow \text{rec } q.(c?(ack).(ack!\langle\rangle \mid q)) \tag{2.1.11}$$
$$Sys_1 \Leftarrow \text{new}(d).(F_1\langle d\rangle \mid F_2\langle d\rangle) \tag{2.1.12}$$

Above, both F_1 and F_2 have (infinite) recursive behavior, and we can think of Sys_1 as a term that 'kicks off' both of them after creating a shared channel for them to communicate on.

In an iteration of F_1, a new channel *ack* is created for acknowledgement. This channel is sent along c and then F_1 waits for input on *ack* before executing some term R and calling another iteration. F_2 receives *ack* along c and then uses *ack* to send an empty signal to F_1 that it has received input on c.

This is just a toy example: the only thing being communicated is the acknowledgement channel itself. We might imagine a more complex system where F_1 sends more input along c and waits to make sure F_2 receives it. Note that the channel *ack* is only used *once* – we want to ensure that the acknowledgement that F_1 receives is definitely for *that* instance of communication that it just initiated. This allows us to guarantee that F_2 has executed before F_1 continues with R. ■

2.2 Structural Equivalence

A natural question at this point is, given two π-calculus terms, how can we determine if they are equivalent? Intuitively, we want them to be equivalent if they *act* the same, but actually defining this equivalence relation can be a bit subtle. In exploring this issue, we will first look at identifier substitution, giving rules for when we can safely interchange identifiers without creating a different term. Following this, we will develop the notion of *contexts* and then use it to build an equivalence relation among processes.

2.2.1 Identifier Substitution and α-Equivalence

As a first step in our notion of equivalence, we might assert that the way identifiers are named shouldn't change how they act. However, this doesn't mean we can start interchanging symbols with carefree abandon. In a component process, some terms might be important for how the process acts in a larger system. For example, if we changed the channel that a mobile phone uses to communicate with a tower, that phone certainly wouldn't act the same – the tower would no longer know how to talk to it. On the other hand, we should be able to change any of the channels that the phone uses to communicate internally with itself without much issue.

bound identifiers ↪ page 7

The only identifiers we can safely change in a process without potentially affecting the way it behaves in a larger system are *bound identifiers*. Intuitively, bound identifiers are those which are introduced by some name-binding operator within the

process term. There are three π-calculus operators that bind identifiers. First, a name n is bound in the process term R to a new channel by the restriction operator as in new$(n).R$. Second, each channel variable x_i in the pattern X is bound in R to some channel v_i from the sending process when matched in a receive expression $c?(X).R$. Finally, the recursive variable x is bound within R in the recursive expression rec $x.R$, bound particularly to the whole process itself. We denote the set of bound identifiers in a term R by $bi(R)$; all those which are not bound we call *free*, denoting them $fi(R)$. Similarly, we denote the set of bound and free channel names in a term R by $bn(R)$ and $fn(R)$, respectively. We call a term with no free identifiers *closed*.

free identifiers

closed terms

For example, in the term:

$$\text{rec } p.c?(x_1).(x_1!\langle\rangle \mid p), \tag{2.2.1}$$

p is a bound recursive variable, while x_1 is bound by the receive operator. The channel c is free. Now consider the term

$$c!\langle n\rangle \mid \text{new}(n).(\text{rec } p.c?(x_1).(x_1!\langle\rangle \mid p)) \tag{2.2.2}$$

Here x_1, p are bound as before. However, the use of n being sent on c is *not* bound, since it occurs outside the scope of the restriction operator. In fact, the following term (with the restriction operator removed) is equivalent (see Example 2.2.9 for justification).

$$c!\langle n\rangle \mid \text{rec } p.c?(x_1).(x_1!\langle\rangle \mid p) \tag{2.2.3}$$

When a receive term like $c?(X).R$ is executed, we substitute the free variables $\overline{X}$ occurring in R with the values $\overline{V}$ that were received. We denote the term that results from such substitutions by $R[\![\overline{V}/\overline{X}]\!]$. During the course of a substitution, we might inadvertently 'capture' a bound term. For example, suppose the system

$$P\langle a\rangle \mid a!\langle n\rangle$$

where $P(a)$ is given by

$$P(a) \Leftarrow\ a?(x).(\text{new}(n).(n!\langle\rangle \mid x!\langle\rangle)) \tag{2.2.4}$$

In running $a?(x)$ we would perform the substitution $[\![n/x]\!]$. But the received term n is not the same as the n occurring in $P(a)$ – it was a free channel sent by a the process $a!\langle n\rangle$ outside the scope restriction new(n). Thus, the problem in performing $[\![n/x]\!]$ is that it would have the free name n from the outside process bound as if it where the same n bound by $P(a)$. We say that performing this substitution would *capture* the bound name n. We can avoid this by first replacing the bound n with a new name that is 'fresh' – that is it does not occur elsewhere in the term. For example:

capture

$$P'(a) \Leftarrow\ a?(x).(\text{new}(n').(n'!\langle\rangle \mid c!\langle\rangle)) \tag{2.2.5}$$

We can now safely perform the receive and the corresponding substitution, yielding

$$\text{new}(n').(n'!\langle\rangle \mid n!\langle\rangle) \tag{2.2.6}$$

α-equivalency

Obviously we want to say $P(a)$ and $P^{'}(a)$ are equivalent. In general when two terms are the same up to their choice of bound identifiers, we say they are *α-equivalent*, and write $P(a) \equiv_\alpha P^{'}(a)$. Though this can be treated more explicitly, from now on when we perform a substitution $R[\![\overline{V}/\overline{X}]\!]$, we will implicitly pick a term α-equivalent to R where the substitution will not capture terms bound in R. From now on we will intend that a term represents its entire α-equivalency class, and thus will not explicitly specify α-equivalency in the equivalence relation introduced below.

interface
↪ page 4

Notice the use of interfaces to define the process $P(a)$ in (2.2.4). When a process has free channel names occurring in its body, as is the case with a, we will sometimes place those names inside parentheses after the the process to suggest a process' interface. We say that a process Q with interface $Q(\overline{V})$ *exposes* the names $\overline{V}$. The idea is that since these terms are free, when the interface is used as a component in a larger system, we can have other components that expose the same channels. That way, our system can bind those channels at the top level and its components can communicate on them. For example, consider the following system:

exposes

$$\text{new}(b, c).Q_1\langle b\rangle \mid Q_2\langle b, c\rangle$$

Here we are assuming the existence of some process Q_1 exposing b and some process Q_2 exposing b and c. Both process components Q_1, Q_2 both expose a common name b, which the system has bound. When some larger system uses *new* to bind channels that are exposed by a component interface we say that system *instantiates* the interface. Since usage of b in the processes is free, b is not captured when it is bound by the system. Instead, both processes have the bound term in common, and can use it to interact.

instantiate

2.2.2 Contexts and Equivalence

Perhaps the next idea we might have for building our equivalence relation is that equivalent processes should act the same when dropped into any larger system. Let us define more precisely what we mean by 'dropping in' a process.

Definition 2.2.7 (Context) A context $\mathbb{C}$ is given by:

$$\mathbb{C} := \begin{cases} [\,] & \\ \mathcal{C} \mid Q \text{ or } Q \mid \mathcal{C} & \text{for any process } Q, \text{ context } \mathcal{C} \\ \text{new}(n).\mathcal{C} & \text{for any name } n, \text{ context } \mathcal{C}. \end{cases}$$

$\mathbb{C}[Q]$ denotes the result of replacing the placeholder $[\,]$ in the context $\mathbb{C}$ with the process term Q.

Notice that with contexts, we do not pay any attention to whether a name in Q is bound in $\mathbb{C}$. Hence, unlike with substitution, free variables in Q can become bound

in $\mathbb{C}[Q]$. So, for example, channel c in the process $P(c) \Leftarrow c?().R$ can become bound in $\mathbb{C}[P(c)]$ where $\mathbb{C}[\ ]$ is the context

$$\text{new}(c).c!\langle\rangle \mid [\ ]$$

We say that a relation $\sim$ between processes is *contextual* if $P \sim Q$ implies $\mathbb{C}[P] \sim \mathbb{C}[Q]$ for any context $\mathbb{C}$. We are now ready to define our notion of equivalency using contexts.

contextual

Definition 2.2.8 (Structural Equivalence) Structural Equivalence, denoted $\equiv$ is the smallest contextual equivalence relation that satisfies the following axioms:

$$\begin{aligned}
P \mid Q &\equiv Q \mid P && \text{(S-COMP-COMM)}\\
(P \mid Q) \mid R &\equiv P \mid (Q \mid R) && \text{(S-COMP-ASSOC)}\\
P \mid \mathit{stop} &\equiv P && \text{(S-COMP-ID)}\\
\text{new}(c).\mathit{stop} &\equiv \mathit{stop} && \text{(S-REST-ID)}\\
\text{new}(c).\text{new}(d).P &\equiv \text{new}(d).\text{new}(c).P && \text{(S-REST-COMM)}\\
\text{new}(c).(P \mid Q) &\equiv P \mid \text{new}(c).Q, \text{ if } c \notin \text{fi}(P) && \text{(S-REST-COMP)}
\end{aligned}$$

These axioms are simply a set of syntactic rules we use to identify as the same processes that are syntactically different. The first and second state that composition is commutative and associate. Thus, we will omit parentheses around compositions when our meaning is clear. (S-COMP-ID) states that a terminated process can be eliminated from a composition. (S-REST-ID) states that a channel scope restriction operator can be eliminated when its scope is only over a terminated process. (S-REST-COMP) states that scope ordering does not matter, justifying our shorthand $\text{new}(c,d).P$. The last of these axioms is most important – it is the basis for *scope extrusion*, upon which process mobility is based (as we shall demonstrate in Example 2.3.2 below).

scope extrusion
↪ page 8
mobility
↪ page 3

Example 2.2.9 In our discussion of bound identifiers above, we asserted that we can eliminate a scope restriction operation when none of the scoped identifiers occur in its scope. We can now show why this is permissible:

$$\begin{aligned}
&\text{new}(n,m).a?(x_1,x_2).x_1!\langle\rangle \mid \text{rec } x.(a!\langle n,m\rangle \mid x)\\
\equiv\ &\text{new}(n,m).(a?(x_1,x_2).x_1!\langle\rangle \mid \mathit{stop}) \mid \text{rec } x.(a!\langle n,m\rangle \mid x) && \text{(S-COMP-ID)}\\
\equiv\ &a?(x_1,x_2).x_1!\langle\rangle \mid \text{new}(n,m).\mathit{stop} \mid \text{rec } x.(a!\langle n,m\rangle \mid x) && \text{(S-REST-COMP)}\\
\equiv\ &a?(x_1,x_2).x_1!\langle\rangle \mid \mathit{stop} \mid \text{rec } x.(a!\langle n,m\rangle \mid x) && \text{(S-REST-ID)}\\
\equiv\ &a?(x_1,x_2).x_1!\langle\rangle \mid \text{rec } x.(a!\langle n,m\rangle \mid x) && \text{(S-COMP-ID)}
\end{aligned}$$

■

2.3 Reduction Semantics

We are now ready to give the *semantic* properties that a process in our language should possess. By specifying the behavior of processes, we define how computation

proceeds in the π-calculus. The set of rules given below show how a process can internally evolve through a number of computation steps.

TODO: make all index terms lowercase

Definition 2.3.1 (Reduction) The *reduction relation* $\longrightarrow$ is the smallest contextual relation that satisfies the following rules:

$$c!\langle \overline{V} \rangle \mid c?(\overline{X}).R \longrightarrow R[\![\overline{V}/\overline{X}]\!] \quad \text{(R-COMM)}$$

$$\text{rec } p.R \longrightarrow R[\![\text{rec } p.R/p]\!] \quad \text{(R-REP)}$$

$$\text{if } v = v \text{ then } P \text{ else } Q \longrightarrow P \quad \text{(R-EQ)}$$

$$\text{if } v_1 = v_2 \text{ then } P \text{ else } Q \longrightarrow Q \quad (\text{where } v_1 \neq v_2) \quad \text{(R-NEQ)}$$

$$\frac{P \equiv P', P \longrightarrow Q, Q \equiv Q'}{P' \longrightarrow Q'} \quad \text{(R-STRUC)}$$

We use the notation $P \cdots\!\longrightarrow Q$ when an arbitrary number of these rules have been applied in reducing P to Q.

The first of these allows a computation step for the transmission of values over a channel. Note that we only allow the application of (R-COMM) when the substitution $[\![\overline{V}/\overline{X}]\!]$ makes sense. That is, $\overline{V}$ and $\overline{X}$ must have the same arity and in a typed system we'd want to ensure that their types were compatible[1]. (R-EQ) enables a computational step for value-matching. For example, in

arity ↪ page 8

$$c!\langle a \rangle \mid c?(x).\text{if } x = a \text{ then } P \text{ else } stop$$

we apply (R-COMM) to obtain

$$\text{if } a = a \text{ then } P \text{ else } stop$$

from which we can apply (R-EQ) to obtain P. (R-REP) allows us to unravel a recursive expression so that it contains replicas of itself. For example,

$$\text{rec } p.c!\langle\rangle \mid p$$

expands to

$$c!\langle\rangle \mid (\text{rec } p.c!\langle\rangle \mid p)$$

Finally, (R-STRUCT) says that a reduction is defined up to structural equivalence. We give an example of its use below.

scope extrusion ↪ page 8

Example 2.3.2 We will give a demonstration of how scope extrusion is obtained using the rules and axioms of reduction and structural equivalence. Consider the expression

$$d?(x).x!\langle\rangle \mid \text{new}(c).(d!\langle c \rangle \mid c?().stop) \tag{2.3.3}$$

You can see above that c's scope can be extruded by sending over d and that the left side of the term will then use c to communicate with the right side. Now we will

[1] See [Hen07] for a discussion of type systems for the π-calculus.

show that this is enabled by the reduction rules. First, we can use (S-REST-COMP) to bring the restriction to the outside, giving:

$$\text{new}(c).(d?(x).x!\langle\rangle \mid d!\langle c\rangle \mid c?().stop) \tag{2.3.4}$$

Since (2.3.3) is equivalent to (2.3.4), we can apply (R-STRUCT) to deduce that if 2.3.4 reduces to some Q, then (2.3.3) will. To find this Q, we can apply (R-COMM), which we can do inside of the restriction operator thanks to the reduction relation's contextuality, and apply a substitution to (2.3.4), resulting in:

$$\text{new}(c).(c!\langle\rangle \mid c?().stop) \tag{2.3.5}$$

Finally, we can apply (R-COMM) again, so the process simply reduces to *stop*.

Now let $P(d)$ be the right half of our original process, and $\mathbb{C}$ be the remaining context:

$$P(d) \Leftarrow \text{new}(c).(d!\langle c\rangle \mid c?().stop) \tag{2.3.6}$$

$$\mathbb{C} = d?(x).x!\langle\rangle \mid [\,] \tag{2.3.7}$$

Then we have shown that if we drop the component $P(d)$ into the context $\mathbb{C}$ (forming $\mathbb{C}[P\langle d\rangle]$), it will establish a new channel c and extend its scope, using d. This means that components that are 'dropped in' to a larger system can create new channels on the fly and then extrude them to communicate with the rest of the system (provided they share at least one channel to begin with). In fact, this is the very procedure that allows a system to change its communication topology dynamically via scope extrusion. ■

2.4 Action Semantics

Our description of process behaviors so far has been limited to talking about the internal computational steps through which it might evolve. Now, we want to give a more general description of how a process might evolve when placed within the context of a larger system. A process can interact with other processes within such a system by sending or receiving values along channels they share. To describe these abilities, we will use the notion of a *labelled transition system*, or *lts*.

Definition 2.4.1 (Labelled Transition System) A *labelled transition sytem* $\mathcal{L}$ is a tuple $(\mathcal{S}, \mathcal{A})$ where $\mathcal{S}$ is a set of processes and $\mathcal{A}$ is a set of labels called *actions*. Furthermore, for each action α, there is a binary relation:

$$R_\alpha \subseteq \mathcal{S} \times \mathcal{S}$$

To denote that $\langle P, Q\rangle \in R_\alpha$, we will use the notation $P \xrightarrow{\alpha} Q$.

Hence, the transition $P \xrightarrow{\alpha} Q$ indicates that there is an action under which the process P becomes Q. We will refer to Q is the *residual* of P after α.

residual

There are three types of actions that may cause a process to evolve. First, the process might receive a value. That is, a process P is said to be capable of making the transition $P \xrightarrow{c?\overline{X}} Q$, which is to say that P can receive $\overline{X}$ along c to become the residual process Q. The capability of a process to evolve in this way is given by the rule (A-IN) in Figure 2.4.2, which says that for a general process c and input $\overline{V}$

$$c?(\overline{X}).R \xrightarrow{c?\overline{V}} R[\![\overline{V}/\overline{X}]\!]$$

As with (R-COMM), evolution under (A-IN) is only possible when the substitution $[\![\overline{V}/\overline{X}]\!]$ makes sense. For example, a process $c!\langle\overline{V}\rangle \mid P$ will evolve to $stop \mid P \equiv P$ when it exercises its capability for output on c.

The second type of action available is sending. Here we need to be a bit more careful. In the case of receiving, the received names are always bound to new names in $\overline{X}$, so we needn't worry about issues of scope. In sending, however, we might be transmitting either free or bound names, or a mix of them. In the latter case, we need to take account of the fact that the scope of the bound name is being extruded to whatever process receives the name. We denote the set of names that are bound in the send action by $(\overline{B})$, and we say that this set of names is *exported* by the process. Hence, the transition $c!\langle\overline{V}\rangle \xrightarrow{(\overline{B})c!\overline{V}} stop$ expresses the capability to send the values $\overline{V}$ over c, exporting $(\overline{B})$ and resulting in *stop*. For example,

scope extrusion
↪ page 8

bound
↪ page 10

exported

$$\text{new}(d).(c!\langle d\rangle \mid Q) \xrightarrow{(d)c!d} Q$$

The capability of a process to evolve by sending a value is described by (A-OUT) for a general channel c and output values $\overline{V}$:

$$c!\langle\overline{V}\rangle \xrightarrow{c!\overline{V}} stop$$

external action

We will refer to sending and receiving as *external actions*.

internal action

Our third action we call *internal action*. This is caused by some internal evolution in P like those described by (R-COMM) in our Section 2.3. We call actions like sending and receiving external since in order to occur, they need some external process (given in some system of which the process in question is a part) to do the corresponding receiving or sending. However, with internal action there is no external process needed to proceed. We use τ to denote such an internal evolution step. Thus, we say $P \xrightarrow{\tau} Q$ if P is able evolve into Q by performing a reduction step without any external contributions. For example (thanks to (R-COMM)),

$$c!\langle a\rangle \mid c?(x).x!\langle\rangle \xrightarrow{\tau} a!\langle\rangle$$

The capability for τ is defined through the rules (A-COMM), (A-REP), (A-EQ) and (A-NEQ). (A-REP), (A-EQ) and (A-NEQ) simply provide the same internal evolution

capabilities as (R-REP), (R-EQ) and (R-NEQ) do in the reduction semantics. (A-COMM), however, is more subtle then (R-COMM). We describe it below, after a discussion of some of the other rules in action semantics.

We define the action relation under the following rules (with their preconditions listed alongside them):

TODO: make the arrow for evolution size according to the text above it

Definition 2.4.2 (Action) The *action relation* $\longrightarrow$ is the smallest relation between processes that satisfy the following rules:

$$c?(\overline{X}).R \xrightarrow{c?\overline{V}} \mathrm{R}[\![\overline{V}/\overline{X}]\!] \qquad \text{(A-IN)}$$

$$c!\langle\overline{V}\rangle \xrightarrow{c!\overline{V}} \mathit{stop} \qquad \text{(A-OUT)}$$

$$\mathrm{rec}\ x.R \xrightarrow{\tau} R[\![\mathrm{rec}\ x.R/x]\!] \qquad \text{(A-REP)}$$

$$\text{if } v = v \text{ then } P \text{ else } Q \xrightarrow{\tau} P \qquad \text{(A-EQ)}$$

$$\text{if } v_1 = v_2 \text{ then } P \text{ else } Q \xrightarrow{\tau} Q \quad v_1 \neq v_2 \qquad \text{(A-NEQ)}$$

$$\frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \quad bn(\alpha) \cap fn(Q) = \emptyset \qquad \text{(A-COMP)}$$

$$\frac{P \xrightarrow{\alpha} P'}{\mathrm{new}(b).P \xrightarrow{\alpha} \mathrm{new}(b).P'} \quad b \notin n(\alpha) \qquad \text{(A-REST)}$$

$$\frac{P \xrightarrow{(\overline{B})c!\overline{V}} P'}{\mathrm{new}(n).P \xrightarrow{(n,\overline{B})c!\overline{V}} P'} \quad n \neq c, n \in \overline{V} \qquad \text{(A-OPEN)}$$

$$\frac{P \xrightarrow{c?\overline{X}} P',\ Q \xrightarrow{(\overline{B})c!\overline{V}} Q'}{P \mid Q \xrightarrow{\tau} \mathrm{new}(\overline{B}).(P' \mid Q')} \quad (\overline{B}) \cap fn(P) = \emptyset \qquad \text{(A-COMM)}$$

Together, (A-COMP) and (A-REST) provide a capability similar to contextually. That is, given that $P \xrightarrow{\alpha} P'$, (A-COMP) allows us to 'drop in' P alongside a progress Q running in parallel, forming $P \mid Q \xrightarrow{\alpha} P' \mid Q$. Similarly, (A-REST) allows us to drop P into a new scope restriction operator, forming $\mathrm{new}(b).P \xrightarrow{\alpha} \mathrm{new}(b).P'$. The key difference between these rules and contextually is that here we need to be careful about inadvertently capturing bound variables. The action α could export names, so we need to make sure those names don't conflict with names in the process we are trying to infer.

Jim: I wonder if these couldn't be simplified (ie removing A-EQ, A-REP, etc.) by allowing a transition that happens over the reduction semantics? my sense is this approach is avoided because the action relation isn't actually contextual in our above definition (since we need to be more careful about bound variable captures). i wonder if there is a way to gracefully sidestep the issue and avoid all the redundancy...

Thus, in (A-REST) we require that the newly bound variable b does not occur in the names $n(\alpha)$ of the action α. By $n(\alpha)$, we mean the channel name itself, along with all the names $\overline{V}$ in the case of a send action. Suppose we ignored this precondition and tried the following:

$$\frac{c?().\mathit{stop} \xrightarrow{c?} \mathit{stop}}{\mathrm{new}(c).c?().\mathit{stop} \xrightarrow{c?} \mathrm{new}(c).\mathit{stop}} \qquad \text{(A-REST)}$$

contextual ↪ page 13

This is not what we want. We should not allow $\text{new}(c).c?().stop$ to evolve since the scope of c cannot be extruded and thus no other process could ever send along $c!$ Similarly, we should not allow the the introduced restriction to have the same channel name as any of the values transmitted in α. We wouldn't want to end up, for example, with an action relation like

$$\text{new}(b).c!\langle b\rangle \xrightarrow{c!b} \text{new}(b).stop$$

since this would imply that b's scope hasn't been extruded when it fact it has.

In (A-COMP), we need to ensure that none of the bound names of α conflict with the free variables in Q. There are no names (only variables) in a receive expression. The only names that are bound in an action are those of the set $(\overline{B})$ exported by a send action. Thus, the precondition on (A-COMP) simply requires that bound names that are transmitted will be 'fresh' in the composed process Q. For example, consider the following:

$$\frac{\text{new}(b).c!\langle b\rangle \xrightarrow{(b)c!x} stop}{\text{new}(b).c!\langle b\rangle \mid b!\langle a\rangle \xrightarrow{(b)c!x} stop \mid b!\langle a\rangle} \quad \text{(A-COMP)}$$

Above, we have a process sending on b but the scope of b has *not* been extruded (it hasn't been received on c). We don't want to allow b's scope to be 'accidentally' extruded simply by capturing a free b somewhere in Q. Hence, we need to be careful not to 'capture' a channel like b by allowing exported channels to have the same name as free variables outside the original scope of the process.

(A-OPEN) expresses scope extrusion of a name n. Essentially, if we already know that

$$P \xrightarrow{(\overline{B})c!\overline{V}} P'$$

then we can take a name $n \in n(\overline{V})$ and bind it, inferring

$$\text{new}(n).P \xrightarrow{(n,\overline{B})c!\overline{V}} P'$$

The name n is then exported as well. Note that n cannot be c since restricting n (i.e. c) to P would cause a scoping issue that interfered with an outside process' ability to receive on c.

Finally we have (A-COMM), which expresses scope extrusion as well but this time in the context of the internal action that we described with (R-COMM) in the reduction semantics. Suppose we have that $P \xrightarrow{c?\overline{X}} P'$ and $Q \xrightarrow{(\overline{B})c!\overline{V}} Q'$. Then according to (A-COMM), if we compose the two, forming $P \mid Q$, then the simultaneous (compositionally) occurrence of an input action $c?\overline{X}$ and matching output action $(\overline{B})c!\overline{V}$ can be replaced with τ. That is, we can say:

$$P \mid Q \xrightarrow{\tau} \text{new}(\overline{B}).(P' \mid Q'),$$

Note that the exported names $(\overline{B})$ are now scoped to both processes as a result of the original term Q exporting them. Just as in (A-COMP), we need to make sure that none of these exported names are going to capture free terms in P. Hence, we require that $(\overline{B}) \cap fn(P) = \emptyset$.

Example 2.4.3 In Example 2.1.9, we defined a simple process for modeling (recursive) synchronous sends:

$$F_1(c) \Leftarrow \text{ rec } z.(\text{new}(ack).(c!\langle ack\rangle \mid ack?().(R \mid z)))$$

We then went on to show how this process might work as a component in a example system. But suppose we wanted to characterize this process' behavior in general. We could use reduction semantics to described how it it can evolve internally (in this case there isn't much interesting we can do except unwind the recursion), but we'd also want to include a description of how that process behaves externally. If possible, we'd like to do this without having to come up with a system in which to place the process. We don't want our characterization of the process to rely on some auxiliary system. Without defining a particular system for it to work in, we can use action semantics to provide a characterization of how $F_1(c)$ behaves externally. Consider the following inferences:

$$\begin{array}{lll} F_1(c) & \xrightarrow{\tau} & \text{new}(ack).(c!\langle ack\rangle \mid ack?().(R \mid F_1\langle c\rangle)) \quad \text{(A-REP)} \\ & \xrightarrow{(ack)c!ack} & stop \mid ack?().(R \mid F_1\langle c\rangle) \quad \text{(A-OPEN, A-OUT)} \\ & \xrightarrow{(ack?)} & stop \mid (R \mid F_1\langle c\rangle) \quad \text{(A-IN)} \end{array}$$

Using the structural equivalence rule (S-COMP-ID), we know the final step of this sequence of action results in a process equivalent to:

$$R \mid F_1(c)$$

Above, we have shown presence of the important behaviors we expect in $F_1(c)$. That is, we have shown that $F_(c)$ can:

- Recursively spawn a process that can...
- Send ack over c, extruding its scope and resulting in a residual process that can...
- Receive the acknowledgement handshake on ack, resulting in a residual that can...
- Run R and wait for the next recursive iteration to occur (internally).

Thus, we have completely characterized $F_1(c)$'s capabilities to act as a recursive synchronous output process in a larger system — and we have done so without having to say anything about what that larger system actually looks like. ■

2.5 Extended Example: Memory Cells

We said in Chapter 1 that distributed systems usually involve information that is shared between processes executing independently on multiple machines. In the π-calculus, the principle means of sharing information is via message passing. Suppose

that we wanted instead to model a memory cell that stores information in a less fleeting way. Our cell might have channels for getting and setting its value. A process could use the *get* channel to request that the value be sent back over a supplied channel. Passing a new value on the *set* channel would result in the cell changing it's value to the value supplied. The main point of our memory cell is that multiple processes could use the cell's getter channel without the value disappearing. It turns out that we can create this cell just using message passing. This section will explore the creation and use of a memory cell, which serves as a good review and usage example of the concepts introduced in this chapter.

The problem that we have to work around is that receiving a identifier from a channel removes that value from the channel. It cannot be retrieved again by some other process. Thus, we need a way to push it back in for the next time. Consider the following system:

$$\begin{aligned} Cell(get, set) \Leftarrow\ & \text{new}(c).(c!\langle init\rangle \\ & \quad \mid \text{rec } g.(get?(x).c?(y).(x!\langle y\rangle \mid c!\langle y\rangle \mid g)) \\ & \quad \mid \text{rec } s.(set?(x,b).c?(y).(x!\langle b\rangle \mid c!\langle b\rangle \mid s))) \end{aligned} \tag{2.5.1}$$

In *Cell*, we first create a new channel c that will be used to 'store' the value. We then initialize c with the value *init*. The following line has a recursive listener process that, when given a channel x via *get*, gets the value from c and sends it back via x. In parallel, it sends the value back into c so it can be retrieved again. The next line is similar except that two values are supplied via *set* – a new value b and a channel x. The old value is pulled from c and simply discarded (i.e. not used). The new value b is pushed to c this time. It is also sent via x as an acknowledgment that the cell's new value has been set.

Now let use look at how this memory cell might be used. In the following system we assume that we have access to the integers and standard arithmetic operations[2]. The memory cells are initialized with the integer 0. We also give an exposed 'output' channel o.

$$\begin{aligned} MemoryClient(get_1, set_1, o) \Leftarrow\ & (\text{new}(a).(get_1!\langle a\rangle \\ & \quad \mid \text{new}(a_2).(a?(y).set_1!\langle y+1, a_2\rangle \mid a_2?(z).o!\langle z\rangle))) \end{aligned} \tag{2.5.2}$$

The system gets the current value from the channel get_1 and then uses that value to send an incremented value to set_1, receiving acknowledgement on channel a_2. The value received on a_2 is sent to the output channel. Notice that we created new channels for acknowledgement but left the cell's channels free and exposed. Thus, the two components might be put together in the following system (which leaves the output channel free):

$$\text{new}(get_1, set_1).(Cell\langle get_1, set_1\rangle \mid MemoryClient\langle get_1, set_1, o\rangle)$$

[2]See [**?** , miln99]or an example of how the integers and arithmetic operations can be build from π-calculus primitives.

To see why this system produces the behavior we expect, we substitute the definitions for $Cell$ and $MemoryClient$ into the system and observe scope restrictors for a and a_2 can be moved out using (S-REST-COMP):

$$\begin{aligned}\text{new}(get_1, set_1, a, a_2).(\text{new}(c).(&c!\langle init\rangle \\ &\mid \text{rec } g.(get_1?(x).c?(y).(x!\langle y\rangle \mid c!\langle y\rangle \mid g)) \\ &\mid \text{rec } s.(set_1?(x,b).c?(y).(x!\langle b\rangle \mid c!\langle b\rangle \mid s))) \\ &\mid get_1!\langle a\rangle \\ &\mid a?(y).set_1!\langle y+1, a_2\rangle \mid a_2?(z).o!\langle z\rangle)\end{aligned} \tag{2.5.3}$$

We need to first apply (R-REP), since we'd like to work with the get_1 receiver which is inside the *rec* operator.

$$\begin{aligned}\text{new}(get_1, set_1, a, a_2).(\text{new}(c).(&c!\langle init\rangle \\ &\mid get_1?(x).c?(y).(x!\langle y\rangle \mid c!\langle y\rangle \\ &\mid \text{rec } g.(get_1?(x).c?(y).(x!\langle y\rangle \mid c!\langle y\rangle \mid g)) \\ &\mid \text{rec } s.(set_1?(x,b).c?(y).(x!\langle b\rangle \mid c!\langle b\rangle \mid s))) \\ &\mid get_1!\langle a\rangle \\ &\mid a?(y).set_1!\langle y+1, a_2\rangle \mid a_2?(z).o!\langle z\rangle)\end{aligned} \tag{2.5.4}$$

Now we can apply (R-COMM) to the memory cell's getter, performing the substitution $[\![a/x]\!]$ to the term $c?(y).(x!\langle y\rangle \mid c!\langle y\rangle ...$

$$\begin{aligned}\text{new}(get_1, set_1, a, a_2).(\text{new}(c).(&c!\langle init\rangle \\ &\mid c?(y).(a!\langle y\rangle \mid c!\langle y\rangle \\ &\mid \text{rec } g.(get_1?(x).c?(y).(x!\langle y\rangle \mid c!\langle y\rangle \mid g)) \\ &\mid \text{rec } s.(set_1?(x,b).c?(y).(x!\langle b\rangle \mid c!\langle b\rangle \mid s))) \\ &\mid a?(y).set_1!\langle y+1, a_2\rangle \mid a_2?(z).o!\langle z\rangle)\end{aligned} \tag{2.5.5}$$

...and apply it once again to the memory's 'internal' channel c, stripping out the initializer and substituting in the value $init$:

$$\begin{aligned}\text{new}(get_1, set_1, a, a_2).(\text{new}(c).(&\\ &\mid a!\langle init\rangle \mid c!\langle init\rangle \\ &\mid \text{rec } g.(get_1?(x).c?(y).(x!\langle y\rangle \mid c!\langle y\rangle \mid g) \\ &\mid \text{rec } s.(set_1?(x,b).c?(y).(x!\langle b\rangle \mid c!\langle b\rangle \mid s))) \\ &\mid a?(y).set_1!\langle y+1, a_2\rangle \mid a_2?(z).o!\langle z\rangle)\end{aligned} \tag{2.5.6}$$

Which, with one more (R-COMM), finally sends back the $init$ value to the $MemoryClient$ on a. It should now be quite painfully obvious why action semantics would be a better way to analyze the behavior of our memory cell.

It is nice to have a consistent memory cell that many processes can access at once. However, there is a classic issue that often arises in the use of shared memory like

ours. Consider the following modification to our system:

$$\begin{aligned}\text{new}(get_1, set_1).(&Cell\langle get_1, set_1\rangle \\ &\mid MemoryClient_1\langle get_1, set_1, o\rangle \\ &\mid MemoryClient_2\langle get_1, set_1, o\rangle)\end{aligned} \tag{2.5.7}$$

race condition

If we were run these as such, we may find that the output channel was producing some unexpected results. This is because our system contains a *race condition*, which means that the output of a system is dependent on the timing of computations in the system. For example, consider what happens if $MemoryClient_1$ gets the value 0 of the cell first, but before it can set it $MemoryClient_2$ steps in, receives the 0 again and then sets the cell to 1. In that case, $MemoryClient_1$ will have a value for the cell (0) that is no longer current. It will then set the value of the cell to 1 again, where we would expect it to set 2 since the two clients each incremented once.

Race conditions often occur in concurrent programs where access to a location in computer memory is shared between multiple threads. In such systems, as in ours, to order in which threads read and manipulate the shared memory is not predictable or consistent. Hence, concurrent application developers have long been accustomed to using programmatic devices to avoid race conditions.

lock

The usual solution to this type of problem is by implementing a *lock*. A lock ensures that only one client at a time is doing critical operations like incrementing a value. The client asks for the lock, and if it aquires it it proceeds with its critical operations before giving up the lock. If the lock is not available, the client has to wait until another client *releases* it. A lock provides mutually exclusive access to a memory cell. Normally, a client will lock the cell in order to read and/or modify it, releasing the lock when finished. Asynchronous channels map naturally to this type of behavior, as we will see in our $MemoryClient$, which we can modify to use a lock.

Given a channel l, we use the short hand $l?lock.P$ to mean:

$$\text{rec } q.(l?(x).\text{if } x = true \text{ then } (P \mid l!\langle false\rangle) \text{ else } (q \mid l!\langle false\rangle))$$

Above we check the lock channel l and execute P if it is currently set to *true* (meaning the lock is still available). Otherwise we recursively try to get the lock again. Notice that whether the lock is already set to *false*or we are able to acquire it, we need to send*false*to l so that another process can't acquire the lock.

Let's use action semantics to make sure locking behaves as expected. We first note using (A-REP) that the above can internally evolve over τ to

$$\begin{aligned}\xrightarrow{\tau} &l?(x).\text{if } x = true \text{ then } (P \mid l!\langle false\rangle) \text{ else} \\ &(l?lock.P \mid l!\langle false\rangle)\end{aligned} \tag{2.5.8}$$

If $l?(x)$. receives a *true*, then

$$\begin{aligned}\xrightarrow{c?true} &\text{if } true = true \text{ then } (P \mid l!\langle false\rangle) \text{ else} \\ &(l?lock.P \mid l!\langle false\rangle)\end{aligned} \tag{2.5.9}$$

We can now apply (A-EQ), yielding:

$$\xrightarrow{\tau} P \mid l!\langle false \rangle \tag{2.5.10}$$

As expected. Otherwise, if $l?(x).$ receives a *false*, then $l?lock.P$

$$\begin{aligned} \xrightarrow{c?false} & \text{if } false = true \text{ then } (P \mid l!\langle false \rangle) \text{ else} \\ & (l?lock.P \mid l!\langle false \rangle) \end{aligned} \tag{2.5.11}$$

This time using (A-NEQ),

$$\xrightarrow{\tau} l?lock.P \mid l!\langle false \rangle \tag{2.5.12}$$

Which again is what we would expect. Thus, we have verified that the lock works as expected.

TODO: line up all the above equations by putting them in one array and using breaks

Now, unlocking is done via the shorthand $l!unlock$, meaning

$$l?(x).l!\langle true \rangle$$

This term simply discards the current l value and sets it to *true*.

Our $MemoryClient$ thus becomes:

$$\begin{aligned} LMemoryClient(get_1, set_1, o, l) \Leftarrow (\text{new}(a).(l?lock.(& get_1!\langle a \rangle \\ & \mid \text{new}(a_2).(a?(y).set_1!\langle y+1, a_2 \rangle \\ & \mid a_2?(z).(l!unlock \mid o!\langle z \rangle))))) \end{aligned} \tag{2.5.13}$$

Note that the lock l is exposed. It needs to be initialized to *true* by the system that defines it so that the first process trying to get the lock can get it. We will return to the subject of locks in Chapter 4, where they play a surprisingly similar role in the implementation of a synchronous π-calculus.

3
The Synchronous π-Calculus

In Section 2.1, we introduced a version of the π-calculus which we said was made *asynchronous* by its use of non-blocking send operations. Rather than allowing an operation to happen after a sent value has been received by some other process, a sending process simply ends. In Example 2.1.9 we gave a straightforward method of simulating synchronous sending in the asynchronous π-calculus using acknowledgement channels. Also absent in our calculus was the non-deterministic choice operator, also known as the summation operator. We gave a method of simulating this in Example 2.1.7.

We have already had a taste of the expressive power of the synchronous π-calculus in Chapter 1's mobile phone network example. In fact, the original calculus given by Milner, Parrow and Walker was synchronous and included the two operators – summation and synchronous sending – that we omitted in our asynchronous π-calculus.

Our original calculus featured synchronous receiving, but we now make sending synchronous as well, allowing a process to execute after the output has been consumed by some other receiver. Hence, in the synchronous π-calculus, processes can be guarded by both receiving *and* sending.

guarded
↪ page 8

If we have a group of guarded processes joined by the summation operator, only one of those will be processes will be 'chosen' (non-deterministically) to be executed. The rest will simply terminate without having any effect. The power of the synchronous π-calculus comes when we have another group of summed processes, running in parallel, something like:

$$c!\langle\rangle.P_1 + d_1!\langle\rangle.P_2 + d_2?().P_3 \mid c?().R_1 + d_3!\langle\rangle.R_2$$

In the above isolated system, we actually *can* know which of the processes will be chosen to run, despite the non-deterministic nature of the choice operator. Because both sends and receives are guarded, the P_k's and R_k's above can only run when their respective transmission guards complete. Hence, only P_1 and R_1 will be allowed to run since sending on c is the only transmission with a 'matching' reception on the same channel. Now imagine a slightly different system:

$$c!\langle\rangle.P_1 + d!\langle\rangle.P_2 \mid c?().R_1 + d?().R_2$$

Here we have *two* transmissions with matches, on c and on d. Hence, we cannot make a guarantee about which processes are chosen. However, we *can* say that if P_1 executes on the left side, R_1 will be chosen on the right side. We can say the same for

P_2 and R_2. There is a silent, implicit sort of communication that happens between groups of parallel processes when a non-deterministic choice is made among them. This is a powerful feature which comes with the special operators of the synchronous π-calculus.

Now that we have gotten an idea of the synchronous π-calculus' unique properties, we will turn in the next section to its formal syntax and rules of computation.

3.1 The Synchronous π-Calculus

	Action Prefixes	
$\pi :=$	$n!\langle\overline{V}\rangle$	Send
	$n?(\overline{X})$	Receive
	Process terms	
$R :=$	$\sum_i \pi_i.R_i$	Summation
	$R_1 \mid R_2$	Composition
	$\text{new}(n).R$	Restriction
	if $v_1 = v_2$ then R_1 else R_2	Matching
	rec $x.R$	Recursion
	stop	Termination

Figure 3.1.1: *Terms in the synchronous π-calculus*

Note the important difference between Figure 2.1.1 and Figure 3.1.1. First, we have grouped sending and receiving together as *action prefixes*. These prefixes are made available to processes via the summation operator.

Consider the term:

$$\sum_i \pi_i.R_i$$

The notation $\pi_i.R_i$ requires that the action π_i happen before the guarded process R_i can be executed. If R_a is executed in this way, then for all $j \neq a$, the capabilities for both the action π_j and the execution of R_j are lost. In other words, the summation ensures that only one of n guarded processes will be executed, providing a branching behavior in the logic of a term. Which of these branches is picked depends on which action prefix capability is exercised first. As we saw above, we cannot always guarantee which of the action capabilities will be exercised, so we say that the summation operator is non-deterministic.

For the cases that summation is of size 1 or 2, we will use the notation $\pi.R$ and $\pi_1.R_1 + \pi_2.R_2$, respectively. Notice that the former is the equivalent of the process

terms of our asynchronous calculus. To accommodate the new operator, we add to our congruence relation given in Definition 2.2.8 that summation is commutative (S-SUM-COMM) and associative (S-SUM-ASSOC).

A summation of size 0 is what is meant by our *stop* termination process. It behaves just as it did in our asynchronous calculus. We also add to our congruence relation the following trivially true fact, which we call (S-SUM-ID)

$$R + stop \equiv R$$

guarded
↪ page 8

Note also that in our action prefixes, we have made sending a guarded operation, which means that in the term

$$n!\langle \overline{V} \rangle.R$$

R will not execute until some other process receives $\overline{V}$ along n. Receiving is also guarded, as in the asynchronous version.

$$\begin{array}{rcll}
R + Q & \equiv & Q + R & \text{(S-SUM-COMM)} \\
(P + Q) + R & \equiv & P + (Q + R) & \text{(S-SUM-ASSOC)} \\
R + stop & \equiv & R & \text{(S-SUM-ID)} \\
P \mid Q & \equiv & Q \mid P & \text{(S-COMP-COMM)} \\
(P \mid Q) \mid R & \equiv & P \mid (Q \mid R) & \text{(S-COMP-ASSOC)} \\
P \mid stop & \equiv & P & \text{(S-COMP-ID)} \\
\text{new}(c).stop & \equiv & stop & \text{(S-REST-ID)} \\
\text{new}(c).\text{new}(d).P & \equiv & \text{new}(d).\text{new}(c).P & \text{(S-REST-COMM)} \\
\text{new}(c).(P \mid Q) & \equiv & P \mid \text{new}(c).Q, \text{ if } c \notin fi(P) & \text{(S-REST-COMP)}
\end{array}$$

Figure 3.1.2: *Structural equivalence axioms in the synchronous π-calculus*

3.2 Computation in The Synchronous π-Calculus

We are now ready to give a description of the computation behavior. As we might expect, it does not differ hugely from computation in the asynchronous calculus. In the reduction rules, the only changes are to make room for the summation operator. Because only one process gets chosen from a group of summed processes, we need two 'matching' sender and receiver terms each running in parallel. This behavior is described by (R-SYNC). It expresses the commutation step where the processes have transmitted their value. Hence, we perform the appropriate substitution to the receiving process, run the guarded processes and terminate all the other terms in the sum.

$$c!\langle\overline{V}\rangle.P + Q \mid c?(\overline{X}).R + B \longrightarrow P \mid R[\![\overline{V}/\overline{X}]\!] \quad \text{(R-SYNC)}$$

$$\text{rec } x.R \longrightarrow R[\![\text{rec } x.R/x]\!] \quad \text{(R-REP)}$$

$$\text{if } v = v \text{ then } P \text{ else } Q \longrightarrow P \quad \text{(R-EQ)}$$

$$\text{if } v_1 = v_2 \text{ then } P \text{ else } Q \longrightarrow Q \quad (\text{where } v_1 \neq v_2) \quad \text{(R-NEQ)}$$

$$\frac{P \equiv P', P \longrightarrow Q, Q \equiv Q'}{P' \longrightarrow Q'} \quad \text{(R-STRUC)}$$

Figure 3.2.1: *Reduction rules for the synchronous π-calculus*

Example 3.2.2 It is not hard to show that the synchronous π-calculus can model the asynchronous version. To see why, first note that asynchronous sending can be encoded simply by $n!\langle\overline{V}\rangle.stop$. This we will abbreviate with the familiar notation $n!\langle\overline{V}\rangle$. As we noted above, the summation notation allows for a single guarded process. If we limit ourselves to these single summations and limit all sending to be of the form $n!\langle\overline{V}\rangle.stop$, then we have the asynchronous π-calculus. To see why the reduction semantics are compatible, consider the following which shows that (R-COMM) of Definition 2.3.1 can be considered to be a special case of (R-SYNC).

First, we use (S-SUM-ID):

$$c!\langle\overline{V}\rangle.stop + stop \mid c?(\overline{X}).R + stop \equiv c!\langle\overline{V}\rangle.stop \mid c?(\overline{X}).R$$

Using (R-SYNC) enables the action:

$$c!\langle\overline{V}\rangle.stop \mid c?(\overline{X}).R \longrightarrow (stop \mid R)[\![\overline{V}/\overline{X}]\!]$$

We now use (S-COMP-ID):

$$(stop \mid R)[\![\overline{V}/\overline{X}]\!] \equiv R[\![\overline{V}/\overline{X}]\!]$$

Thus, we can now conclude, using (R-STRUC):

$$c!\langle\overline{V}\rangle \mid c?(\overline{X}).R \longrightarrow R[\![\overline{V}/\overline{X}]\!]$$

It should be evident from our presentation of synchronous action rules below that they needn't be shown to be a general version of the asynchronous rules – they are compatible simply by ignoring the extra rule and by using our encoding of asynchronous sending as $n!\langle\overline{V}\rangle.stop$. ■

Neither do the action rules for the synchronous π-calculus differ significantly from those in the asynchronous version. As we would expect, (A-OUT) has been modified to express synchronous sending. It now evolves over output to a process R and not simply to the termination process *stop*:

$$c!\langle\overline{V}\rangle.R \xrightarrow{c!\overline{V}} R$$

We have also added a new rule, (A-SUM). It describes the action of the summation operator much as (A-COMP) describes the action of the composition operator.

$$\sum_{i\in\{1,..,n\}} \frac{P_i \xrightarrow{\pi_i} P_i'}{\pi_i.P_i \xrightarrow{\pi_i} P_i'}$$

(A-SUM) expresses the fact that, when a single guarded process can evolve in one step to a process P_i' due to an action π_i, then it can also do so as part of a summation. Notice that unlike (A-COMP), (A-SUM) does not need to be careful about capturing names. Since no other processes are run by the summation, we needn't worry about causing a naming issue when we possibly export terms by running the action α.

$$c?(\overline{X}).R \xrightarrow{c?\overline{V}} R[\![\overline{V}/\overline{X}]\!] \quad \text{(A-IN)}$$

$$c!\langle\overline{V}\rangle.R \xrightarrow{c!\overline{V}} R \quad \text{(A-OUT)}$$

$$\text{rec } x.R \xrightarrow{\tau} R[\![\text{rec } x.R/x]\!] \quad \text{(A-REP)}$$

$$\text{if } v = v \text{ then } P \text{ else } Q \xrightarrow{\tau} P \quad \text{(A-EQ)}$$

$$\text{if } v_1 = v_2 \text{ then } P \text{ else } Q \xrightarrow{\tau} Q \qquad v_1 \neq v_2 \quad \text{(A-NEQ)}$$

$$\sum_{i\in\{1,..,n\}} \frac{P_i \xrightarrow{\pi_i} P_i'}{\pi_i.P_i \xrightarrow{\pi_i} P_i'} \quad \text{(A-SUM)}$$

$$\frac{P \xrightarrow{\pi} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad bn(\alpha) \cap fn(Q) = \emptyset \quad \text{(A-COMP)}$$

$$\frac{P \xrightarrow{\alpha} P'}{\text{new}(b).P \xrightarrow{\alpha} \text{new}(b).P'} \qquad b \notin n(\alpha) \quad \text{(A-REST)}$$

$$\frac{P \xrightarrow{(\overline{B})c!\overline{V}} P'}{\text{new}(n).P \xrightarrow{(n,\overline{B})c!\overline{V}} P'} \qquad n \neq c, n \in \overline{V} \quad \text{(A-OPEN)}$$

$$\frac{P \xrightarrow{c?\overline{X}} P',\ Q \xrightarrow{(\overline{B})c!\overline{V}} Q'}{P \mid Q \xrightarrow{\tau} \text{new}(\overline{B}).(P' \mid Q')} \qquad (\overline{B}) \cap fn(P) = \emptyset \quad \text{(A-COMM)}$$

Figure 3.2.3: *Action rules for the synchronous π-calculus*

3.3 Extended Example: Leader Elections

Leader elections, a classic problem in distributed systems, are a good example of the power of the synchronous π-calculus. A leader election is a system where a group of processes, each with a unique identifier (via integers, perhaps), must agree on the selection of a 'leader' process. The processes vote on a process to be their leader by sending an integer-valued 'vote' v_i on a given output channel o. Ideally, we want

each of the processes to operate using the same 'program', without any preference or priority hard-coded into that program.

One means of specifying when two processes' programs are the same is by the concept of *symmetry*. We say that two processes are symmetric if they are equivalent under structural equivalence and a systematic renaming of their identifiers. To better understand what is meant by 'a systematic renaming', assume that each process p_i, channel name c_i, and vote v_i has a unique identifier $i \in \{1, .., n\}$. Now suppose we have an isomorphism σ, given by following recursive definition, that maps these identifiers to other identifiers in the space $\{1, ..., n\}$.

symmetry

Action Prefixes

$$\begin{aligned} \sigma(c_i!v_i) &= c_{\sigma(i)}!v_{\sigma(i)} \\ \sigma(c_i?v_i) &= c_{\sigma(i)}?v_{\sigma(i)} \end{aligned}$$

Processes

$$\begin{aligned} \sigma(\sum_{i\in\{1,..,n\}} \pi_i.R_i) &= \sum_{i\in\{1,..,n\}} \sigma(\pi_i).\sigma(R_i) \\ \sigma(R_1 \mid R_2) &= \sigma(R_1) \mid \sigma(R_2) \\ \sigma(\text{new}(n).R) &= \text{new}(n).\sigma(R) \\ \sigma(\text{if } v_i = v_j \text{ then } R_1 \text{ else } R_2) &= \text{if } v_{\sigma(i)} = v_{\sigma(j)} \text{ then } \sigma(R_1) \text{ else } \sigma(R_2) \\ \sigma(\text{rec } x.R) &= \text{rec } x.\sigma(R) \\ \sigma(stop) &= stop \end{aligned}$$

Figure 3.3.1: *Rules for applying σ*

Using these rules, we have a systematic function σ for renaming identifiers in a process. When a system of processes running in parallel are all symmetric to one another, we will say that the system is symmetric. For example, consider the following symmetric system:

$$P_0 \mid P_1 \Leftarrow c_0!\langle\rangle.o!\langle 0\rangle + c_1?().o!\langle 1\rangle \mid c_1!\langle\rangle.o!\langle 1\rangle + c_0?().o!\langle 0\rangle \tag{3.3.2}$$

Here the isomorphism σ operates in the space $\{0, 1\}$, mapping 1 to 0 and 0 to 1. Equivalently, it takes an identifier i to $(i + 1) mod 2$. The output channel o is special so we extend σ to always map o to itself. Hence $P_0 = \sigma(P_1)$ and we say that P_0 is symmetric to P_1.

Now that we have shown that (3.3.2) is a system of processes running 'the same program', we need to show that it actually solves the leader election. There are two possibilities: either P_0 notifies P_1 on c_0 first, or P_1 notifies P_0 on c_1 first. Applying the reduction rule (R-SYNC) to (3.3.2), the first possibility gives

$$o!\langle 0\rangle \mid o!\langle 0\rangle$$

Here we see that both processes will agree in this "election" — that is, they output the same value. Note that no substitutions were necessary since $c!\langle\rangle$ is simply a handshake

handshake
↪ page 8

signal. Note also that these resulting processes are *not* symmetric: applying the isomorphism σ to P_0 would yield $o!\langle 1\rangle$.

If, on the other hand, P_1 notifies first, then again we apply (R-SYNC) to get

$$o!\langle 1\rangle \mid o!\langle 1\rangle$$

Again, a leader has been elected. Hence, we have given a term that successfully solves the leader election problem for symmetric systems in the special case of a two processes. We will discuss more general leader elections in the next chapter.

4

Synchronicity and Distributed Systems

In the preceding chapters, we presented both the synchronous and asynchronous variants of the π-calculus and explored some of the systems they can model. We also demonstrated in Example 3.2.2 that the asynchronous π-calculus can be fully encoded in the synchronous version. This begs the question: can we represent the expressive communication power of the synchronous π-calculus using only our asynchronous π-calculus? That is, using the simulations given in Examples 2.1.9 and 2.1.7 (or perhaps some similar but more complicated approach) can we fully capture the implicit 'communication' (see the introduction to Chapter 3) between non-deterministically chosen processes discussed above? In this chapter, we will explore the surprising complexity of this question and some of its implications in implementing the synchronous π-calculus on distributed systems.

4.1 Separation Results

When trying to compare the expressive power of different calculi, one approach is to provide explicit encodings from one language to another as we did in Example 3.2.2. We say that we are trying to encode a *source language* into the terms of a *target language*. Thus, having an encoding gives us a sort of 'compiler' from terms in the source language to terms written in the target language. If we succeed in finding an encoding, we have shown that the target language is at least as expressive as the source. We used this idea in Example 3.2.2 to demonstrate that the synchronous calculus is at least as expressive as the asynchronous calculus by giving an encoding from the asynchronous to the synchronous. We use the notation

source language

target language

$$[\![P]\!] \stackrel{\text{def}}{=} Q$$

to mean that P in the source language is encoded by Q in the target language.

To prove a separation result between languages, it is enough to show that there are problems that are solvable in the source language that are not solvable in the target language. In Section 3.3, we found that a special case of the leader election problem for symmetric systems was solvable in the synchronous π-calculus. In fact, Palamidessi [Pal03] proves that the synchronous π-calculus is capable of solving the leader election problem in general. In the same paper, she show that the asynchronous π-calculus is *not* capable of solving the leader election problem in symmetric systems.

The latter result stems directly from the lack of the choice operator and the requirement that the election system be symmetric: without guarded choice summation,

symmetric processes have no way to pick a leader non-deterministically in general, without potentially disagreeing with one another. It is only through the implicit communication underlying the choice operator that synchronous processes are able to agree on a leader.

We will not give the full argument here, but we will try to give a sketch of its important properties and will also look at an illustrative example in Section 4.2. Palamidessi's proof lies on an important property enjoyed by only the *asynchronous* π-calculus: *confluence*, which comes from lemma 4.1 from Palamidessi's proof.

> **Lemma 4.1** *Let P be a process of the asynchronous π-calculus. Assume that P can make two transitions $P \xrightarrow{\alpha_s} Q$ and $P \xrightarrow{\alpha_r} Q'$, where α_s is a send action while α_r is a receive action. Then there exists a process R such that $Q \xrightarrow{\alpha_r} R$ and $Q' \xrightarrow{\alpha_s} R$.*

Our processes may start out symmetrical, but as we showed in Section 3.3 in the synchronous calculus the processes must collectively break the original symmetry of the system in order to elect a leader. The issue with the asynchronous calculus is that if a process has two options available to it (say to vote or wait for the other process to vote first) and it chooses one, confluence guarantees that we can still make the action representing the other choice, bringing us to a state that is the same as if we had performed the actions in the opposite order. Consider trying to encode the symmetric election system from Section 3.3:

$$P_0 \mid P_1 \Leftarrow c_0!\langle\rangle.o!\langle 0\rangle + c_1?().o!\langle 1\rangle \mid c_1!\langle\rangle.o!\langle 1\rangle + c_0?().o!\langle 0\rangle \tag{4.1.1}$$

Whatever our encoding looks like, $[\![P_0]\!]$ will have a choice between voting 0 our waiting - call these α_s and α_r respectively. Suppose it chooses α_s. Then $[\![P_1]\!]$ must make the opposite choice $[\![\alpha_r]\!]$ in order for them to agree on a leader. But by confluence $[\![P_0]\!]$ still has the capability for α_r and $[\![P_1]\!]$ the capability for $[\![\alpha_s]\!]$. After these actions, $[\![P_0 \mid P_1]\!]$ is no different than if $[\![P_0]\!]$ had made the opposite choice first! Hence, no matter what the processes do, they will not be able to break their initial symmetry and are unable to elect a leader.

Using her separation results, Palamidessi goes on to give a useful set of requirements that formally separate the two calculi. The first of those requirement is *uniformity*, which means that:

uniformity

TODO: better labeling of these proerties?

$$[\![\sigma(P)]\!] = \sigma([\![P]\!]) \tag{4.1.2}$$

$$[\![P \mid Q]\!] = [\![P]\!] \mid [\![Q]\!] \tag{4.1.3}$$

Rule (4.1.2) simply states that an arbitrary renaming function σ is not violated in the process of the encoding. That is, if we σ-rename a process P and then encode it we get the same result as if we encode it and then σ-rename it. Because our encoding is required to preserve σ, a symmetric term in the source language will still be symmetric in the target language.

Rule (4.1.3) is related to the requirements of a distributed system. Parallel processes really should just map to parallel processes, with no top level 'manager' process

or the like to aid the encoding. That is, we wouldn't want to encode $P \mid Q$ to something like

$$P \mid Manager \mid Q$$

Hence, in uniformity, Palamidessi has built symmetry and distributivity into a general requirement on the encoding.

reasonability

The other requirement is on *reasonability* and is on the target language's semantics. Reasonability to Palamidessi means that the language can distinguish between two processes when their actions are different on a certain given channel. This requirement essentially builds the requirements of the leader election problem into a criteria for the language. Thus, reasonability of a target language means the capability of semantically distinguishing an electoral system, where all actions on the output channel to be the same, from a non-electoral system, where actions on the output channel differ or never happen at all.

TODO: come up with an implement a short hand for the calculi: SPI and API?

Given these results, it should come as no surprise that the implementation of synchronous calculi on distributed systems is a thorny issue. On the one hand, its additional operators allow us to model many problems more naturally and easily. On the other, communication in a distributed system is asynchronous in nature and any synchronous communication must implemented as a layer on top of an asynchronous base. How can we reconcile the requirements of implementing a distributed system with Palamidessi's result, which indicates that there are important problems that cannot be solved without the full generality of the synchronous calculus? Is such a calculus even implementable on distributed systems? We will look below at two encodings that attempt to do so in the proceeding sections. Both encodings based on those given by by Uwe Nestmann in [Nes00].

4.2 Encoding Choice

Our first encoding limits the source language a bit: we do not allow processes in a given summation to be a mix of terms with receive prefixes and send prefixes. A group of summed terms must either be all 'receivers' or all 'senders'. We will call this limitation on summation *separated choice*.

With this limitation in place, we can provide a good encoding. The basic idea for this encoding is similar to that of Section 2.5, where we used a lock to protect a memory cell from multiple clients trying to use the cell at once. Here again, we want to provide mutually exclusive access to something — only instead of the ability to access to the cell we are providing the chosen process the ability to execute. The chosen process is the only one that gets run, so once the process runs the lock never become unlocked. In order to proceed, a receiver term in one summation depends on a sender term in another summation proceeding as well. Thus we need *two* locks: acquiring one ensures that a receiving process has been chosen in one summation, and acquiring the other lock ensures that the corresponding sending process in the other summation is also chosen to run.

To model synchronous sending, we use an acknowledgement channel *ack* in send actions as in Example 2.1.9. On the other side of a transmission, our receive action

uses our 'double lock' to implement choice and is also responsible for sending the correct acknowledgement depending on the results of trying to get that lock. Note that we instantiate one shared lock for each summation, as specified by the encoding of summation:

$$[\![\sum_{i\in\{1,..,n\}} \pi_i.R_i]\!] \stackrel{\text{def}}{=} \text{new}(l).(l!\langle true\rangle \mid \prod_{i\in\{1,..,n\}} [\![\pi_i.R_i]\!]_l)$$

The notation $\prod$ means composition. We have parameterized the encoding $[\![\pi_i.R_i]\!]$ with the created lock channel l so that all the terms in a sum have access to the same lock. Thus, a group of summed senders will have one lock, which we will call a *remote* lock r, and a group of summed receivers will have another lock, which we will call a *local* lock l.

Now we give the encoding for a sending process:

$$\begin{aligned}[\![c!\langle\overline{V}\rangle.P]\!]_r \stackrel{\text{def}}{=} &\text{new}(ack).(c!\langle r, ack, \overline{V}\rangle \mid ack?(x).\text{if } x = true \text{ then } [\![P]\!] \\ &\text{else (if } x = retry \text{ then } c!\langle r, ack, \overline{V}\rangle \text{ else } stop))\end{aligned}$$

This encoding simply sends on c the value, lock, and acknowledgement channel and then waits to receive the acknowledgement. If it receives *true*it runs the encoding of P and if it receives *retry*then it resends the input, which we will explain below. Otherwise it has received *false*and it terminates accordingly.

Our receiver process encoding is simply a recursive listener that tries the lock when it get input on c:

$$[\![c?(\overline{X}).P]\!]_l \stackrel{\text{def}}{=} \text{rec } q.c?(r, ack, \overline{X}).(l, r)?d - lock.[\![P]\!]$$

We use the notation $(l, r)?d - lock.P$ as shorthand for our double lock. Note that the recursive variable q defined above is actually called in the body of the double lock. So what does this double lock look like? Without further ado:

$$\begin{aligned}&l?(x).(\text{if } x = true \\ &\quad \text{then } r?(y).(\text{if } y = true \\ &\quad\quad \text{then } l!\langle false\rangle \mid r!\langle false\rangle \mid ack!\langle true\rangle \mid P \\ &\quad\quad \text{else } l!\langle true\rangle \mid r!\langle false\rangle \mid ack!\langle false\rangle \mid q) \\ &\quad \text{else } l!\langle false\rangle \mid ack!\langle retry\rangle)\end{aligned} \tag{4.2.1}$$

Let's go through the cases to see how this works. First we check the local lock, and if it's available, we check the remote lock (note that this order is useful in the real-world case where a remote lock is more expensive to query than a local one). If both are available, first we send *false*to both locks to make sure no processes in either summation can acquire the lock. Then, we run P and send *true*to *ack* so the matching sender in the other summation will run.

If the local lock is acquired but the remote lock isn't, we make sure that the local lock is still available by sending $l!\langle true\rangle$ and that the remote lock is *not* made available by sending $l!\langle false\rangle$. We also need a recursive call to q to ensure we continue to poll

the lock ourselves. The fact that the remote lock is not available means another sender has already run (i.e. it was sent *true*on its acknowledgement channel by some receiver) so we need to make sure *this* sender is sent *false*on its acknowledgement channel.

Finally, if we fail to acquire a local lock, we send *false*to the lock to ensure no one else gets it either, and then we send the acknowledgement channel the special message *retry*so that sender still has a chance to run, since the remote lock might still be available.

Though this clever double lock successfully encodes our source language restricted to separated choice, it does not effectively encode the full synchronous π-calculus. For example, our two-party electoral system breaks the check and balance system. Here is that system again:

$$P_0 \mid P_1 \Leftarrow c_0!\langle\rangle.o!\langle 0\rangle + c_1?().o!\langle 1\rangle \mid c_1!\langle\rangle.o!\langle 1\rangle + c_0?().o!\langle 0\rangle \tag{4.2.2}$$

Notice that the encoding of the senders will both send, on c_0 and c_1 respectively. Now suppose that both receivers are running and have acquired their local lock. Since one's local lock is the other's remote lock, both will be trapped in a circular wait, with no hope of exiting the deadlock. As per Palamidessi's results, neither process can break out of its symmetry and thus neither will ever have a chance to vote.

4.3 A 'Bakery' Algorithm

We showed in the last section that the asynchronous π-calculus is fully capable of encoding the summation and synchronous send operators when limited to separated choice. However, to truly encode the synchronous π-calculus, we need to derive a way to break out of symmetry.

In fact, this is not a new problem in computer science and many solutions have been proposed, though all of them violate Palamidessi's encoding criteria to some extent. Palamidessi herself gives a probabilistic encoding [PH01] which does not break uniformity but is not reasonable since, though it succeeds with probability 1, it cannot distinguish between the freak case divergence and a successful election. Another route is to violate symmetry by comparing process ids. In this case, processes can make asymmetrical decisions by simply comparing their ids and deciding accordingly.

uniformity ↪ page 32
reasonability ↪ page 33

Better yet is a symmetric — though still not uniform — variation on Lamport's classic Bakery algorithm by given in [Nes00]. Instead of statically encoding the asymmetry in process ids, Lamport's algorithm has processes dynamically obtain a number from a number server much as a patron might when entering the queue at a bakery counter. Because this asymmetry is dynamically determined, the real-world implications on fairness are acceptable but the processes can always elect a leader. Requiring a number server also breaks uniformity, since it requires a top-level number server process, but again in a real-world distributed system the cost of running a single number server to provide ids to processes is not serious. Below, we assume the existence of a top level number server which is really just a shared channel d initialized with some integer value. Our encoding of the summation operator simply grabs the

TODO: citation for lamport

current number from the server and then sends the next integer value for the next summation operator. Other than that it is the same as our last encoding:

$$[\![\sum_{i\in\{1,..,n\}} \pi_i.R_i]\!]^d \stackrel{\text{def}}{=} d?(n).(d!\langle n+1\rangle \mid \text{new}(l).(l!\langle \mathit{true}\rangle \mid \prod_{i\in\{1,..,n\}} [\![\pi_i.R_i]\!]^d_{n,l}))$$

Notice that the encoding passes on the number *and* number server to the summed terms. We pass on the latter because some component process of R_i might contain a summation term itself, in which case it will need access to d.

The only change in the encoding of the send operator is that we now send n on c as well:

$$\begin{aligned}[\![c!\langle\overline{V}\rangle.P]\!]^d_{n,l} \stackrel{\text{def}}{=} &\text{new}(\mathit{ack}).(c!\langle n, l, \mathit{ack}, \overline{V}\rangle \mid \mathit{ack}?(x).\text{if } x = \mathit{true} \text{ then } [\![P]\!]^d \\ &\text{else (if } x = \mathit{retry} \text{ then } c!\langle n, l, \mathit{ack}, \overline{V}\rangle \text{ else } \mathit{stop}))\end{aligned}$$

The basic new idea of Nestmann's application of the Bakery algorithm is that our leader election deadlock could have been avoided if the receiver processes had checked their locks in a different order. Since both receiver processes checked their local locks and then the remote lock, each held the lock the other was hoping to acquire. If each process had a number, they could simply compare numbers and have the one with the lower number check its *local* lock first. The process with the higher number would check its *remote* lock first. If two processes have the same number, then that means that they are of the same summation and that *neither* should run (without the help of some matching transmission in another summation). Here we give the encoding for the receiver:

$$\begin{aligned}[\![c?(\overline{X}).P]\!]^d_{n,l} \stackrel{\text{def}}{=} &\text{rec } q.(c?(m, r, \mathit{ack}, \overline{X}).(\\ &\text{if } n = m \text{ then } (\mathit{ack}!\langle \mathit{retry}\rangle \mid q) \text{ else } (\\ &\text{if } n < m \text{ then } (l, r)?d - \mathit{lock}.[\![P]\!]^d \text{ else } (r, l)?d - \mathit{lock}.[\![P]\!]^d)))\end{aligned}$$

The double lock (l, r) is just as before, but the other double lock is slightly different. It expresses that r is checked before l. We use $(r, l)?d - \mathit{lock}.P$ to denote:

$$\begin{aligned}&r?(x).(\text{if } x = \mathit{true} \\ &\quad\text{then } l?(y).(\text{if } y = \mathit{true} \\ &\qquad\text{then } l!\langle \mathit{false}\rangle \mid r!\langle \mathit{false}\rangle \mid \mathit{ack}!\langle \mathit{true}\rangle \mid P \\ &\qquad\text{else } l!\langle \mathit{false}\rangle \mid r!\langle \mathit{true}\rangle \mid \mathit{ack}!\langle \mathit{retry}\rangle) \\ &\quad\text{else } r!\langle \mathit{false}\rangle \mid \mathit{ack}!\langle \mathit{false}\rangle \mid q)\end{aligned} \tag{4.3.1}$$

Because the order of the conditions is reversed, we've had to reverse the order of the else statements too. The case of successfully getting both the remote and local locks is the same as in (l,r). If the process only gets the remote lock it tells the remote sender to keep trying and continues to make the remote lock available. If on the other hand it fails to get the remote lock, it restarts the receiver and makes sure to tell the remote sender its failed.

Note that Nestmann's 'implementation' of the synchronous π-calculus does not pay attention to the efficiency questions that would be crucial in a real system. Perhaps its worst property is that in the $n = m$ case there is a possible divergence. If two terms (a sender and a receiver) in the same summation are communicating with one another, our $n = m$ case has them both retry. Thus, they might continue trying forever. Though this represents potentially divergent behavior, it is important to note that this is not the same as a live-lock: the processes will run forever without progressing, but only because they are both in a summation that is unable to progress anyway. However, it still represents a violation of reasonability since it is possible (though unlikely) that the two might continue communicating forever, never allowing for the computation step of sending the leader votes to the output channel o[1]. Hence, while Nestmann's encoding violates both of Palamidessi's criteria to some extent, it nevertheless provides the full behavior of the synchronous in a way that could easily be implemented on a distributed system using only asynchronous primitives.

[1]Though we will not show it here, Nestmann does goes on in [Nes00] to give the same encoding, using a slightly enhanced target language, that does not have this divergence problem.

5
Conclusions

In Chapter 4 we said with Palamidessi that no uniform encoding of the synchronous π-calculus into the asynchronous π-calculuspreserving reasonable semantics exists. This is a very strong result. Weakening either of the requirements that Palamidessi assumes seems like it might produce an encoding not rigorous enough to study. For this reason, the fully expressive synchronous π-calculus seems like a better candidate for formal study than the asynchronous π-calculus.

However, we need still consider the implementation of a synchronous π-calculus. On distributed systems, we have only asynchronous sending available to us. Hence, it also seems useful to study the asynchronous π-calculus since it models these systems more accurately than a synchronous model. For the study of distributed systems, rather than showing the asynchronous to be not worth our time, Palamidessi's separation result raises the question of whether we should be considering *synchronous* calculi.

Ideally, we'd like the best of both worlds. The expressiveness of the synchronous π-calculus allows us to solve a large class of problems much more easily and clearly. We saw just how useful the synchronous π-calculus can be for expressing distributed systems in our extended mobile phone network example in the Chapter 1. We could have modeled this system in the asynchronous π-calculus, but it would have involved a convoluted mess of acknowledgement channels just to express the necessary ordering of events in the system. Hence, the last chapter looked at some of the more implementation-minded encodings of the synchronous π-calculus in the asynchronous π-calculus, and to what extent we need to relax Palamidessi's requirements to allow these encodings.

The creators of Pict, the Join-calculus, and other implementations based on the π-calculus all decided to have their primitives support only asynchronous communication, while synchronous communication is made available overtop of this via a library or higher-level language. This these greatly simplifies implementation, resulting in a cleaner, more efficient core language. The summation operator in particular is difficult and expensive to fully simulate. In the implementation of Pict, for example, David Turner notes [Tur96] that "the additional costs imposed by summation are unacceptable.". Turner goes on to say that *essential* uses of summation are infrequent in practice.

Speaking in an interview on developing the π-calculus, Robin Milner notes [Ber03]:

> That was to me the challenge: picking communication primitives which could be understood at a reasonably high level as well as in the way these

> systems are implemented at a very low level...There's a subtle change from the Turing-like question of what are the fundamental, smallest sets of primitives that you can find to understand computation...as we move towards mobility... we are in a terrific tension between (a) finding a small set of primitives and (b) modeling the real world accurately.

This tension is quite evident in the efforts of process algebraists to find the 'right' calculus for modeling distributed systems. While the synchronous π-calculus is more elegant and fundamental, actual implementations must commit to asynchronous communication as their primitives. Hence, which we choose as a model depends in part on our goals. In any case, it is evident that by limiting ourselves to smaller calculi, many useful new concepts and structures arise in order to solve the problems posed by asynchronous communication. While these structures might not belong in the 'smallest set of primitives', they are useful for bringing the power of the π-calculus to a model that more closely resembles the implementation of distributed systems.

References

[Ber03] Martin Berger. An interview with robin milner, September 2003.

[Cas02] I. Castellani, editor. *Models of Distribution and Mobility: State of the Art*. MIKADO Global Computer Project, August 2002.

[FG00] C. Fournet and G. Gonthier. The join calculus: a language for distributed mobile programming, 2000.

[Hen07] Matthew Hennessy. *A Distributed Pi-Calculus*. Cambridge University Press, 2007.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[HP00] Oltea Mihaela Herescu and Catuscia Palamidessi. Probabilistic asynchronous π-calculus. *Lecture Notes in Computer Science*, 1784:146–??, 2000.

[HT92] Kohei Honda and Mario Tokoro. On asynchronous communication semantics. In Mario Tokoro, Oscar Nierstrasz, and Peter Wegner, editors, *Proceedings of the ECOOP'91 Workshop on Object-Based Concurrent Computing*, volume 612, pages 21–51. Springer-Verlag, 1992.

[Lam74] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

[Lum99] Markus Lumpe. *A -calculus based approach to software composition*. PhD thesis, University of Bern, , Institute of Computer Science and Applied Mathematics, 1999.

[Mil82] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[Mil92] Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the π-calclulus*. Cambridge University Press, 1999.

[Nes00] Uwe Nestmann. What is a "good" encoding of guarded choice? *Inf. Comput.*, 156(1-2):287–319, 2000.

[Pal03] Catuscia Palamidessi. Comparing the expressive power of the synchronous and the asynchronous π-calculus. *Mathematical Structures in Computer Science*, 13(5):685–719, 2003.

[PH01] C. Palamidessi and O. Herescu. A randomized encoding of the -calculus with mixed choice, 2001.

[SW01] Davide Sangiorgi and David Walker. *The π-calculus:A Theory of Mobile Processes*. Cambridge University Press, 2001.

[SWP99] Peter Sewell, Paweł T. Wojciechowski, and Benjamin C. Pierce. Location-independent communication for mobile agents: A two-level architecture. *Lecture Notes in Computer Science*, 1686:1+, 1999.

[Tur96] David N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, 1996.

[WS99] Pawel Wojciechowski and Peter Sewell. Nomadic pict: Language and infrastructure design for mobile agents. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA, 1999.