The Pennsylvania State University
The Graduate School
Department of Computer Science and Engineering

THE PROBABILISTIC ASYNCHRONOUS PI-CALCULUS

A Thesis in
Computer Science and Engineering
by
Oltea Mihaela Herescu

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

December 2002

We approve the thesis of Oltea Mihaela Herescu.

Date of Signature

Catuscia Palamidessi

Professor of Computer Science and Engineering

Thesis Adviser

Chair of Committee

John J. Hannan

Associate Professor of Computer Science and Engineering

Mahmut Kandemir

Assistant Professor of Computer Science and Engineering

Stephen Simpson

Professor of Mathematics

Raj Acharya

Professor of Computer Science and Engineering

Head of the Department of Computer Science and Engineering

Abstract

In this dissertation, we consider a distributed implementation of the π-calculus, more precisely, the version of the π-calculus with mixed choice. To this end, we present the probabilistic asynchronous π-calculus, which is an extension of the asynchronous π-calculus enhanced with a notion of random choice. We define an operational semantics which distinguishes between probabilistic choice, made internally by the process, and nondeterministic choice, made externally by an adversary scheduler. This distinction will allow us to reason about the probabilistic correctness of algorithms under certain schedulers. We show that in this language we can solve the electoral problem, which was proved not possible in the asynchronous π-calculus.

We propose a randomized distributed encoding of the π-calculus, using the probabilistic asynchronous π-calculus, and we show that our solution is correct with probability 1 under any proper adversary with respect to a notion of testing semantics.

Finally, in order to prove that the probabilistic asynchronous π-calculus is a sensible paradigm for the specification of distributed algorithms, we define a distributed implementation of the synchronization-closed probabilistic asynchronous π-calculus in the Java language.

Table of Contents

List of Figures

Acknowledgments

It is difficult to overstate my gratitude to my supervisor, Dr. Catuscia Palamidessi, for all the guidance and support she has shown me. Throughout my graduate studies at Penn State, she has been a constant source of confidence and inspiration. I would have been lost without her.

A special thanks to Vicki Keller and the rest of the staff in the CSE Department at Penn State for always being there and assisting me in many different ways.

I am indebted to my friends and colleagues at Penn State for providing a stimulating and fun environment. I am especially grateful to Liliana Florea, Elaine Pimentel, Jeremie Wajs, Adam Fischbach and Matt Davis.

A big thank you to my husband Florin Herescu for his continuous encouragement, support and incredible patience.

Last, but not least, I wish to thank my parents, Anda and Viorel Delarascruci, for coping with my desire of living abroad and for their unconditional love. This thesis is dedicated to them.

Chapter 1

Introduction

The theory of concurrency provides a formal basis for the specification of concurrent systems, and includes formalisms for modelling, expressing properties, validation and verification of such systems.

The history of the theory of concurrency started more than thirty years ago. Various models of communicating systems, such as Petri nets [44], CSP [13], ACP [2], CCS [24, 25], the π-calculus ([27]) have been proposed since then.

Robin Milner's invention of the Calculus of Communicating Systems (CCS) was a cornerstone in the history of theory of concurrency. CCS deals with interactive systems which are not mobile and was designed to help understanding formal tools in concurrency by using the least number of concepts. The goal of the designer of CCS was to create a calculus that plays an analogous role for concurrency as the λ-calculus plays for sequential computing.

The development of mobile computing brought new challenges into the understanding of communicating systems. The π-calculus was developed in the late 1980s with the goal of analyzing the behavior of mobile systems, i.e. systems whose communication topology can change dynamically. The π-calculus is a calculus of mobile processes introduced by Robin Milner, Joachim Parrow and David Walker. The π-calculus has its roots in the process algebra CCS, namely CCS with mobility, introduced by Uffe Engberg and Mogens Nielsen [9]. The capacity of dynamic reconfiguration of the network of processes gives π-calculus a much greater expressiveness than CCS. The transfer of a communication link between two processes represents the basic computational step in the π-calculus.

The asynchronous π-calculus, as proposed by Boudol [5] and, independently, by Honda and Tokoro [14], is a subset of the π-calculus in which communication is asynchronous in the sense that output processes are not allowed to carry continuations. Asynchronous communication models the situation when the action of sending a message and the action of receiving it do not have to occur at the same time, whereas synchronous

communication corresponds to the simultaneous exchange of information between two partners.

For several years there have been different opinions regarding the expressiveness of asynchronous communication. The encodings of Honda-Tokoro [14] and Boudol [5] for the output prefix and of Nestmann-Pierce [32] for the input prefix partially legitimated the idea that the two communication mechanisms are equivalent. However, the full π-calculus, and process algebras with synchronous communication in general, have a mixed choice mechanism, i.e. choice of prefixed processes where the prefixes can be input, output or silent actions, that increases their expressive power. In fact, Palamidessi [36] has shown that the π-calculus is strictly more expressive than the asynchronous π-calculus, in the sense that it is not possible to encode the first into the latter in a uniform way while preserving a reasonable semantics. Uniform essentially means homomorphic with respect to the parallel and the renaming operators, and reasonable means sensitive to the capability of achieving success in all possible computations. The additional expressive power is due exactly to the mixed choice construct, as shown by Nestmann [29].

The motivation for this work comes from the observation that the π-calculus, and the formalisms that are based on synchronous communication and contain a guarded choice operator, are difficult to implement in an entirely distributed way with non-probabilistic methods. The combination of synchronous communication and choice requires in fact solving certain problems of distributed consensus which are known to have only randomized solutions. On the other hand, the asynchronous π-calculus, as well as the formalisms based on asynchronous communication, are usually more suitable for a distributed implementation, but are not as expressive as the synchronous formalisms.

The difference in expressive power is due to the specification of symmetric systems which need achieving consensus among remote components. In the π-calculus it is easy to specify solutions to distributed consensus problems, such as the leader election or the dining philosophers. In an asynchronous formalism on the contrary this is not possible, unless some form of randomization is introduced [21]. However, a probabilistic solution is usually much more complicated and difficult to reason about. A distributed and necessarily probabilistic implementation of the π-calculus will therefore offer a powerful language for programming the solution of distributed problems which would otherwise require complicated randomized algorithms. Moreover, the main difficulties of verification will be solved at the implementation level. In this way proving the correctness of a solution will be much easier because it will not require reasoning about probabilistic behaviors, but only reasoning about the correct use of certain high-level mechanisms.

We propose the probabilistic asynchronous π-calculus with the aim of providing a fully distributed implementation of the π-calculus. The implementation of the π-calculus will be achieved in two phases, by translating the π-calculus into an intermediate formalism, the probabilistic asynchronous π-calculus, and then implementing the latter. There are various advantages to this approach. First, the complexity of the implementation is factorized. Second, the main difficulty in proving the correctness of the implementation, which lies in the probabilistic algorithm, will be confined to the first phase. Since the intermediate formalism is itself a small calculus, it will be feasible to develop rigorous semantics and verification methods for it. In this way high level mathematical techniques can be used to prove the correctness of the implementation. Finally, the intermediate level will provide a formalism for refining specifications. Namely, if a more efficient solution to a problem is desired, i.e. more efficient than the resulting probabilistic implementation of a non-probabilistic specification, then one can use the intermediate calculus for specifying a better probabilistic algorithm, and then prove its correctness with respect to the original specification.

The probabilistic asynchronous π-calculus is an extension of the asynchronous π-calculus with a notion of random choice. The operational semantics of the probabilistic asynchronous π-calculus is based on the probabilistic automata of Segala and Lynch. The main characteristic of this model is that it distinguishes between probabilistic and nondeterministic behavior. The first is associated with the random choices of the process, while the second is related to the arbitrary decisions of an external scheduler. With respect to other probabilistic process algebras which have been defined in literature (see for example [54]), this separation is a novelty. One of its advantages is that it allows us to reason about adverse conditions, i.e. schedulers that try to prevent processes from achieving their goals. This is fundamental for our goal of implementing the π-calculus, since we are aiming at an implementation which is correct with respect to adversary schedulers. Another novelty of our proposal with respect to probabilistic process algebra is the definition of the parallel operator in a CCS style, as opposed to the SCCS style. Namely, parallel processes are not forced to proceed simultaneously. Also note that for general probabilistic automata it is not possible to define the parallel operator ([47]), or at least, there is no natural definition. In our proposal the parallel operator is defined as a natural extension of the non-probabilistic case. This can be considered, in our opinion, another argument for the suitability of our calculus for a distributed implementation.

Based on the probabilistic asynchronous π-calculus, we propose a randomized distributed encoding of the π-calculus, which requires solving a resource allocation problem

similar to the one of the generalized dining philosophers. The encoding is robust with respect to a large class of adversary schedulers: they can make use of all the information about the state and history of the system, including the result of the past random choices of the processes. The only assumption we need is that the scheduler treats the output action of the asynchronous π-calculus "properly", i.e. as a message that should eventually become available to the reader. Note that the definition of a proper scheduler is weaker than the notion of fair scheduler, which requires that *any* process which is ready infinitely often will eventually be scheduled for execution. The importance of considering adversary schedulers is not only theoretical, as argued in [42]: "*We allow for the possibility of an adversary scheduler since we assume that the interaction we describe* [...] *are only the visible part of an iceberg of complex relations about which we do not know and that we are not willing to study. We are to assume that the worst may happen, which is a very sound principle of system design.*". We also regard as a pleasant feature of our encoding the fact that it does not require the fairness assumption on the scheduler. Most of the randomized algorithms for coordination of distributed processes do require fairness, including the one in [42], but the implementations of concurrent programming languages (for instance Java) usually do not guarantee a fair scheduling policy.

In order to prove the correctness of the encoding we develop an extension of the notion of testing semantics ([34, 3]) for the probabilistic asynchronous π-calculus. This semantics is sensitive to divergencies and deadlocks, hence it is "reasonable" in the sense of [36]. We will show that our encoding is correct in the sense that translated processes preserve, under any proper adversary, and with probability 1, the may and must conditions with respect to each translated observer. There have been other notions of testing semantics developed for probabilistic automata or similar systems, see [17, 16, 48], however, those notions are formalized as orderings among probabilistic processes, and as such they would not be suitable to formulate the correctness of the encoding, which needs to be stated as a correspondence between processes of different kind (non-probabilistic and probabilistic). It is worth noting that we could not use bisimulation, barbed bisimulation, or coupled simulation either, not even in their weak and asynchronous versions, because these semantics are on one hand "too concrete" for the kind of translation developed here, and, on the other hand, they are not sensitive to divergencies.

The interest in considering π_{pa} as target language also lies on the fact that it can be implemented in a distributed way, i.e. without using centralized control or shared memory. In fact, like in the asynchronous π-calculus, the output actions are not allowed

to have a continuation, hence they can be mapped naturally into asynchronous communication, which is the only form of communication available in a distributed architecture. We define a uniform implementation of the probabilistic asynchronous π-calculus into the Java language. The condition of uniformity on the encodings of the π-calculus into the probabilistic asynchronous π-calculus and of the latter into Java ensure that the distribution and symmetry are preserved, thus we can argue that our results provide an approach to the distributed and symmetric implementation of the π-calculus.

Outline of the thesis

The rest of the dissertation is organized as follows. Chapter 2 contains an overview of the π-calculus and of the asynchronous π-calculus, followed by a sequence of examples of synchronous and asynchronous communication. Chapter 3 reviews the probabilistic automata. Chapter 4 presents the probabilistic asynchronous π-calculus (π_{pa} for short) and shows an example of a distributed problem that can be solved in π_{pa}, namely the election of a leader in a symmetric network. In Chapter 5 we consider a generalization of the dining philosophers problem to arbitrary connection topologies, which requires solving a resource allocation problem similar to the one of encoding the π-calculus with mixed choice into π_{pa}. In Chapter 6 we show a distributed and randomized encoding of the π-calculus with mixed choice and we prove the correctness of our encoding with respect to a probabilistic extension of testing semantics. Chapter 7 contains a distributed implementation of the probabilistic asynchronous π-calculus into the Java language. In Chapter 8 we conclude and give possible future directions for the work presented in this dissertation.

Chapter 2

The synchronous and the asynchronous π-calculi

We start this chapter by reviewing the definition of the π-calculus with mixed choice and of the asynchronous π-calculus. We recall both the syntax and the operational semantics of these calculi and we give a sequence of examples that illustrates mobility in the π-calculus.

The π-calculus is a theory of mobile systems based on synchronized communication, in which the exchange of information between two processes is simultaneous. Synchronous communication is a very effective mechanism but has the disadvantage of being costly since it requires the partners to synchronize to establish the communication.

Asynchronous communication on the other hand is less costly since the action of sending a message and the action of receiving it usually happens at different times. Many mobile systems, especially distributed systems, use forms of asynchronous communication.

The asynchronous π-calculus is a subset of the π-calculus in which the action of sending a message is a non-blocking operation. The asynchronous π-calculus is one of the richest paradigms for asynchronous communication in mobile systems that has been introduced so far. Therefore studying the expressiveness of the asynchronous fragment of the π-calculus has been of much interest in the recent years.

In the second part of the chapter we review the results regarding the expressive power of the asynchronous π-calculus compared to the synchronous π-calculus.

2.1 The π-calculus

Many variants of the π-calculus have been proposed. The original version of the π-calculus is the one given by Milner, Parrow and Walker in [27]. We chose to follow the variant of the π-calculus given by Davide Sangiorgi in [45]. The main difference with respect to [27] is the absence of a matching operator, and the replacement of free choice with a choice of prefixed processes where the prefixes can be input, output or silent actions. This type of choice is usually called a mixed choice. For this presentation we found more convenient to use recursion instead of the replication operator. Replication

is a simple form of recursion and it is well known that the two operators are equivalent in terms of expressive power ([26], [46]).

Consider a countable set of *channel names*, $x, y, \ldots$, and a countable set of *process names* $X, Y, \ldots$. The set of prefixes, $\alpha, \beta, \ldots$, and the set of π-calculus processes, $P, Q, \ldots$, are defined by the following syntax:

$$\begin{array}{rrcl} \textit{Prefixes} & \alpha & ::= & x(y) \mid \bar{x}y \mid \tau \\ \textit{Processes} & P & ::= & \sum_i \alpha_i.P_i \mid \nu x P \mid P \mid P \mid X \mid rec_X P \end{array}$$

Processes express mobile systems. Channel names can be thought of as names of communication links and are used by processes to interact. Processes evolve by performing actions. Prefixes represent the basic actions of processes: $x(y)$ is the *input* of the (formal) name y from channel x, $\bar{x}y$ is the *output* of the name y on channel x, τ stands for any silent (non-communication) action.

The process $\sum_i \alpha_i.P_i$ represents guarded choice and it is usually assumed to be finite. This process behaves like one or the other of the P_i. We use the abbreviations $\mathbf{0}$ (*inaction*) to represent the empty sum, $\alpha.P$ (*prefix*) to represent sum on one element only, and $\alpha_1.P_1 + \alpha_2.P_2$ for the binary sum.

The process $P_1 \mid P_2$ consists of P_1 and P_2 acting in parallel. The components may act independently or they may synchronize to create a silent action.

The symbol νx is the *restriction* operator. The process $\nu x P$ behaves like P except for two aspects. The first aspect is that the top-level transitions labeled by actions on x are not allowed, so such actions, performed by components of P, are forced to synchronize with their matching actions. In other words, the restriction operator allows enforcing synchronization (and communication) among parallel components of the system. The second aspect is that the name x is bound by the operator ν, as it is explained below, so it can be seen as a newly created name, private to P.

Note that communication between components of P along channel x are allowed.

The process $rec_X P$ represents a process X defined as $X \stackrel{\text{def}}{=} P$, where P may contain occurrences of X (recursive definition). We assume that all occurrences of X in P are prefixed. Recursion is the operator that makes it possible to express infinite behavior.

We use the following operator precedence for process expressions: prefixing, restriction and replication bind more tightly than parallel composition, and prefixing more

tightly than sum. The parallel composition is left associative, i.e. $P_0|P_1|P_2|\ldots|P_k$ stands for $(\ldots((P_0|P_1)|P_2)|\ldots|P_k)$. We use parentheses in all other cases to avoid ambiguity.

The operators νx and $y(x)$ are *x-binders*, i.e. in the processes νxP and $y(x).P$ the occurrences of x in P are considered *bounded* with scope P. The *free names* of P, i.e. those names which do not occur in the scope of any binder, are denoted by $fn(P)$.

The *alpha-conversion* of bounded names formalizes the idea that the names of bound variables do not matter. More formally, processes P and Q are *alpha-convertible* if Q can be obtained from P by a finite number of changes of bound names.

The renaming (or substitution) $P[y/x]$ is defined as the result of replacing all occurrences of x in P by y, possibly applying alpha-conversion to avoid capture of names by binders.

The operational semantics is specified via a transition system labeled by *actions*. A π-calculus transition $P \xrightarrow{\alpha} Q$ means that P can evolve into Q, and in doing so performs the action α.

The actions are given by the following grammar:

$$\textit{Actions} \quad \mu \quad ::= \quad x(y) \;\mid\; \bar{x}y \;\mid\; \bar{x}(y) \;\mid\; \tau$$

The four kinds of action are explained in the following:

1. $x(y)$ is an input action. The transition $P \xrightarrow{x(y)} Q$ means that P can receive any name w on channel x and then evolve into $Q[w/y]$.

2. $\bar{x}y$ is a free output. The transition $P \xrightarrow{\bar{x}y} Q$ means that P can send the free name y on channel x

3. $\bar{x}(y)$ is a bound output. The transition $P \xrightarrow{\bar{x}(y)} Q$ means that P sends a private name on channel x and (y) is a reference to where this private name occurs. Hence y is not a free name.

4. τ is the silent action. The transition $P \xrightarrow{\tau} Q$ means that P can evolve into Q, and in doing so requires no interaction with the environment. This kind of action arises from processes of the form $\tau.P$ or from internal communication.

We have all the actions corresponding to prefixes, plus the *bounded output* $\bar{x}(y)$. This is introduced to model *scope extrusion*, i.e. the result of sending to another process

a private (ν-bounded) name. The bounded names of an action μ, $bn(\mu)$, are defined as follows: $bn(x(y)) = bn(\bar{x}(y)) = \{y\}$; $bn(\bar{x}y) = bn(\tau) = \emptyset$. Furthermore, we indicate by $n(\mu)$ all the *names* which occur in μ.

In literature there are two main definitions for the transition system of the π-calculus, which induce two different semantics: the *early* and the *late* bisimulation semantics ([28]). The two differ mainly because of how they mimic the input action. In an early transition $P \xrightarrow{xy} Q$, the action xy records both the name used for receiving, and the name received. In a late transition $P \xrightarrow{x(y)} Q$, z is a placeholder for the name to be received and not the name itself. In the early bisimulation semantics variables are instantiated at the time of inferring the input transition. In the late bisimulation semantics the input actions contain bound objects which become instantiated only when inferring an internal communication. Here we choose to present the late bisimulation semantics because it is more refined, hence more challenging for obtaining positive embedding results.

The rules for the late semantics are given in Table 2.1. The symbol $\equiv$ used in Rule Cong stands for *structural congruence*, a form of equivalence which identifies "statically" two processes and which is used to simplify the presentation. We assume this congruence to satisfy the following:

(i) $P \equiv Q$ if Q can be obtained from P by alpha-renaming, notation $P \equiv_\alpha Q$,

(ii) $P \mid \mathbf{0} \equiv P, \quad P \mid Q \equiv Q \mid P, \quad$ and $\quad (P \mid Q) \mid R \equiv P \mid (Q \mid R)$,

(iii) $\sum_i \alpha_i.P_i \equiv \sum_i \alpha_{\rho(i)}.P_{\rho(i)}$ if ρ is a permutation of indexes,

(iv) $rec_X P \equiv P[rec_X P/X]$,

(v) $(\nu x P) \mid Q \equiv \nu x(P \mid Q)$ if $x \notin fn(Q)$.

2.2 Examples of synchronous communication

Example 2.1. The first example is from [38] and illustrates the interaction between a server, a client and a printer. The client wishes to use a printer that is controlled by the server. Initially there is a link a between the server and the printer and a link b between the server and the client. When the client wants to access the printer, the server sends a along b. This is represented by $\bar{b}a.S$. The client receives some link along b and then uses this link to send data to the printer. This is represented by $b(c).\bar{c}d.P$. The interaction described above is illustrated by the following transition:

Sum	$\sum_i \alpha_i.P_i \xrightarrow{\alpha_j} P_j$
Open	$\dfrac{P \xrightarrow{\bar{x}y} P'}{\nu y P \xrightarrow{\bar{x}(y)} P'} \quad x \neq y$
Res	$\dfrac{P \xrightarrow{\mu} P'}{\nu y P \xrightarrow{\mu} \nu y P'} \quad y \notin n(\mu)$
Par	$\dfrac{P \xrightarrow{\mu} P'}{P \mid Q \xrightarrow{\mu} P' \mid Q} \quad bn(\mu) \cap fn(Q) = \emptyset$
Com	$\dfrac{P \xrightarrow{\bar{x}y} P' \quad Q \xrightarrow{x(z)} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'[y/z]}$
Close	$\dfrac{P \xrightarrow{\bar{x}(y)} P' \quad Q \xrightarrow{x(y)} Q'}{P \mid Q \xrightarrow{\tau} \nu y(P' \mid Q')}$
Cong	$\dfrac{P \equiv P' \quad P' \xrightarrow{\mu} Q' \quad Q' \equiv Q}{P \xrightarrow{\mu} Q}$

Table 2.1. The late-instantiation transition system of the π-calculus.

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

Notice that a represents both the name of the communication link and an object which is transferred between the server and the client.

The fact that a is a local link between the server and the printer is captured by $(\nu a)(\bar{b}a.S \mid R)$, where R represents the printer. The result of sending the link a along b to the client is a private link shared by the server, the client and the printer. The transition in this case is:

$(\nu a)(\bar{b}a.S \mid R) \mid b(c).\bar{c}d.P \xrightarrow{\tau} (\nu a)(S \mid R \mid \bar{a}d.P)$

This behavior is called link passing in [27].

Example 2.2. Scope intrusion is one of the basic examples presented in the first paper on the π-calculus ([27]). In this case there is a link x between P and R. P wishes to pass this link to Q along a link y that P and Q share. If Q already has a private link named x that is shared with S, then this link has to be renamed in order to avoid confusion. For this reason it is said that P intrudes the scope of the private link x between Q and S. Assuming that P is of form $\bar{y}x.P'$ and Q is $y(z).Q'$, then the previous situation is represented by the transition

$\bar{y}x.P' \mid R \mid (\nu x)(y(z).Q' \mid S) \xrightarrow{\tau} P' \mid R \mid (\nu x')(Q" \mid S')$

where $Q"$ is $Q'\{x'/x\}\{x/z\}$ and S' is $S\{x'/x\}$.

Example 2.3. Another example showed in [27] is scope extrusion. Consider P and Q of the same form as in the previous example. P has a private link x to R that wishes to pass along its link y to Q. Q has no link called x. By exporting the private link x to Q, P extrudes the scope of this link. The following transition describes this situation:

$(\nu x)(\bar{y}x.P' \mid R) \mid y(z).Q' \xrightarrow{\tau} (\nu x)(P' \mid R \mid Q")$

where $Q"$ is $Q'\{x/z\}$.

It can be noticed from the previous examples that a communication occurs between two parallel processes when one sends a name on a particular channel and the other one is waiting for a name along the same channel. The output operation is blocking in the sense that an output guard cannot be executed unless an input action is simultaneously executed. The following transition is a typical example of synchronous communication:

$(\bar{x}v.S \mid x(y).R) \mid \xrightarrow{\tau} (S \mid R\{v/y\})$

2.3 The asynchronous π-calculus

The asynchronous π-calculus, as proposed by Boudol [5] and, independently, by Honda and Tokoro [14], is a subset of the π-calculus which contains no explicit operators for choice and output prefix.

We consider the definition of the asynchronous π-calculus given in [1], which differs from the original ones ([14, 5]) for the presence of a non-output choice construct, namely a summation of processes prefixed with τ or input actions. Actually, [1] considers a binary non-output sum operator instead of a n-ary one, but under the assumption that the binary sum is commutative and associative, the two definitions coincide. Thanks to [32], we know that this construct does not increase the expressive power. In [32] it is shown that the asynchronous π-calculus with input-guarded choice can be encoded into its choice-free fragment. It is easy to extend the encoding to include also τ prefixes.

The asynchronous π-calculus (π_a-calculus for short) is defined by the following grammar:

$$\begin{array}{rrcl} \textit{Non-output prefixes} & \alpha & ::= & x(y) \mid \tau \\ \textit{Processes} & P & ::= & \bar{x}y \mid \sum_i \alpha_i.P_i \mid \nu x P \mid P \mid P \mid X \mid \textit{rec}_X P \end{array}$$

The difference with respect to the π-calculus is that $\sum_i \alpha_i.P_i$ is restricted to non-output prefixes only, and output prefixes are replaced by output-action processes. The asynchronous communication is captured by preventing output prefixes from being part of guarded choices. As noted in [46], terms such as $\bar{x}y + a(z).Q$ are excluded. The reason is that such a process is capable of receiving via a, and if this happens, then the sending of y via x is rendered void. In other words, there is no correspondence between a datum that has been sent but not received and the appearance of an output-action process containing the datum.

The rule for the output-action process is described in Table 2.2, where **0** stands again for inaction. All the rules for the other operators are the same as those in Table 2.1.

The reason why this calculus is considered a paradigm for asynchronous communication is that there is no primitive *output prefix*, hence no primitive notion of continuation after the execution of an output action. While the communication between two partners in the (synchronous) π-calculus is usually understood as simultaneous, in the asynchronous π-calculus the action of sending a message and the action of receiving it do

OUT $\quad \bar{x}y \xrightarrow{\bar{x}y} \mathbf{0}$

Table 2.2. The output rule for the π_a-calculus.

not have to occur at the same time. In other words, a process executing an output action will not be able to detect when the corresponding input action is actually executed.

Note that the π_a-calculus is a proper subset of the π-calculus. The process $\bar{x}y$, in fact, could be equivalently replaced by $\bar{x}y.\mathbf{0}$.

2.4 Examples of asynchronous communication

In the asynchronous πcalculus output processes do not carry continuations. A name z is simply sent along a channel x and can be received by a process $x(y).P$ acting in parallel. The result of this interaction is $P\{z/y\}$. Note that the output action $\bar{x}z$ does not suspend the sender and that the underlying model of interaction among processes is the same as in the π-calculus, namely the simultaneous execution of complementary actions or also known as handshaking. The handshaking between $\bar{x}z$ and $x(y).P$ can be seen as the moment in which the message is received.

Example 2.4. (Asynchronous communication) An example of a simple reduction in π_a is illustrated by the following transition:

$x(v).\bar{y}v \mid y(z).P \mid \bar{x}w \xrightarrow{\tau} \bar{y}w \mid y(z).P \xrightarrow{\tau} P$

An asynchronous process that nondeterministically chooses to send $\bar{x}z$ or to receive at $x(y).P$ is written $\tau.\bar{x}z + x(y).P$.

For the next example consider the process $P = (\nu x)(\bar{y}x) \mid y(z).\bar{z}x$. The following reductions can be applied to P:

$P \xrightarrow{\tau} (\nu x')(\bar{y}x' \mid y(z).\bar{z}x) \xrightarrow{\tau} (\nu x')\bar{x}'x$

An α-conversion was applied to P in order to avoid capture. Also note that the scope of x' is extruded.

The next example shows that some semantic sequentialization of output actions can be achieved using private names, even if output prefixes are not allowed ([46]). Consider the following process:

$(\nu y \nu z)(\bar{x}y|\bar{y}z|\bar{z}a|R)$, where $y, z \notin fn(R)$.

The process R can proceed, but the three output processes can proceed only from left to right, i.e. in the order $\bar{x}y$, $\bar{y}z$, $\bar{z}a$.

2.5 Comparing the expressive power of the synchronous and the asynchronous π-calculus

As we mentioned before, the π-calculus is a very expressive specification language for concurrent programming, but difficult to implement in a distributed environment. Certain mechanisms of the π-calculus, in fact, require solving a problem of distributed consensus. The asynchronous π-calculus, on the other hand, is more suitable for a distributed implementation. Therefor, it is naturally to ask whether the π-calculus can be encoded in the asynchronous π-calculus. It is well known that the asynchronous communication can be simulated with synchronous mechanisms (see for example [15]).

In the following we briefly review some of the variants of the π-calculus and the encodings and separation results which have been investigated in the literature so far. A very good summary of the π-calculus hierarchy is shown in [35]. A more exhaustive overview of the variants of the π-calculus and of their expressive power is presented in [46].

- the π-calculus with mixed choice is a subset of the full π-calculus where the choice operator can occur only among prefixed processes. The mixed choice terminology is used to emphasize the fact that we can have input, output and silent guards in the same guarded choice. Another characteristic of the π-calculus with mixed choice is that the match operator is excluded.

- the π-calculus with separate choice is a subset of the π-calculus with mixed choice where the prefixes in a choice are all of the same kind. More specifically, in a guarded choice all prefixes which are not τ must be either input or output actions.

- the π-calculus with input choice is a subset of the π-calculus with separate choice where only input actions can be used in a choice.

- the π-calculus without choice excludes the choice operator, but has output prefixes.

- the asynchronous π-calculus is the subset of the π-calculus without choice and without output prefixes.

Honda and Tokoro, and independently Boudol, have shown that the output prefix can be simulated ([14], [5]) in the asynchronous π-calculus. Boudol used a *rendez-vous* mechanism to define an encoding of the output prefix in the asynchronous π-calculus. In this technique the receiver sends an acknowledgement to the sender when it receives the message, and the sender waits until it receives the acknowledgement. Honda and Tokoro proposed a fully compositional encoding in which the receiver is responsible for synchronizing with the sender.

More recently, Nestmann and Pierce presented the encoding of the π-calculus with input choice ([32]) into its choice-free fragment. This result was a significant step in the evolution of the asynchronous π-calculus since the input guarded choice mechanism is often used for programming concurrent systems. Moreover, since the encoding of Nestmann and Pierce, several authors have included the input choice in the definition of the asynchronous π-calculus (see for example [1] and [4]). The encoding of the input choice consists in running a mutual exclusion protocol which associates a lock with the parallel composition of its branches. The processes corresponding to branches concurrently try to acquire the lock and only the first process that is able to get the lock wins the competition and proceeds with its continuation. All the other branches abort their continuations. The encoding is divergence free and is fully abstract with respect to coupled simulation, which is an equivalence that does not require bisimilarity of internal branching decisions.

In [36] it has been shown that the π-calculus with mixed choice is strictly more expressive than the asynchronous π-calculus, in the sense that it is not possible to encode the first into the latter in a uniform way while preserving a reasonable semantics. Uniform means homomorphic with respect to the parallel and the renaming operators, and reasonable means sensitive to the capability of achieving success in all possible computations. In other words a reasonable semantics distinguishes two processes P and Q whenever in some computation of P the actions on certain intended channels are different from those in any computation of Q. The homomorphic behavior of the parallel operator ensures that two parallel processes are translated into two parallel processes, i.e. $[\![P|Q]\!] = [\![P]\!]|[\![Q]\!]$. In this way no coordinator can be added by the translation and therefore the degree of distribution of the processes is preserved by the encoding. The renaming preservation ensures that the translation does not depend on channel names

and therefore it preserves the portability of processes across the nodes of a distributed network.

The separation result is essentially based on the fact that in the π-calculus we can define an algorithm for distributed consensus, while this is not possible with the asynchronous π-calculus. A very simple example intuitively shows this fact. Consider two symmetric processes P and Q, where $P = \bar{x}z.P_1 + yv.P_2$ and $Q = xp.Q_1 + \bar{y}q.Q_2$. We can easily enforce communication on x and y due to the use of the mixed choice mechanism, namely the parallel composition $P|Q$ becomes one of the asymmetric processes $P_1|Q_1[z/p]$ or $P_2[q/v]|Q_2$. However, this behavior cannot be simulated in the asynchronous π-calculus. Processes P and Q have to agree on the communication channel (x or y) by breaking the initial symmetry, but because both P and Q behave confluently the symmetry of $P|Q$ is preserved under computation and no leader can be elected. The confluence property for a process P means that if P can make two transitions $P \xrightarrow{\bar{x}z} P_1$ and $P \xrightarrow{xv} P_2$, then there exists P' such that $P_1 \xrightarrow{xv} P'$ and $P_2 \xrightarrow{\bar{x}z} P'$. Figure 2.1 shows the behavior of a confluent process P.

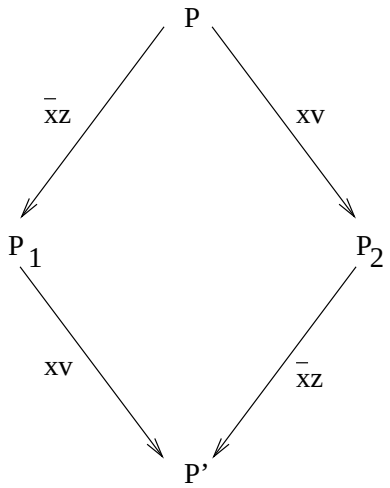

Fig. 2.1. The confluence property for a process P

Another remarkable result was proved by Nestmann in [29]. Nestmann has been shown that the additional expressive power is due exactly to the mixed choice construct since the π-calculus with separate choices can be encoded in the asynchronous π-calculus. Nestmann's encoding of separate choices is an extension of the encoding of input guarded choice of [32]. The encoding relies on two locks, a local lock that is associated with an input branch and is always tested first, and a remote lock that corresponds to an output branch and is tested only if the outcome of testing the local lock is positive. The encoding also uses an acknowledgement channel for sending messages when both locks were tested positively. Nestmann's encoding of separate choice is deadlock and divergence free.

Figure 2.2 summarizes the previous encodings and impossibility result.

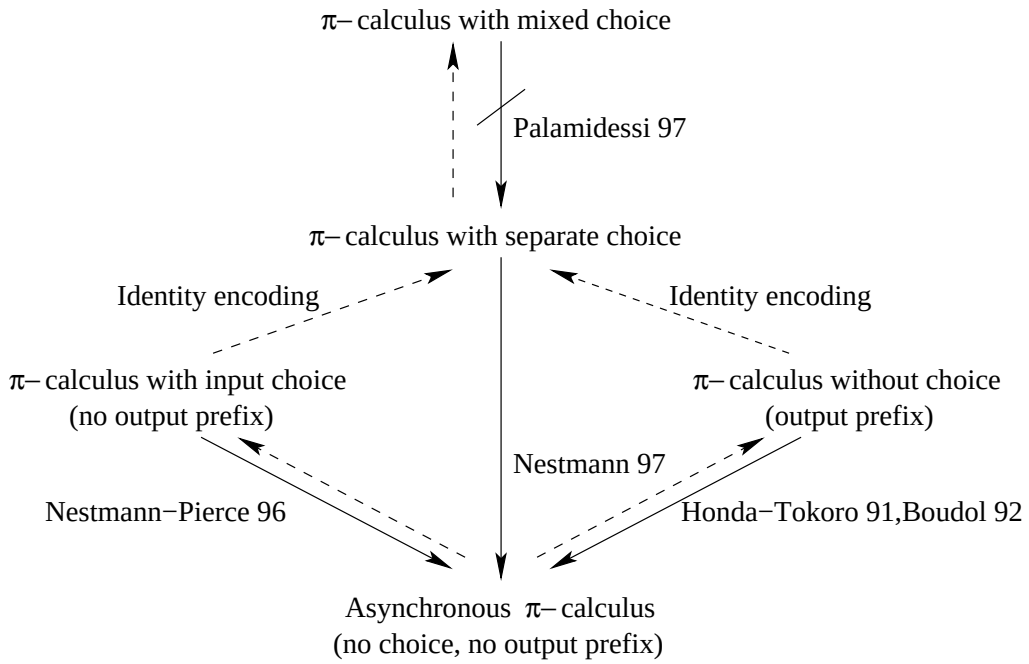

Fig. 2.2. The π-calculus hierarchy. The dashed line represents an identity encoding.

Chapter 3

Probabilistic automata

Probabilistic automata have been proposed in [49, 47]. The characteristic of this model is that it distinguishes between nondeterminism and probabilistic choice. The nondeterminism refers to the choices made by an external agent. Schedulers are examples of such agents. The probabilistic choice is a choice made internally by the process, and not controlled by the external agent. In this chapter we recall the definitions of probabilistic automata, executions, adversaries and probabilistic statements. The presentation follows [49], [47], to which the reader is referred for more details. The main difference is that we consider only discrete probabilistic spaces, and that the concept of deadlock is simply a node with no out-transitions.

Probabilistic automata describe systems that can evolve according to some probability distribution. Flipping a coin is an example of such system.

Definition 3.1. *A discrete probabilistic space is a pair* (X, pb) *where* X *is a set and* pb *is a function* $pb : X \to (0, 1]$ *such that* $\sum_{x \in X} pb(x) = 1$.

Given a set Y, we define

$Prob(Y) = \{(X, pb) \mid X \subseteq Y$ and (X, pb) is a discrete probabilistic space$\}$.

Definition 3.2. *Given a set of states* S *and a set of actions* A*, a* probabilistic automaton *on* S *and* A *is a triple* $(S, \mathcal{T}, s_0)$ *where* $s_0 \in S$ *(initial state) and* $\mathcal{T} \subseteq S \times Prob(A \times S)$.

We call the elements of $\mathcal{T}$ *transition groups* (in [49] they are called *steps*). The idea behind this model is that the choice between two different groups is made nondeterministically and possibly controlled by an external agent, e.g. a scheduler, while the transition within the same group is chosen probabilistically and is controlled internally

(e.g. by a probabilistic choice operator). In [49] each transition may contain a special symbol δ, representing the situations in which the system deadlocks. We omit the symbol δ in this presentation, and we represent deadlock as a node with no out-transitions.

If at most one transition group is allowed for each state, the automaton is called *fully probabilistic*.

Example 3.1. Figures 3.1 and 3.2 give examples of a probabilistic and a fully probabilistic automaton, respectively. The transition groups of the probabilistic automaton are labeled by I, II, ..., VI.

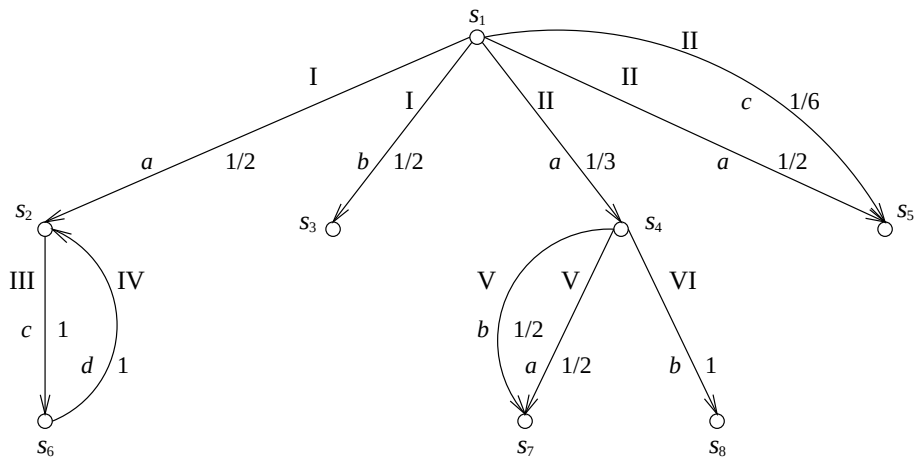

Fig. 3.1. Example of a probabilistic automaton M.

In [49] it is remarked that this notion of automaton subsumes and extends both the *reactive* and *generative* models of probabilistic processes ([54]). In particular, the generative model corresponds to the notion of fully probabilistic automaton.

Given a probabilistic automaton $M = (S, \mathcal{T}, s_0)$, define $tree(M)$ as the tree obtained by unfolding the transition system, i.e. the tree with a root n_0 labeled by s_0, and such that, for each node n, if $s \in S$ is the label of n, then for each $(s, (X, pb)) \in \mathcal{T}$, and

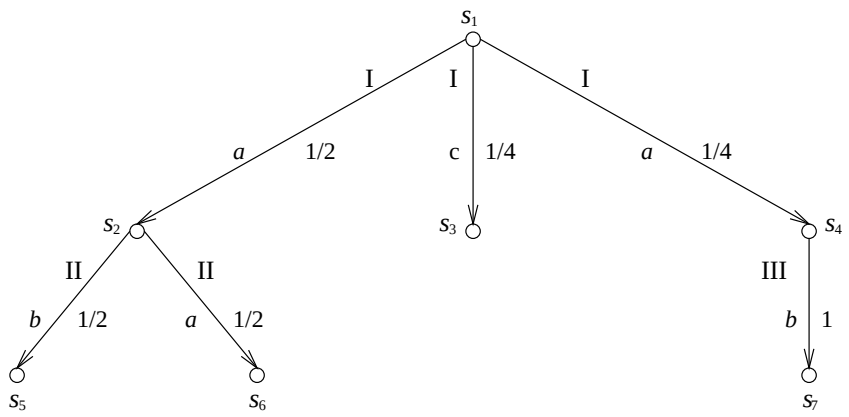

Fig. 3.2. A fully probabilistic automaton

for each $(\mu, s') \in X$, there is a node n' child of n labeled by s', and the arc from n to n' is labeled by μ and $pb(\mu, s')$.

We will denote by $nodes(M)$ the set of nodes in $tree(M)$, and by $state(n)$ the state labeling a node n.

Example 3.2. Figure 3.3 represents the tree obtained from the probabilistic automaton M of Figure 3.1.

A trace α in a probabilistic automaton M is any path in $tree(M)$. We are only interested in finite traces, and we use the notation $lnode(\alpha)$ for the last node of α and $fnode(\alpha)$ for the first node of α. If α_1 is $n_0 \xrightarrow[p_0]{\mu_0} n_1 \dots \xrightarrow[p_{k-1}]{\mu_{k-1}} n_k$ and α_2 is $n_k \xrightarrow[p_k]{\mu_k} n_{k+1} \dots \xrightarrow[p_l]{\mu_l} n_l$, then the trace $\alpha_1 \frown \alpha_2 = n_0 \xrightarrow[p_0]{\mu_0} n_1 \dots \xrightarrow[p_{k-1}]{\mu_{k-1}} n_k \xrightarrow[p_k]{\mu_k} n_{k+1} \dots \xrightarrow[p_l]{\mu_l} n_l$ represents the concatenation of the traces α_1 and α_2. We denote by $traces(M)$ the set of finite traces of M.

We define now the notion of execution of an automaton under a *scheduler*, by adapting and simplifying the corresponding notion given in [49]. A scheduler can be seen as a function which solves the nondeterminism of the automaton by selecting, at each moment of the computation, a transition group among all the ones allowed in the present state. Schedulers are sometimes called *adversaries*, thus conveying the idea of

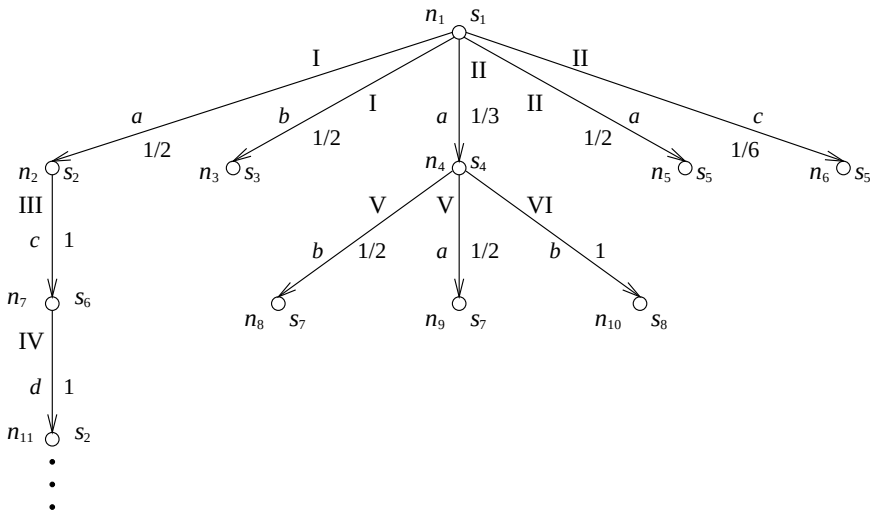

Fig. 3.3. The tree obtained from the probabilistic automaton M

an external entity playing "against" the process. A process is *robust* with respect to a certain class of adversaries if it gives the intended result for each possible scheduling imposed by an adversary in the class. Clearly, the reliability of an algorithm depends on how "smart" the adversaries of this class can be. We will assume that an adversary can decide the next transition group depending not only on the current state, but also on the whole history of the computation till that moment, including the random choices made by the automaton.

Definition 3.3. *An adversary for M is a function that takes a finite trace α in M and returns a transition group among those which are allowed in state($lnode(\alpha)$). More formally, $\zeta : traces(M) \rightarrow Prob(A \times S)$ such that $\zeta(\alpha) = (X, pb)$ implies $(state(lnode(\alpha)), (X, pb)) \in \mathcal{T}$.*

Definition 3.4. *The execution tree of an automaton $M = (S, \mathcal{T}, s_0)$ under an adversary ζ, denoted by $etree(M, \zeta)$, is the tree obtained from $tree(M)$ by pruning all the arcs*

corresponding to transitions which are not in the group selected by ζ. *More formally,* $etree(M, \zeta)$ *is a fully probabilistic automaton* $(S', \mathcal{T}', n_0)$, *where* $S' \subseteq nodes(M)$, n_0 *is the root of* $tree(M)$, *and* $(n, (X', pb')) \in \mathcal{T}'$ *iff* $X' = \{(\mu, n') \mid (\mu, state(n')) \in X\}$ *and* $pb'(\mu, n') = pb(\mu, state(n'))$, *where* $(X, pb) = \zeta(n)$.

Example 3.3. Figure 3.4 represents the execution tree of the automaton M of Figure 3.1, under an adversary ζ.

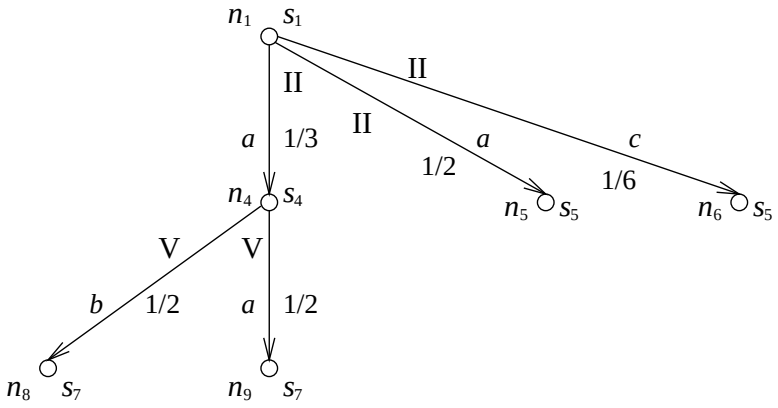

Fig. 3.4. etree(M, ζ), where M is the probabilistic automaton M of Figure 3.1, and (the significant part of) ζ is defined by $\zeta(n_1) = \text{II}$, $\zeta(n_4) = \text{V}$

An *execution fragment* ξ is any path (finite or infinite) from the root of $etree(M, \zeta)$. The notation $\xi \leq \xi'$ means that ξ is a prefix of ξ'. If ξ is $n_0 \xrightarrow[p_0]{\mu_0} n_1 \xrightarrow[p_1]{\mu_1} n_2 \xrightarrow[p_2]{\mu_2} \ldots$, the *probability* of ξ is defined as $pb(\xi) = \prod_i p_i$. If ξ is maximal, then it is called *execution*. We denote by $exec(M, \zeta)$ the set of all executions in $etree(M, \zeta)$

We define now a probability on certain sets of executions, following a standard construction of Measure Theory. Given an execution fragment ξ, let $C_\xi = \{\xi' \in exec(M, \zeta) \mid \xi \leq \xi'\}$ (*cone* with prefix ξ). Define $pb(C_\xi) = pb(\xi)$. Let $\{C_i\}_{i \in I}$ be a countable set

of disjoint cones (i.e. I is countable, and $\forall i, j.\ i \neq j \Rightarrow C_i \cap C_j = \emptyset$). Then define $pb(\bigcup_{i \in I} C_i) = \sum_{i \in I} pb(C_i)$. It is possible to show that pb is well defined, i.e. two countable sets of disjoint cones with the same union produce the same result for pb. We can also define the probability of an empty set of executions as 0, and the probability of the complement of a certain set of executions as the complement with respect to 1 of the probability of the set. The closure of the cones with respect to the empty set, the countable union, and the complementation generates what in Measure Theory is known as a σ-field.

Progress statements

Progress statements have been introduced in [23, 47] in order to study the properties of probabilistic automata. Progress statements are also called probabilistic statements and are generally used for the analysis of randomized algorithms. A progress statement is denoted by $S \xrightarrow[p]{A} S'$, where S and S' are sets of states, p is a probability, and A is a class of adversaries. Its meaning is that starting from any state in S, under any adversary in A, a state in S' is reached with probability at least p.

Progress statements can be combined to obtain more complex progress statements. This is a very important property since it allows the decomposition of a complex problem into simpler problems. The following properties of progress statement, proved in [23, 47], will be used in this presentation.

Lemma 3.1 (Union). *If* $S_1 \xrightarrow[p_1]{A} S_1'$ *and* $S_2 \xrightarrow[p_2]{A} S_2'$, *then* $S_1 \cup S_2 \xrightarrow[p]{A} S_1' \cup S_2'$ *with* $p = min\{p_1, p_2\}$.

A very useful property is the concatenation of progress statements. This property holds for adversaries whose power is not reduced if a prefix of the past history of a trace is not known.

Definition 3.5. *A class of adversaries A for a probabilistic automaton M is finite history insensitive iff for each adversary ζ of A and each trace α of M there is an adversary ζ' of M such that for each trace α' of M with $fstate(\alpha') = lstate(\alpha)$, $\zeta'(\alpha') = \zeta(\alpha \cap \alpha')$.*

Note that this definition corresponds exactly to the definition of execution closed adversary schema given in [23].

Lemma 3.2 (Concatenation). *Let A be a finite history insensitive class of adversaries. If $S_1 \xrightarrow[p_1]{A} S_2$ and $S_2 \xrightarrow[p_2]{A} S_3$, then $S_1 \xrightarrow[p_1 p_2]{A} S_3$.*

The next property shows how progress with probability 1 can be derived from a progress statement with probability p such that $0 < p < 1$. The statement S_1 *unless* S_2 is used, where S_1 is a set of states and S_2 is either a set of states only or a set of actions only. The statement is true for a probabilistic automaton M iff, once in S_1, the probabilistic automaton M remains in S_1 until the condition expressed by S_2 is satisfied. More formally, for each transition (s, P) of M, if $s \in S_1 - S_2$ then for each $(a, s') \in Prob(A \times S)$ either $a \in S_2$ or $s' \in S_1 \cup S_2$.

Lemma 3.3 (Progress with probability 1). *If $S_1 \xrightarrow[p]{F} S_2$ with $p > 0$, and S_1 unless S_2, then $S_1 \xrightarrow[1]{A} S_2$.*

Progress statements are used in the context of probabilistic automata to prove properties of randomized distributed algorithms. This method provides a structured and formal way to analyze numerous algorithms. To illustrate its efficiency, the method has been applied in [23, 47] to prove the correctness of the Lehmann and Rabin's Dinning Philosophers Algorithm.

Chapter 4

The probabilistic asynchronous π-calculus

In this chapter, we describe the probabilistic asynchronous π-calculus, an extension of the asynchronous π-calculus enhanced with a notion of random choice. The model is based on the probabilistic automata of Segala and Lynch ([49, 47]). An overview of the probabilistic automata theory is presented in Chapter 3.

We start the chapter with the definition of the probabilistic asynchronous π-calculus, π_{pa} for short, which is followed by several examples of communication in π_{pa}, with particular focus on synchronization and interleaving.

Next we show an example of a distributed problem that can be solved with the probabilistic asynchronous π-calculus, namely the election of a leader in a symmetric network. This problem cannot be solved with the asynchronous π-calculus as it has been proved in [36]. We propose an algorithm for the solution of this problem, and we prove that it is correct, i.e. that the leader will eventually be elected, with probability 1, under a large class of adversary schedulers. The only assumption is that the scheduler treats the output action of the asynchronous π-calculus "properly", i.e. an output message eventually becomes available to the input process. Our algorithm is reminiscent of the algorithm used in [42] for solving the dining philosophers problem, but in our case we do not need the fairness assumption. Also, the fact that we give the solution in a language provided with a rigorous operational semantics allows us to give a more formal proof of correctness.

4.1 Syntax and operational semantics

In this section we introduce the probabilistic asynchronous π-calculus and we give its operational semantics in terms of probabilistic automata. We define the operational semantics such that it distinguishes between probabilistic choice, made internally by the process, and nondeterministic choice, made externally by an adversary scheduler. This distinction will allow us to reason about the probabilistic correctness of algorithms under certain schedulers.

As argued in [11] we propose the probabilistic asynchronous π-calculus as a paradigm for the specification of distributed algorithms. The probabilistic asynchronous π-calculus increases the expressive power of the asynchronous π-calculus, which is suitable for a distributed implementation but is rather weak for solving distributed problems ([36])

The π_{pa}-calculus is obtained from the asynchronous π-calculus by replacing $\sum_i \alpha_i.P_i$ with the following *probabilistic choice operator*

$$\sum_i p_i \alpha_i.P_i$$

where the p_i's represents positive probabilities, i.e. they satisfy $p_i \in (0,1]$ and $\sum_i p_i = 1$, and the α_i's are input or silent prefixes.

In order to give the formal definition of the probabilistic model for π_{pa}, we find it convenient to introduce the following notation for representing transition groups: given a probabilistic automaton $(S, \mathcal{T}, s_0)$ and $s \in S$, we write

$$s \, \{\xrightarrow[p_i]{\mu_i} s_i \mid i \in I\}$$

iff $(s, (\{(\mu_i, s_i) \mid i \in I\}, pb)) \in \mathcal{T}$ and $\forall i \in I \; p_i = pb(\mu_i, s_i)$, where I is an index set. When I is not relevant, we will use the simpler notation $s \, \{\xrightarrow[p_i]{\mu_i} s_i\}_i$. We will also use the notation $s \, \{\xrightarrow[p_i]{\mu_i} s_i\}_{i:\phi(i)}$, where $\phi(i)$ is a logical formula depending on i, for the set $s \, \{\xrightarrow[p_i]{\mu_i} s_i \mid i \in I \text{ and } \phi(i)\}$.

The operational semantics of a π_{pa} process P is defined as a probabilistic automaton whose states are the processes reachable from P and the $\mathcal{T}$ relation is defined by the rules in Table 4.1. In order to keep the presentation simple, we impose some restrictions on the syntax of terms (see the caption of Table 4.1). In Appendix A we give an equivalent definition of the operational semantics without these restrictions.

The SUM rule models the behavior of a choice process. Note that all possible transitions belong to the same group, meaning that the transition is chosen probabilistically by the process itself. RES models restriction on channel y: only the actions on channels different from y can be performed and possibly synchronize with an external process. The probability is redistributed among these actions. PAR represents the interleaving of

$$\textsc{Sum} \quad \sum_i p_i \alpha_i.P_i \; \{\xrightarrow[p_i]{\alpha_i} P_i\}_i$$

$$\textsc{Out} \quad \bar{x}y \; \{\xrightarrow[1]{\bar{x}y} \mathbf{0}\}$$

$$\textsc{Open} \quad \frac{P \; \{\xrightarrow[1]{\bar{x}y} P'\}}{\nu y P \; \{\xrightarrow[1]{\bar{x}(y)} P'\}} \qquad x \neq y$$

$$\textsc{Res} \quad \frac{P \; \{\xrightarrow[p_i]{\mu_i} P_i\}_i}{\nu y P \; \{\xrightarrow[p'_i]{\mu_i} \nu y P_i\}_{i:y \notin fn(\mu_i)}} \qquad \begin{array}{l} \exists i.\ y \notin fn(\mu_i) \text{ and} \\ \forall i.\ p'_i = p_i / \sum_{j:y \notin fn(\mu_j)} p_j \end{array}$$

$$\textsc{Par} \quad \frac{P \; \{\xrightarrow[p_i]{\mu_i} P_i\}_i}{P \mid Q \; \{\xrightarrow[p_i]{\mu_i} P_i \mid Q\}_i}$$

$$\textsc{Com} \quad \frac{P \; \{\xrightarrow[1]{\bar{x}y} P'\} \qquad Q \; \{\xrightarrow[p_i]{\mu_i} Q_i\}_i}{P \mid Q \; \{\xrightarrow[p_i]{\tau} P' \mid Q_i[y/z_i]\}_{i:\mu_i = x(z_i)} \cup \{\xrightarrow[p_i]{\mu_i} P \mid Q_i\}_{i:\mu_i \neq x(z_i)}}$$

$$\textsc{Close} \quad \frac{P \; \{\xrightarrow[1]{\bar{x}(y)} P'\} \qquad Q \; \{\xrightarrow[p_i]{\mu_i} Q_i\}_i}{P \mid Q \; \{\xrightarrow[p_i]{\tau} \nu y(P' \mid Q_i[y/z_i])\}_{i:\mu_i = x(z_i)} \cup \{\xrightarrow[p_i]{\mu_i} P \mid Q_i\}_{i:\mu_i \neq x(z_i)}}$$

$$\textsc{Cong} \quad \frac{P \equiv P' \qquad P' \; \{\xrightarrow[p_i]{\mu_i} Q'_i\}_i \qquad \forall i.\ Q'_i \equiv Q_i}{P \; \{\xrightarrow[p_i]{\mu_i} Q_i\}_i}$$

Table 4.1. The late-instantiation probabilistic transition system of the π_{pa}-calculus. In Sum we assume that all branches are different, namely, if $i \neq j$, then either $\alpha_i \neq \alpha_j$, or $P_i \not\equiv P_j$. Furthermore, in Res and Par we assume that all bounded variables are distinct from each other, and from the free variables.

parallel processes. All the transitions of the processes involved are made possible, and they are kept separated in the original groups. In this way we model the fact that the selection of the process for the next computation step is determined by a scheduler. In fact, choosing a group corresponds to choosing a process. COM models communication by handshaking. The output action synchronizes with all matching input actions of a partner, with the same probability of the input action. The other possible transitions of the partner are kept with the original probability as well. CLOSE is analogous to COM, the only difference is that the name being transmitted is private to the sender. OPEN works in combination with CLOSE like in the standard (asynchronous) π-calculus. The other rules, OUT and CONG, should be self-explanatory.

Note that the parallel operator is associative. This property can be easily shown by case analysis.

Proposition 4.1. *For every process P, Q and R, the probabilistic automata of $P \mid (Q \mid R)$ and of $(P \mid Q) \mid R$ are isomorphic, in the sense that they differ only for the name of the states (i.e. the syntactic structure of the processes).*

We conclude this section with a discussion about the design choices of π_{pa}.

4.1.1 The rationale behind the design of π_{pa}

In defining the rules of the operational semantics of π_{pa} we felt there was only one natural choice, with the exception of the rules COM and CLOSE. For them we could have given a different definition, with respect to which the parallel operator would still be associative.

The alternative definition we had considered for COM was:

$$Com' \quad \frac{P \,\{\xrightarrow[1]{\bar{x}y} P'\} \qquad Q \,\{\xrightarrow[p_i]{\mu_i} Q_i\}_i}{P \mid Q \,\{\xrightarrow[p'_i]{\tau} P' \mid Q_i\}_{i:\mu_i = x(y)}} \quad \begin{array}{l} \exists i.\ \mu_i = x(y) \text{ and} \\ \forall i.\ p'_i = p_i / \sum_{j:\mu_j = x(y)} p_j \end{array}$$

and similarly for CLOSE.

The difference between COM and COM′ is that the latter forces the process performing the input action (Q) to perform only those actions that are compatible with the output action of the partner (P).

At first COM′ seemed to be a reasonable rule. At a deeper analysis, however, we discovered that COM′ imposes certain restrictions on the schedulers that, in a distributed setting, would be rather unnatural. In fact, the natural way of implementing the π_a communication in a distributed setting is by representing the input and the output partners as processes sharing a common channel. When the sender wishes to communicate, it puts a message in the channel. When the receiver wishes to communicate, it tests the channel to see if there is a message, and, in the positive case, it retrieves it. In case the receiver has a choice guarded by input actions on different channels, the scheduler can influence this choice by activating certain senders instead of others. However, if more than one sender has been activated, i.e. more than one channel contains data at the moment in which the receiver is activated, then it will be the receiver which decides internally which channel to select. COM models exactly this situation. Note that the scheduler can influence the choices of the receiver by selecting certain outputs to be premises in COM, and delaying the others by using PAR.

With COM′, on the other hand, when an input-guarded choice is executed, the choice of the channel is determined by the scheduler. Thus COM′ models the assumption that the scheduler can only activate (at most) one sender before the next activation of a receiver.

The following example illustrates the difference between COM and COM′.

Example 4.1. *Consider the processes* $P_1 = \bar{x}_1 y$, $P_2 = \bar{x}_2 z$, $Q = 1/3\ x_1(y).Q_1 + 2/3\ x_2(y).Q_2$, *and define* $R = (\nu x_1)(\nu x_2)(P_1 \mid P_2 \mid Q)$. *Under* COM, *the transition groups starting from* R *are*

$$R\,\{\xrightarrow[1/3]{\tau} R_1, \xrightarrow[2/3]{\tau} R_2\} \qquad R\,\{\xrightarrow[1]{\tau} R_1\} \qquad R\,\{\xrightarrow[1]{\tau} R_2\}$$

where $R_1 = (\nu x_1)(\nu x_2)(P_2 \mid Q_1)$ *and* $R_2 = (\nu x_1)(\nu x_2)(P_1 \mid Q_2)$. *The first group corresponds to the possibility that both* $\bar{x}_1$ *and* $\bar{x}_2$ *are available for input when* Q *is*

scheduled for execution. The other groups correspond to the availability of only $\bar{x}_1$ *and only* $\bar{x}_2$ *respectively.*

Under COM′*, on the other hand, the only possible transition groups are*

$$R\ \{\xrightarrow[1]{\tau} R_1\} \qquad R\ \{\xrightarrow[1]{\tau} R_2\}$$

Note that, in both cases, the only possible transitions are those labeled with τ*, because* $\bar{x}_1$ *and* $\bar{x}_2$ *are restricted at the top level.*

4.2 Examples of synchronization and interleaving in π_{pa}

In this section we will present some examples of processes in π_{pa}, with particular focus on synchronization and interleaving.

Example 4.2. *Consider the processes* $P = \mu_X(1/2\ x(y).\mathbf{0} + 1/2\ \tau.X)$*,* $Q = \bar{x}y$ *and define* $R = P \mid Q$*. The transition groups starting from* R *are:*

$$R\ \{\xrightarrow[1/2]{x(y)} Q\ ,\ \xrightarrow[1/2]{\tau} R\} \qquad R\ \{\xrightarrow[1/2]{\tau} \mathbf{0}\ ,\ \xrightarrow[1/2]{\tau} R\} \qquad R\ \{\xrightarrow[1]{\bar{x}y} P\}$$

Figure 4.1 illustrates the probabilistic automaton corresponding to R*. The above transition groups are labeled by I, II and III respectively.*

Example 4.3. *Consider the processes* P *and* Q *of example 4.2 and define* $R = (\nu x)(P \mid Q)$*. In this case the transition groups starting from* R *are:*

$$R\ \{\xrightarrow[1]{\tau} R\} \qquad R\ \{\xrightarrow[1/2]{\tau} \mathbf{0}\ ,\ \xrightarrow[1/2]{\tau} R\}$$

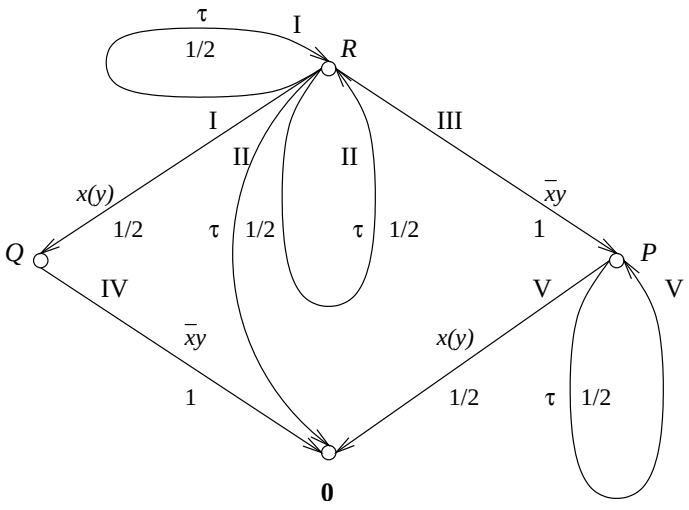

Fig. 4.1. The probabilistic automaton of Example 4.2

Figure 4.2 illustrates the probabilistic automaton corresponding to this new definition of R. The above transition groups are labeled by I and II respectively.

Example 4.4. *This example shows that the expansion law does not hold in* π_{pa}*. This should be no surprise, since the choices associated to the parallel operator and to the sum, in* π_{pa}*, have a different nature: the parallel operator gives rise to nondeterministic choices of the scheduler, while the sum gives rise to probabilistic choices of the process.*

Consider the processes $R_1 = x(z).P \mid y(z).Q$ *and* $R_2 = p\, x(z).(P \mid y(z).Q) + (1-p)\, y(z).(x(z).P \mid Q)$*. The transition groups starting from* R_1 *are:*

$$R_1 \{\xrightarrow[1]{x(z)} P \mid y(z).Q\} \qquad R_1 \{\xrightarrow[1]{y(z)} x(z).P \mid Q\}$$

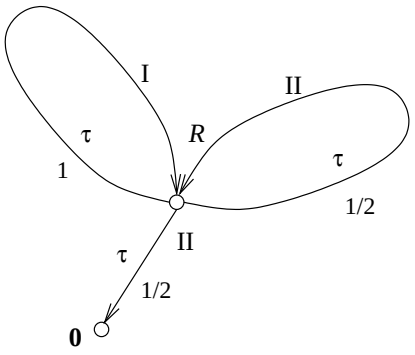

Fig. 4.2. The probabilistic automaton of Example 4.3

On the other hand, there is only one transition group starting from R_2, namely:

$$R_2 \ \{\xrightarrow[p]{x(z)} P \mid y(z).Q \ \ , \ \ \xrightarrow[1-p]{y(z)} x(z).P \mid Q\}$$

Figure 4.3 illustrates the probabilistic automata corresponding to R_1 and R_2.

4.3 Solving the electoral problem in π_{pa}

In [36] it has been proved that, in certain networks, it is not possible to solve the leader election problem by using the asynchronous π-calculus. The problem consists in ensuring that all processes will reach an agreement (elect a leader) in finite time. One example of such network is the system consisting of two symmetric nodes P_0 and P_1 connected by two internal channels x_0 and x_1 (see Figure 4.4).

In this section we will show that it is possible to solve the leader election problem for the above network by using the π_{pa}-calculus. Following [36], we will assume that the processes communicate their decision to the "external word" by using channels o_0 and o_1.

The reason why this problem cannot be solved with the asynchronous π-calculus is that a network with a leader is not symmetric, and the asynchronous π-calculus is not

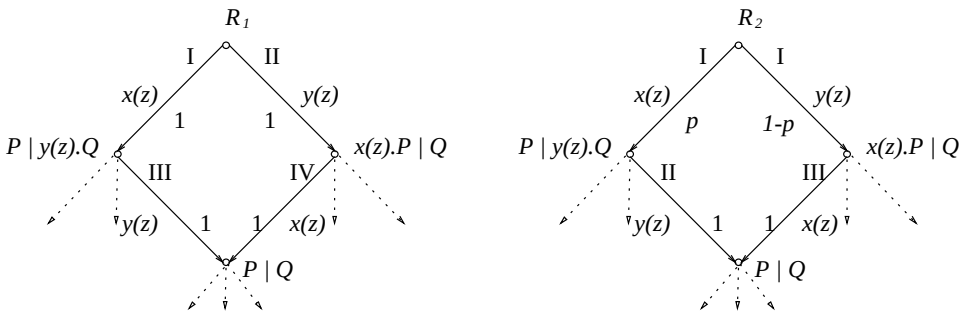

Fig. 4.3. The probabilistic automata R_1 and R_2 of Example 4.4

able to force the initial symmetry to break. Suppose for example that P_0 would elect itself as the leader after performing a certain sequence of actions. By symmetry, and because of lack of synchronous communication, the same actions may be performed by P_1. Therefore P_1 would elect itself as leader, which means that no agreement has been reached.

We propose a solution based on the idea of breaking the symmetry by repeating again and again certain random choices, until this goal has been achieved. The difficult point is to ensure that it will be achieved with probability 1 *under every proper scheduler*.

Our algorithm works as follows. Each process performs an output on its channel and, in parallel, tries to perform an input on both channels. If it succeeds, then it declares itself to be the leader. If none of the processes succeeds, it is because both of them performed exactly one input (thus reciprocally preventing the other from performing the second input). This might occur because the inputs can be performed only sequentially[1]. In this case, the processes have to try again. The algorithm is illustrated in Table 4.2.

In the algorithm, the selection of the first input is controlled by each process with a probabilistic blind choice, i.e. a choice whose branches are prefixed by a silent (τ) action. This means that the process commits to the choice of the channel *before* knowing whether it is available. It can be proved that this commitment is essential for

[1]In the π_{pa}-calculi and in most process algebra there is no primitive for simultaneous input action. Nestmann has proposed in [30] the addition of such construct as a way of enhancing the expressive power of the asynchronous π-calculus. Clearly, with this addition, the solution to the electoral problem would be immediate.

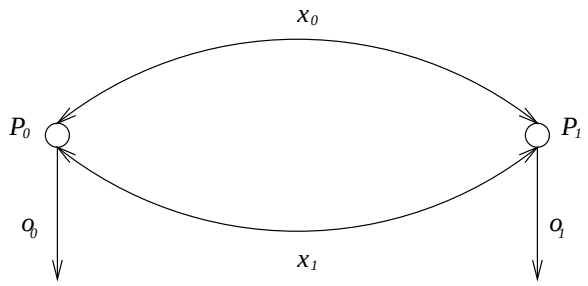

Fig. 4.4. A symmetric network $P = \nu x_0\ \nu x_1(P_0 \mid P_1)$. The restriction on x_0, x_1 is made in order to enforce synchronization.

ensuring that the leader will be elected with probability 1 under every possible adversary scheduler. The distribution of the probabilities, on the contrary, is not essential. This distribution however affects the efficiency (i.e. how soon the synchronization protocol converges). It can be shown that it is better to split the probability as evenly as possible (hence 1/2 and 1/2).

After the first input is performed, a process tries to perform the second input. What we would need at this point is a *priority choice*, i.e. a construct that selects the first branch if the prefix is enabled, and selects the second branch otherwise. With this construct the process would perform the input on the other channel when it is available, and backtrack to the initial situation otherwise. Since such construct does not exists in the π-calculi, we use probabilities as a way of approximating it. The symbol ε represents a small positive number. Thus we do not guarantee that the first branch will be selected for sure when the prefix is enabled, but we guarantee that it will be selected with probability close to 1. Of course, the smallest ϵ is, the more efficient the algorithm is.

When a process, say P_0, succeeds to perform both inputs, then it declares itself to be the leader. It also notifies this decision to the other process. For the notification we could use a different channel, or we may use the same channel, provided that we have a way to communicate that the output on such channel has now a different meaning. We follow this second approach, and we use boolean values t and f for messages. We stipulate that t means that the leader has not been decided yet, while f means that it has

$$
\begin{array}{lcl}
P_i & = & \bar{x}_i\langle t\rangle \\
 & & \mid rec_X(\ 1/2\ \tau.x_i(b).\ \textit{if } b \\
 & & \qquad \textit{then } (\ (1-\varepsilon)\ x_{i\oplus 1}(b).(\bar{o}_i\langle i\rangle \mid \bar{x}_i\langle f\rangle \mid \bar{x}_{i\oplus 1}\langle f\rangle) \\
 & & \qquad\qquad + \\
 & & \qquad\qquad \varepsilon\ \tau.(\bar{x}_i\langle t\rangle \mid X)) \\
 & & \qquad \textit{else } \bar{o}_i\langle i\oplus 1\rangle \\
 & & \quad + \\
 & & \quad 1/2\ \tau.x_{i\oplus 1}(b).\ \textit{if } b \\
 & & \qquad\qquad \textit{then } (\ (1-\varepsilon)\ x_i(b).(\bar{o}_i\langle i\rangle \mid \bar{x}_i\langle f\rangle \mid \bar{x}_{i\oplus 1}\langle f\rangle) \\
 & & \qquad\qquad\qquad + \\
 & & \qquad\qquad\qquad \varepsilon\ \tau.(\bar{x}_{i\oplus 1}\langle t\rangle \mid X)) \\
 & & \qquad\qquad \textit{else } \bar{o}_i\langle i\oplus 1\rangle\ )
\end{array}
$$

Table 4.2. A π_{pa} solution for the electoral problem in the symmetric network of Figure 4.4. Here $i \in \{0,1\}$ and $\oplus$ is the sum modulo 2.

been decided. Notice that the symmetry is broken exactly when one process succeeds in performing both inputs.

In the algorithm we make use of the if-then-else construct, which is defined by the structural rules

$$\mathit{if}\ t\ \mathit{then}\ P\ \mathit{else}\ Q \equiv P \qquad \mathit{if}\ f\ \mathit{then}\ P\ \mathit{else}\ Q \equiv Q$$

As discussed in [32], these features (booleans and if-then-else) can be translated into the asynchronous π-calculus, and therefore in π_{pa}.

4.3.1 Correctness of the algorithm

We prove now that the algorithm is correct, namely that the probability that a leader is eventually elected is 1 under every proper scheduler. First of all, we need to define the notion of proper scheduler. Clearly, we wish the algorithm to be correct with respect to a class of adversaries that is as large as possible. Yet, we cannot allow just *any* adversary. The problem is related to the output actions: a malicious adversary that never schedules $\bar{x}_i\langle t\rangle$ or $\bar{x}_{i\oplus 1}\langle t\rangle$ in the definition of P_i in Table 4.2 will make it impossible for processes to get the lock and therefore will force them to loop forever.

In the intended meaning of the asynchronous π-calculus, however, these actions represent messages rather than processes. The idea is that they are "sent" when they reach the top-level in a parallel context, and are "received" when the handshaking with the corresponding input action takes place. Thus it is reasonable to assume that the scheduler will not delay forever the reception of a message, i.e. if an output action is in parallel with a process able to execute the corresponding input action, then the handshaking will eventually take place.

Definition 4.1. *An adversary ζ for P is proper if, whenever P evolves into a process of the form $\nu x_1 \dots \nu x_k(P1|\dots|P_n)$, in which one of the P_i's is an output action on one of the channels $x_1 \dots x_k$, if ζ selects infinitely often a parallel process ready to execute the corresponding input action, then P_i will eventually be scheduled for handshaking.*

Namely, P_i will be in the premise of a COM *or* CLOSE *rule. We will denote by $\mathcal{P}$ the class of proper adversaries.*

Note that the above definition is weaker than the notion of fair scheduler, which requires that *any* process which is ready infinitely often will eventually be scheduled for execution. Clearly, the fairness assumption would be sufficient for our encoding, however it is not necessary.

In Table 4.2 we showed a solution to the electoral problem in the symmetric network of Figure 4.4. We rewrite this solution as shown in Table 4.3, in order to simplify the notation used for proving the correctness of the algorithm.

We consider the following sets of states, where the states belonging to the same set are symmetric. Two states s_1 and s_2 are symmetric if the probabilistic automata having s_1 and s_2 as start states are isomorphic, in the sense that they differ only for the labels of the nodes (namely the states). In the following, we use the notation $\nu(P) = \nu x_1 \nu x_2 \ldots \nu x_n(P)$, where $x_1,\ x_2 \ldots x_n$ are the free variables of P.

$$
\begin{aligned}
I_0 &= \{\nu(P_0 \mid P_1)\} \\
I_1 &= \{\nu(\bar{x}_0\langle t\rangle \mid A_{00} \mid P_1),\ \nu(\bar{x}_0\langle t\rangle \mid A_{01} \mid P_1), \\
&\qquad \nu(P_0 \mid \bar{x}_1\langle t\rangle \mid A_{10}),\ \nu(P_0 \mid \bar{x}_1\langle t\rangle \mid A_{11})\} \\
I_2 &= \{\nu(B_{00} \mid P_1),\ \nu(\bar{x}_0\langle t\rangle \mid B_{01} \mid Q_1),\ \nu(P_0 \mid B_{11}),\ \nu(Q_0 \mid \bar{x}_1\langle t\rangle \mid B_{10})\} \\
I_3 &= \{\bar{x}_0\langle t\rangle \mid A_{00} \mid \bar{x}_1\langle t\rangle \mid A_{10},\ \bar{x}_0\langle t\rangle \mid A_{01} \mid \bar{x}_1\langle t\rangle \mid A_{11}\} \\
I_4 &= \{\bar{x}_0\langle t\rangle \mid A_{01} \mid \bar{x}_1\langle t\rangle \mid A_{10},\ \bar{x}_0\langle t\rangle \mid A_{00} \mid \bar{x}_1\langle t\rangle \mid A_{11}\}
\end{aligned}
$$

I_1 is the set of states where one of the two processes has committed to a channel. I_2 represents the set of states where one of the two processes has performed its first input. I_3 is the set of states where P_0 and P_1 have committed to the same channel and I_4 is the set of states where P_0 and P_1 have committed to different channels.

The proof is formalized in terms of progress statements defined in [47] and reviewed in Chapter 3. Recall that a progress statement is denoted by $I \xrightarrow[p]{A} I'$, where I and I' are sets of states, p is a probability, and A is a class of adversaries. Its meaning is that starting from any state in I, under any adversary in A, a state of I' is reached with

$$
\begin{array}{rcl}
P_i & = & \bar{x}_i\langle t\rangle |\, Q_i \\
Q_i & = & 1/2\ \tau.A_{i0} \\
 & & + \\
 & & 1/2\ \tau.A_{i1} \\
A_{ij} & = & x_j(b).B_{ij} \\
B_{ij} & = & \mathit{if}\ b \\
 & & \mathit{then}\ (\ (1-\varepsilon)\ x_{j\oplus 1}(b).C_i \\
 & & \qquad + \\
 & & \qquad \varepsilon\ \tau.(\bar{x}_j\langle t\rangle \mid Q_i)) \\
 & & \mathit{else} D_i \\
C_i & = & \bar{o}_i\langle i\rangle \mid \bar{x}_i\langle f\rangle \mid \bar{x}_{i\oplus 1}\langle f\rangle \\
D_i & = & \bar{o}_i\langle i\oplus 1\rangle
\end{array}
$$

Table 4.3. A π_{pa} solution for the electoral problem in the symmetric network of Figure 4.4. Here $i, j \in \{0, 1\}$ and $\oplus$ is the sum modulo 2.

probability at least p. We drop A from the notation of progress statements whenever the class of adversaries is fixed and is clear from the context.

We first show that $I_0 \cup I_1 \cup I_2 \xrightarrow[(1-\epsilon)/2]{A} S$, where S is the set of states from where the leader is elected with probability 1 and A is a proper class of adversaries.

Proposition 4.2. $I_0 \xrightarrow[(1-\epsilon)/2]{} S$.

Proof For the sake of simplicity, we assume that starting from $\nu(P_0 \mid P_1)$ the scheduler chooses the transition group corresponding to the activation of P_0, namely $\{\xrightarrow[1/2]{\tau} \nu(\bar{x}_0\langle t\rangle \mid A_{00} \mid P_1), \xrightarrow[1/2]{\tau} \nu(\bar{x}_0\langle t\rangle \mid A_{01} \mid P_1)\}$. Without loss of generality we can consider the transition to the state $\nu(\bar{x}_0\langle t\rangle \mid A_{00} \mid P_1)$. At this point there are two possible transition groups: $\{\xrightarrow[1]{x_0(b)} \nu(B_{00} \mid P_1)\}$ and $\{\xrightarrow[1/2]{\tau} \nu(\bar{x}_0\langle t\rangle \mid A_{00} \mid \bar{x}_1\langle t\rangle \mid A_{10}), \xrightarrow[1/2]{\tau} \nu(\bar{x}_0\langle t\rangle \mid A_{00} \mid \bar{x}_1\langle t\rangle \mid A_{11})\}$, corresponding to the activation of A_{00} and P_1 respectively.

Let us consider the case corresponding to the first transition group $\{\xrightarrow[1]{x_0(b)} \nu(B_{00} \mid P_1)\}$. By scheduling B_{00} in $\nu(B_{00} \mid P_1)$ the leader is elected with probability $1-\epsilon$. On the other hand, if P_1 is scheduled, then P_1 commits to the same channel as P_0 with probability $1/2$. In this case P_1 cannot perform its second input and P_0 is elected leader with probability $1-\epsilon$. Therefore, starting from $\nu(\bar{x}_0\langle t\rangle \mid A_{00} \mid P_1)$ the leader is elected under these adversaries with probability at least $(1-\epsilon)/4$. The priority imposed on the adversary is important here to ensure that a process trying to perform its second input is not restarted because the adversary did not trigger the corresponding output action.

If, starting from $\nu(\bar{x}_0\langle t\rangle \mid A_{00} \mid P_1)$, P_1 is scheduled, then the processes commit to the same channel with probability $1/2$ and the leader is elected with probability $1-\epsilon$ no matter which process is scheduled first. Therefore under these adversaries the leader is elected with probability at least $(1-\epsilon)/4$.

Since we considered all possible adversaries in A we can conclude that starting from $\nu(\bar{x}_0\langle t\rangle_0 \mid A_{00} \mid P_1)$ the leader is elected with probability at least $(1-\epsilon)/4$.

Consider now the other possible transition from $\nu(P_0 \mid P_1)$, leading to the state $\nu(\bar{x}_0\langle t\rangle \mid A_{01} \mid P_1)$. This state gives rise to an isomorphic execution tree. We can conclude that the probability of reaching a state in S from a state in I_0 is at least $(1-\epsilon)/4 + (1-\epsilon)/4 = (1-\epsilon)/2$. Hence, $I_0 \xrightarrow[(1-\epsilon)/2]{} S$. □

Proposition 4.3. $I_1 \xrightarrow[(1-\epsilon)/2]{} S$.

Proof The proof is analogous to the one of Proposition 4.2. □

Proposition 4.4. $I_2 \underset{(1-\epsilon)/2}{\longrightarrow} S$.

Proof The proof is analogous to the one of Proposition 4.2. □

Proposition 4.5. $I_0 \cup I_1 \cup I_2 \underset{(1-\epsilon)/2}{\longrightarrow} S$.

Proof By combining the progress statements proved in Proposition 4.2, 4.3 and 4.4 according to Lemma 3.1. □

Next, we show that $I_0 \cup I_1 \cup I_2$ *unless* S. This means that the automaton corresponding to the process $\nu(P_0 \mid P_1)$ remains in a state belonging to $I_0 \cup I_1 \cup I_2$ until the leader is elected.

Proposition 4.6. $I_0 \cup I_1 \cup I_2$ *unless* S.

Proof We have to show that for each transition from a state $s \in I_0 \cup I_1 \cup I_2 - S$ the probabilistic automaton corresponding to the process $\nu(P_0 \mid P_1)$ moves to a state $s' \in I_0 \cup I_1 \cup I_2 \cup S$. In other words, once the probabilistic automaton is in a state in $I_0 \cup I_1 \cup I_2$ it remains in $I_0 \cup I_1 \cup I_2$ until it reaches a state in S.

Let s be a state of the probabilistic automaton such that $s \in I_0 \cup I_1 \cup I_2 - S$. Then $s \in I_0 \cup I_1 \cup I_2$ and $s \notin S$.

We first analyze the case $s \in I_0$. Following a simple analysis of the tree of the probabilistic automaton corresponding to the process $\nu(P_0 \mid P_1)$ we find that any transition from I_0 leads to I_1, which means that the probabilistic automaton remains in $I_0 \cup I_1 \cup I_2$.

Next we study the behavior of the probabilistic automata for $s \in I_2$. Recall that I_2 corresponds to the states where one process performed its first input and the other has not been scheduled for execution. If the process that holds one channel, say P_0 is activated next, it will find the second channel available and depending on the probabilistic choice will go to a state in S or I_0. If on the contrary P_1 is scheduled and commits to the same channel as the one that P_0 holds, then P_1 will be suspended and P_0 will declare itself to be the leader and therefore go to a state in S or backtrack and go to a state in I_1. The last possible transition from I_2 corresponds to the activation of

P_1 and leads to a state where P_1 committed to the channel that is not held by P_0. If P_0 is scheduled again, since its second input can be performed, it will go to a state in S or backtrack and go to a state in I_1. In case P_1 performs its first input, no process is able to perform its second input after that because they block each other and by backtracking they go back to a state in I_2.

Therefore any transition from I_2 goes to a state in $I_0 \cup I_1 \cup I_2 \cup S$, which means that the probabilistic automaton either remains in $I_0 \cup I_1 \cup I_2$ or reaches S.

The last case to consider is $s \in I_1$. Following again a simple analysis of the tree of the probabilistic automaton we find that any transition from I_1 leads to a state in $I_2 \cup I_3 \cup I_4$.

We now consider the reachable states from I_3 and I_4. Since I_3 contains the states where both processes committed to the same channel, no matter what policy the scheduler uses next one process is able to perform its second input while the other one is blocked waiting for its first input. Hence from a state in I_3 the next state is either in S or, if the process backtracks, in I_1. In other words the probabilistic automaton remains in I_1 unless it reaches S.

I_4 contains processes committed to different channels. By analyzing all states that can be reached from I_4 we find that either a leader is elected or the processes must backtrack. Therefore any transition from I_4 leads the probabilistic automaton to a state in $I_1 \cup I_2$ or S. □

Theorem 4.7. *Consider the process $\nu(P_0 \mid P_1)$ and the algorithm of Table 4.3. The probability that the leader is eventually elected is* 1 *under every proper adversary.*

Proof By Proposition 4.5 $I_0 \cup I_1 \cup I_2 \underset{(1-\epsilon)/2}{\longrightarrow} S$. By Proposition 4.6 the probabilistic automaton corresponding to the process $\nu(P_0 \mid P_1)$ satisfies $I_0 \cup I_1 \cup I_2$ *Unless* S. Thus, Proposition Lemma 3.3 applies, leading to: $I_0 \cup I_1 \cup I_2 \underset{1}{\longrightarrow} S$. □

We conclude this section with the observation that, if we modify the blind choice to be a choice prefixed with the input actions which come immediately afterward, then the above theorem would not hold anymore. In fact, we can define a scheduler which selects the processes in alternation, and which suspends a process, and activates the other, immediately after the first has made a random choice and performed an input. The latter will be forced (because of the guarded choice) to perform the input on the

other channel. Then the scheduler will proceed with the first process, which at this point can only backtrack. Then it will schedule the second process again, which will also be forced to backtrack, and so on. Since all the choices of the processes are obligated in this scheme, the scheduler will produce an infinite (unsuccessful) execution with probability 1.

Chapter 5

The generalized dining philosophers

So far we have defined the probabilistic asynchronous π-calculus and we have shown how the problem of electing a leader in a symmetric network, which has no solution in the asynchronous π-calculus, can be solved with the probabilistic asynchronous π-calculus. We introduced π_{pa} with the goal of providing a distributed implementation for the π-calculus, since so far only the asynchronous subset of the π-calculus has been implemented [41]. We are considering a randomized implementation since it has been shown that the full π-calculus is strictly more expressive than its asynchronous subset, and there is no hope of implementing the π-calculus with deterministic methods ([36]). Moreover, in [29] Nestmann has shown that the gap in expressive power, and the difficulty in the implementation with deterministic methods, is due to the mixed guarded choice construct of the π-calculus. Such mechanism, however, would be very desirable as it provides a powerful programming primitive for solving distributed conflicts.

By investigating the translation from π to π_{pa} we found out that it requires solving a resource allocation problem similar to the one of the generalized dining philosophers, where the resources of the generalized dining philosophers problem correspond to the channels of the π-calculus. Hence an algorithm for the generalized dining philosophers problem could be used for solving the conflicts associated to the competition for channels arising in presence of guarded-choice commands.

We consider a generalization of the dining philosophers problem to arbitrary connection topologies. We focus on symmetric, fully distributed systems, and we address the problem of guaranteeing progress and lockout-freedom, even in presence of adversary schedulers, by using randomized algorithms. We first show that the well-known algorithms of Lehmann and Rabin do not work in the generalized case, and then we propose an alternative algorithm based on the idea of letting the philosophers assign a random priority to their adjacent forks. The results showed in this chapter serve to some extent to prove the correctness of the randomized and distributed encoding of the π-calculus with mixed choice in Chapter 6. However, the algorithms GDP1 and GDP2 presented in this chapter for solving the generalized dining philosophers problem require

the fairness assumption. Our goal is to provide an encoding that is robust even in the presence of unfair schedulers. For this reason, the algorithms GDP1 and GDP2 are for general interest only. The algorithms are part of the original paper on the generalized dining philosophers ([12]) and we include them in this chapter in order to give a complete solution to the generalized dining philosophers problem.

We start the chapter with an overview of the dining philosophers problem.

5.1 The dining philosophers problem

The problem of the dining philosophers, proposed by Dijkstra in [6], is a very popular example of control problem in distributed systems, and has become a typical benchmark for testing the expressiveness of concurrent languages and of resource allocation strategies.

The typical dining philosophers sit at a round table in positions alternated with forks, so that there is a fork between each two philosophers, and a philosopher between each two forks. Each philosopher can pick up only the forks immediately to his right and to his left, one at the time, and needs both of them to eat. The aim is to make sure that if there are hungry philosophers then some of them will eventually eat (*progress*), or, more ambitiously, that every hungry philosopher will eventually eat (*lockout-freedom*).

The solutions to the problem of the dining philosophers depend fundamentally on the assumptions made on the system. If we do not impose an initial symmetry, or do not impose that the system be completely distributed, then several solutions are possible. Some examples are:

- The forks are ordered and each philosopher tries to get first the adjacent fork which is higher in the ordering.
- The philosophers are colored yellow and blue alternatively. The yellow philosophers try to get first the fork to their left. The blue ones try to get first the fork to their right.
- There is a central monitor which controls the assignment of the forks to the philosophers.
- There is a box with $n-1$ tickets, where n is the number of the philosophers, and each philosopher must get a ticket before trying to get the forks.

In the first two solutions above, the system is not symmetric. In the last two, it is not fully distributed.

Of course, the problem becomes much more challenging when we impose the conditions of symmetry and full distribution. More precisely, symmetry means that the philosophers are indistinguishable, as well as the forks. The philosophers run the same program, and both the forks and the philosophers are all in the same initial state. Full distribution means that there are no other processes except the philosophers, there is no central memory, all philosophers run independently, and the only possible interaction is via a shared fork.

The conditions of symmetry and full distribution are interesting also for practical considerations: in several cases it is desirable to consider systems which are made of copies of the same components, and have no central control or shared memory. In particular, symmetry offers advantages at the level of reasoning about the system, as it allows a greater modularity, and at the level of implementation of concurrent languages, as it allows a compositional compilation. Full distribution is usually convenient as it avoids the overhead of a centralized control.

Lehmann and Rabin have shown in [42] the remarkable result that there are no *deterministic* solutions to the dining philosophers problem, if symmetry and full distributions are imposed, and if no assumption (except fairness) are made on the scheduler. The only possible solution, in such conditions, are *randomized* algorithms, that allow to eventually break the initial symmetry with probability 1. In [42] two such algorithms are proposed, the first guarantees progress, the second guarantees also lockout-freedom.

There are two proofs of correctness of the Lehmann and Rabin algorithms, one in [42] and another one, more structured and formal, in [23, 47]. They both depend in an essential way on the topology. In particular, they depend on the fact that one fork can only be shared by two philosophers (cfr. Lemma 1 in [42], Lemma 7.13 in [23], and Lemma 6.3.14 in [47]). Therefore, a question naturally arises: Would the solutions of Lehmann and Rabin still work in the case of more general connection structures? The problem is also of practical relevance, since the kind of resource network represented by the classic formulation is very restricted. In the rest of this chapter we investigate this question and show that the answer is no: In most situations, both the algorithms of Lehmann and Rabin fail. We then propose another solution, still randomized but based on a rather different idea: we let each philosopher try to establish a partial order on forks, by assigning a random number to his adjacent forks. In other words, we use randomization for breaking the initial symmetry and achieving a situation in which

the forks are partially ordered. Finally, we propose a variant of the algorithm which ensures that no philosopher will starve (lockout-freedom). The algorithms are robust with respect to every fair scheduler.

5.2 The generalized dining philosophers problem

We now introduce a generalization of the dining philosophers problem. The generalization consists in relaxing the assumptions about the topology of the system. In the classic problem the philosophers and the forks are distributed along a ring (table) in alternated positions. On the contrary, we consider arbitrary connection topologies, and in particular we admit the possibility that a fork is shared by more than two philosophers. Thus the number of forks and the number of philosophers is not necessarily the same. The only constraint we impose on the topology is that each philosopher is connected (has access) to two distinct forks. For the rest, the new formulation coincides with the classic one.

Definition 5.1. *A generalized dining philosopher system consists of $n \geq 1$ philosophers and $k \geq 2$ forks. Unlike the classic case, n and k may be different numbers, and a fork can be shared by an arbitrary (positive) number of philosophers. Like in the classic case, every philosopher has access to two forks, which he will refer to as left and right. Every philosopher can think or eat. When a philosopher wants to eat he must pick up the two forks. He can pick up only one fork at the time. He cannot pick up a fork if his neighbor is already holding it. He cannot eat forever. After eating the philosopher releases the two forks and resumes thinking.*

Figure 5.1 shows some examples of generalized dining philosopher systems. We represent a system as an undirected graph where the nodes are the forks (represented by sticks in Figure 5.1), and the arcs are the philosophers (represented by circles in Figure 5.1). Obviously, the forks accessible to a philosopher are the adjacent nodes. Note that we adopt the more general definition of graph, which allows the presence of more than one arc between two nodes (some textbooks use the term *multigraph*).

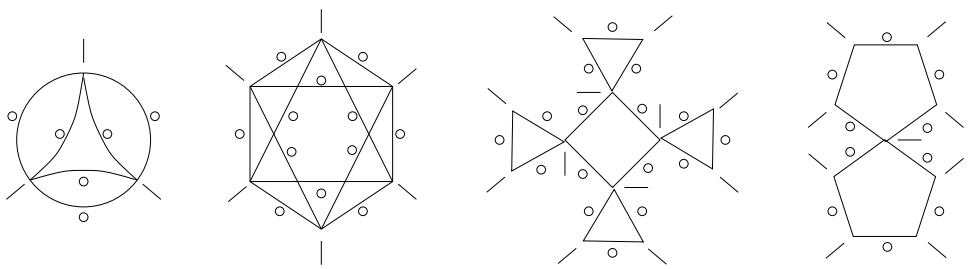

Fig. 5.1. Some examples of generalized dining philosophers. From left to right: 6 philosophers, 3 forks. 12 philosophers, 6 forks. 16 philosophers, 12 forks. 10 philosophers, 9 forks.

The goal is the same as in the classic problem: to program the philosophers so that hungry philosophers will eventually eat. Following the standard terminology, we will say that a solution ensures progress (with respect to a set of philosophers) if it guarantees that, whenever a philosopher of the set is hungry, then a philosopher of the same set (not necessarily the same philosopher) will eventually eat[1]. A solution is *lockout-free* (with respect to a set of philosophers) if it guarantees that, whenever a philosopher of the set is hungry, then the same philosopher will eventually eat.

We shall consider only *fully distributed* and *symmetric solutions*, namely algorithms where the only processes are the philosophers, the only shared variables are the forks, all philosophers run identical programs and both the philosophers and the forks are in the same initial state. We assume that *test-and-set* operations on the forks are performed atomically.

A computation consists in an interleaving of actions performed by the philosophers. Such interleaving is controlled by an *adversary* (or *scheduler*). We assume that the adversary has complete information of the past of the computation, and can decide its next step on the basis of that information. We consider only *fair* adversaries, namely adversaries that ensure that each philosopher executes infinitely many actions in each of the possible computations.

We will consider *randomized algorithms*, namely algorithms which allow a philosopher to select randomly between two or more alternatives. The outcome of the random

[1] In [42] this property is called *deadlock-freedom*.

choice depends on a probability distribution, and it is not controlled by the adversary. For this reason, even under the same adversary, different computations may be possible. This model of computation has been formalized by Lynch and Segala by introducing the concept of probabilistic automata. An overview of this model was given in Chapter 3.

5.3 Limitations of the algorithms of Lehmann and Rabin

In this section we show that the randomized algorithms of Lehmann and Rabin presented in [42] do not work anymore in the general case.

We start by recalling the first algorithm of Lehmann and Rabin, LR1 for short. Each philosopher runs the code written in Table 5.1.

1. *think*;
2. $fork := random\_choice(left, right)$;
3. *if* $isFree(fork)$ *then* $take(fork)$ *else goto* 3;
4. *if* $isFree(other(fork))$
   then $take(other(fork))$
   else $\{release(fork);$ *goto* $2\}$
5. *eat*;
6. $release(fork);\ release(other(fork))$;
7. *goto* 1;

Table 5.1. The algorithm LR1.

Following standard conventions we assume that the action *think* may not terminate, while all the other ones are supposed to terminate. The test-and-set operations on the forks, in Steps 3 and 4, are supposed to be executed atomically. Each outcome (left or right) of the random draw has a positive probability and the sum of the probabilities

is 1. In the classic algorithm the probability is evenly distributed (1/2 for left and 1/2 for right). However our negative results do not depend on this assumption.

It has been shown in [42] that, for the classic dining philosophers, LR1 ensures progress with probability 1 under every fair scheduler. A more formal proof of this result can be found in [23, 47].

In the generalized case this result does not hold anymore. Let us illustrate the situation with an example. We use the following notation: An empty arrow, associated with a philosopher and pointing towards a fork, denotes that the philosopher has committed to that fork (has selected that fork with the random choice instruction) but he has not taken it yet. A filled arrow denotes that the philosopher is holding the fork, namely he has taken the fork and has not released it yet. From now on, we will represent the nodes (forks) as bullets, instead than as sticks.

For the sake of simplicity, for the moment we relax the fairness requirement. We will discuss later how to make the example valid also in presence of fairness.

Consider the system on the leftmost side of Figure 5.2, and consider State 1 depicted in the figure below. Clearly, this state is reachable from the initial state (where all philosophers are at the beginning of the program, i.e. thinking) with a non-null probability.

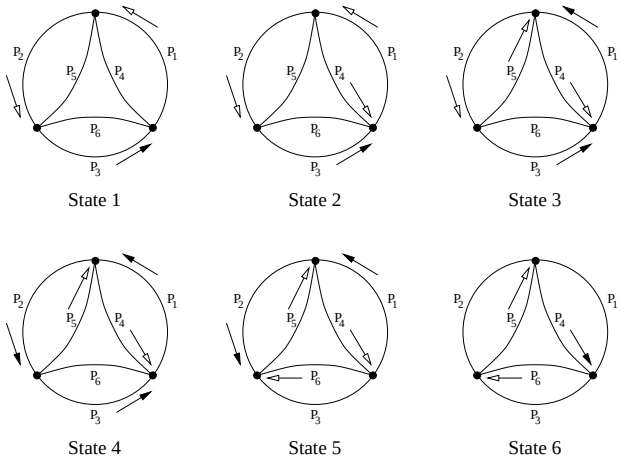

Fig. 5.2. A winning scheduling strategy against the algorithm LR1

The scheduler chooses P_4 next. If P_4 commits to the free fork, then he will take it and then try to get the other fork. Since the other fork is taken by P_3, P_4 has to release the first fork and draw again. The scheduler keeps selecting P_4 until he commits to the fork taken by P_3. When this commitment occurs we are in State 2.

Next the scheduler selects P_1 and P_1 takes the fork he had committed to. Then the scheduler keeps scheduling P_5 until P_5 commits to the fork taken by P_1 (like it was done for P_4). This is State 3.

Then the scheduler selects P_2, and P_2 takes the fork he had committed to. This situation is represented by State 4.

The scheduler continues with P_3. P_3 finds his second fork taken by P_2 and therefore releases the fork that he currently controls. P_6 is then scheduled, until it commits to the fork taken by P_2. This is State 5.

Finally the scheduler runs P_2, and P_2 will have to release his fork, since the other fork is taken by P_1. Then P_4 is selected, and he takes the fork he had committed to. Then the scheduler selects P_1, which will have to release his fork since the other one is taken by P_4. This is State 6.

Observe now that State 6 is isomorphic to State 1, in the sense that they differ only for the names of the philosophers. The scheduler can then go back to State 1 and then repeat these steps forever, thus inducing a computation in which no philosopher is able to eat. Note that the probability of a computation of this kind is 1/4, which is the probability of reaching a state isomorphic to State 1 already at the first attempt. (We are assuming that the probability of picking a particular adjacent fork is evenly distributed between left and right, i.e. it's 1/2. If this is not the case then the figure above will be different from 1/4, but it will still be positive.) It's easy to see that, by repeating the attempt to reach State 1 (possibly after some philosopher has eaten), the scheduler can *eventually* induce a cycle like the above one with probability 1.

Unfortunately the scheduler considered in this example is unfair. In fact, it keeps selecting one philosopher (for instance P_4) until it commits to a taken fork. If the philosopher chooses forever the free fork, then the resulting computation is unfair. Although such a computation has probability 0, according to the definition of fairness, the scheduler is unfair.

However, it is easy to modify the scheduler so to obtain a fair scheduler which achieves the same result, namely a no-progress computation with non-null probability, although smaller than 1/4. Consider a variant of the above scheduler which keeps selecting a "stubborn philosopher" for a finite number of times only, but which increases

this number at every round. By "round" here we mean the computation fragment which goes from State 1 to State 6, and back to State 1, as described above. Let n_k be the maximum number of times which the scheduler is allowed to select the same philosopher during the k-th round. Choose n_k to be big enough so that the probability that the scheduler actually succeeds to complete the k-th round is $1 - p^k$ with $p \leq 1/2$. Consider an infinite computation made of successive successful rounds. The probability of this computation is greater than or equal to

$$\frac{1}{4}\,\Pi_{k=1}^{\infty}(1 - p^k).$$

It is easy to prove by induction that for every $m \geq 1$,

$$\Pi_{k=1}^{m}(1 - p^k) \geq 1 - p - p^2 + p^{m+1}$$

holds. Hence we have

$$\Pi_{k=1}^{\infty}(1 - p^k) \geq 1 - p - p^2.$$

Furthermore, by the assumption $p \leq 1/2$, we have

$$1 - p - p^2 \geq 1/4.$$

5.3.1 A general limitation to the first algorithm of Lehmann and Rabin

We have seen that there is at least one example of graph in which LR1 does not work. One could hope that this example represents a very special situation, and that under some suitable conditions LR1 could still work in more general cases than just the standard one. Unfortunately this is not true: It turns out that as soon as we allow one fork of the ring to be shared by an additional philosopher, LR1 fails.

In the following, we will call *ring* (or *cycle*) a graph which has k nodes, say $0, 1, \ldots k-1$, and k arcs connecting the pairs $(0,1), (1,2), \ldots, (k-1, 0)$.

Theorem 5.1. *Consider a graph G containing a ring subgraph H, and such that one of the nodes of H has at least three incident arcs (i.e. an additional arc in G besides the two*

in H). Then it is possible to define a fair scheduler for LR1 *such that the probability of a computation in which the arcs (philosophers) in H make no progress is strictly positive.*

Proof Figure 5.3 represents the subgraph of G consisting of

- the ring H (a hexagon in the figure, but the number of vertices is not important) with a node f having (at least) three incident arcs,
- the arc P in G but not in H which is incident on f, and
- the node g adjacent to f via P.

It does not matter whether g is a node in H or not.

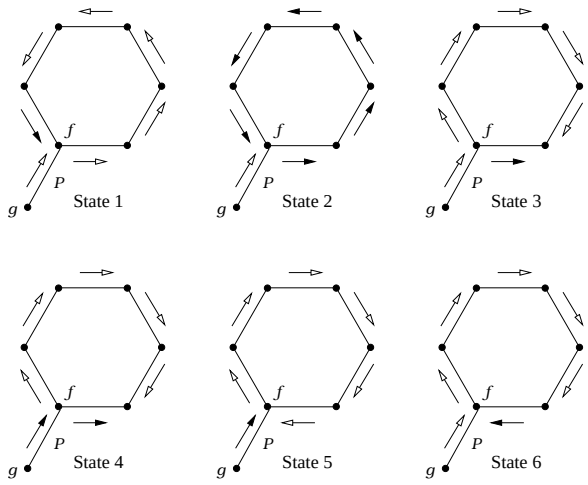

Fig. 5.3. A winning scheduling strategy against the algorithm LR1.

Figure 5.3 shows a possible sequence of states induced by a scheduler S. State 1 is reachable from the initial state (where all philosophers are thinking) with a non-null probability. The scheduler controls the directionality of the arrows by means of the technique explained in the example in previous section. The state transitions should be

rather clear, except maybe for the last one. That transition (between State 5 and State 6) is achieved by the following sequence of actions:

- schedule P and let him eat (this is always possible - the scheduler can always make g free at the moment P needs it)
- schedule the philosopher adjacent to P which is committed to f, and let him take f
- keep scheduling P until he commits to f.

State 6 is symmetric to State 1 and we can therefore define an infinite computation, where no philosopher in H eats, by repeating the actions which bring from State 1 to State 6, and then back to State 1, and so on.

Again, the scheduler S illustrated here is not fair, but we can obtain a fair scheduler S' which approximates S by letting the "level of stubbornness" of S' increase at each round, following the technique used in the example above. □

5.3.2 The second algorithm of Lehmann and Rabin

In this section we consider the second algorithm of Lehmann and Rabin, presented in [42] as a lockout-free solution to the classic dining philosophers.

We consider here a slight generalization of the original algorithm suitable for the a generic topology. Hereafter we will refer to it as LR2. We assume that each fork is provided with the following data structures:

- A list of incoming requests r, with operations *isEmpty*, *insert*, and *remove*. Initially the list is empty.
- A "guest book" g, namely a list which keeps track of the philosophers who have used the fork.

The idea is that when a philosopher gets hungry, he inserts his name *id* in the request list of the adjacent forks[2]. After the philosopher has eaten, he removes his name from these lists, and signs up the guest books of the forks. Before picking up a fork, a

[2]We do not need to assume that all philosophers have different *ids*, but simply that those who share a fork are distinguished form each other. This assumption does not violate the symmetry requirement. In fact, the distinction between the adjacent philosophers could be stored in the fork and used only within the operations on the fork.

philosopher must check that there are no other incoming requests for that fork, or that the other philosophers requesting the fork have used it after he did. This condition will be represented, in the algorithm, by the condition *Cond*(*fork*)[3].

Table 5.2 shows the code run by each philosopher.

The negative result expressed in Theorem 5.1 does not hold for LR2. In fact, once P has eaten, he cannot take Fork f before the neighbor has eaten as well. However the class of graphs in which LR2 does not work is still fairly general:

Theorem 5.2. *Consider a graph G containing a ring subgraph H, and such that two of the nodes in H are connected at least by three different paths (i.e. an additional path P in G besides the two in H). Then it is possible to define a fair scheduler for* LR2 *such that the probability of a computation in which the arcs (philosophers) of H and P make no progress is strictly positive.*

Proof The proof is illustrated in Figure 5.4, which shows the part of G containing the ring H and the additional path between two nodes of H.

Like before, the computation illustrated in the figure is induced by an unfair scheduler S, but by following the usual technique we can define a fair scheduler S' which approximates S and which achieves the same result. Note that none of the philosophers in H and in the additional path ever gets to eat, hence the modification of LR2 with respect to LR1, namely the test *Cond*(*fork*), is useless: *fork.g* remains forever empty. □

5.4 A deadlock-free solution

We now propose a symmetric and fully distributed solution to the generalized dining philosophers and show that it makes progress with probability 1.

Our algorithm works as follows. Each fork has associated a field *nr* which contains an integer number ranging in the interval $[0, m]$, with $m \geq k$, where k is the total number of forks in the system. Initially *nr* is 0 for all the forks. Each philosopher can change the *nr* value of a fork when he gets hold of it, and he tries to make sure that the *nr* values

[3] In the original algorithm the list of incoming request is replaced by two switches associated to the two adjacent philosophers, and instead of the guest book there is a simple variable which indicates which one of the adjacent philosophers has eaten last.

```
1.   think;
2.   insert(id, left.r); insert(id, right.r);
3.   fork := random_choice(left, right);
4.   if isFree(fork) and Cond(fork)
         then take(fork)
         else goto 4;
5.   if isFree(other(fork))
         then take(other(fork))
         else {release(fork); goto 3;}
6.   eat;
7.   remove(id, left.r); remove(id, right.r);
8.   insert(id, left.g); insert(id, right.g);
9.   release(fork); release(other(fork));
10.  goto 1;
```

Table 5.2. The algorithm LR2.

of its adjacent forks become and are maintained different. In order to ensure that this situation will be eventually achieved, each new *nr* value is chosen *randomly*. Note that this random choice is necessary to break the symmetry, otherwise, in presence of a ring, a malicious scheduler could induce a situation where one philosopher changes one fork, then his neighbor changes the other fork to the same value, and so on, for all the forks in the ring.

Our algorithm is similar to LR1, except that the choice of the first fork is done by picking the one with the highest *nr* value (if they are different), instead than randomly. The other difference is that, as explained before, the philosopher may change the *nr* value of a fork when it finds that it is equal to the *nr* value of the other fork. This is done by calling $random[1, m]$, which returns a natural number in the interval $[1, m]$, selected probabilistically. We assume for simplicity that the probability of the outcome is evenly distributed among the numbers in the interval. The algorithm is illustrated in Table 5.3. We will refer to it as GDP1.

We prove now that GDP1 makes progress, under every fair scheduler, with probability 1. The proof is formalized in terms of the *progress* and *unless* statements. Recall that a progress statement is denoted by $S \xrightarrow[p]{A} S'$, where S and S' are sets of states, p is a probability, and A is a class of adversaries. An unless statement is of the form S *unless* S' and means that, if the system is in one of the states of S, then it remains in S (possibly moving through different states of S) until it reaches a state in S'.

Let us define T to be the set of states in which some philosopher tries to eat (*trying section*, steps 2 through 5), and E to be the set of states in which some philosopher is eating. We prove the correctness of GDP1 by showing that $T \xrightarrow[1]{F} E$, where F is the class of all fair adversaries.

Theorem 5.3. $T \xrightarrow[1]{F} E$.

Proof Let us denote by C_1 the set of states in which there is one cycle in the graph where all adjacent forks have different numbers. C_2 is the set of states in which there are two cycles in the graph where all adjacent forks have different numbers, and so on.

We have the following progress statements:

- $T \xrightarrow[p]{F} (T \cap C_1) \cup E$, with $p \geq m!/m^k(m-k)!$. In fact, one of the trying philosophers, say P_1, will find the first fork free and will pick it up. Then, either he will find also his second fork free, and therefore will eat, or it will find the second fork taken by another philosopher, say P_2. Again, either P_2 will eat, or will find

1. *think*;
2. *if* $left.nr > right.nr$
   then $fork := left$
   else $fork := right$;
3. *if* $isFree(fork)$ *then* $take(fork)$ *else goto* 3;
4. *if* $fork.nr = other(fork).nr$
   then $fork := random[1, m]$;
5. *if* $isFree(other(fork))$
   then $take(other(fork))$
   else $\{release(fork);\ goto\ 2\}$
6. *eat*;
7. $release(fork);\ release(other(fork))$;
8. *goto* 1;

Table 5.3. The algorithm GDP1.

his second fork taken by another philosopher, say P_3, etc. Since the number of philosophers is finite, we will end up either with one philosopher eating, or with a ring of forks all picked up as first forks at least once. Since each philosopher changes the nr value of the first fork if this value is equal to that of the other fork, the adjacent forks of this ring will get all different values with probability p not smaller than $m!/m^k(m-k)!$ (this is the probability that, if we assign randomly values in the range $[1, m]$ to the nodes of a complete graph of cardinality k, all the nodes get a different value). Note that, by the assumption $m \geq k$, we have $p > 0$.

- $T \cap C_1 \xrightarrow[p]{F} (T \cap C_2) \cup E$. Similar to previous point.
- ...
- $T \cap C_{h-1} \xrightarrow[p]{F} (T \cap C_h) \cup E$. Similar to previous point.
- $T \cap C_h \xrightarrow[1]{F} E$. When all possible cycles in the graph have adjacent nodes with different nr values, then the algorithm works like a hierarchical resource allocation algorithm based on a partial ordering: Let P be the first philosopher who is holding the first fork, and such that the nr value of the other fork f is the smallest of all the forks adjacent to f. Then either P or one of his neighbors will eat.

From the above statements, and by using Lemma 3.2, the obvious fact that $E \xrightarrow[1]{F} E$, and Lemma 3.1, we derive

$$T \xrightarrow[p^h]{F} E.$$

On the other hand, it's clear that, since philosophers keep trying until they eat, we have also

$$T \textit{ unless } E.$$

Therefore, by applying Lemma 3.3, we conclude $T \xrightarrow[1]{F} E$. □

Note that GDP1 does not guarantee that we will reach, with probability 1, a situation where all adjacent forks will have a different nr. Not even if all philosophers are in the trying section infinitely often. This is because some philosophers may never succeed to pick up a fork, for instance because they are always scheduled when their neighbors are eating.

5.5 A lockout-free solution

The algorithm GDP1 presented in the previous section is not lockout-free. In fact, consider two adjacent philosophers, P_1 and P_2, which share a fork f with a nr value which is smaller than the value of the other fork g of P_1. Then P_1 will keep selecting g as first fork, and the scheduler could keep scheduling the attempt of P_1 to pick the second fork, f, only when f is held by P_2.

We now propose a lockout-free variant of GDP1. The idea is to associate to each fork a list of incoming requests r, and a "guest book" g, like it was done in Section 5.3.2. The test *Cond*(*fork*) is defined in the same way as in Section 5.3.2. The new algorithm, that we will call GDP2, is shown in Table 5.4.

We now show that GDP2 is lockout-free. In the following, T_i will represent the set of states in which the philosopher P_i is trying to eat, and E_i the situation in which the philosopher P_i is eating.

Theorem 5.4. $T_i \xrightarrow[1]{F} E_i$.

Proof Let us denote by $C_{i,r}$ is the set of states in which there are r cycles containing the arc P_i, and where all adjacent forks have different numbers. Furthermore, let us use $W_{i,s}$ to represent the set of states in which there are s philosophers connected to P_i which have already eaten and can't eat until all their adjacent philosophers (and ultimately P_i) have eaten as well.

The proof is similar to the one of Theorem 5.3. The invariant in this case is

$$T_i \cap C_{i,r} \cap W_{i,s} \xrightarrow[p]{F} \begin{array}{l} (T_i \cap C_{i,r+1} \cap W_{i,s}) \\ \cup \\ (T_i \cap C_{i,r} \cap W_{i,s+1}) \cup E_i \end{array}$$

where p has the same lower bound as in the proof of Theorem 5.3. Furthermore, if h is the total number of cycles containing P_i, and m is the total number of philosophers connected to P_i, we have

$$T_i \cap C_{i,h} \cap W_{i,s} \xrightarrow[p]{F} (T_i \cap C_{i,h} \cap W_{i,s+1}) \cup E_i,$$

1. *think*;
2. *insert*$(id, left.r)$; *insert*$(id, right.r)$;
3. *if* $left.nr > right.nr$
   then $fork := left$
   else $fork := right$;
4. *if* $isFree(fork)$ *and* $Cond(fork)$ *then* $take(fork)$ *else goto* 4;
5. *if* $fork.nr = other(fork).nr$
   then $fork := random[1, m]$;
6. *if* $isFree(other(fork))$
   then $take(other(fork))$
   else $\{release(fork);$ *goto* $3\}$
7. *eat*;
8. *remove*$(id, left.r)$; *remove*$(id, right.r)$;
9. *insert*$(id, left.g)$; *insert*$(id, right.g)$;
10. *release*$(fork)$; *release*$(other(fork))$;
11. *goto* 1;

Table 5.4. The algorithm GDP2.

$$T_i \cap C_{i,r} \cap W_{i,m} \xrightarrow[p]{F} (T_i \cap C_{i,r+1} \cap W_{i,m}) \cup E_i,$$

and

$$T_i \cap C_{i,h} \cap W_{i,m} \xrightarrow[1]{F} E_i.$$

Hence, by Lemma 3.2 and 3.1 we derive

$$T_i \xrightarrow[p^{h+m}]{F} E_i.$$

Since T_i *unless* E_i, by Lemma 3.3 we conclude. □

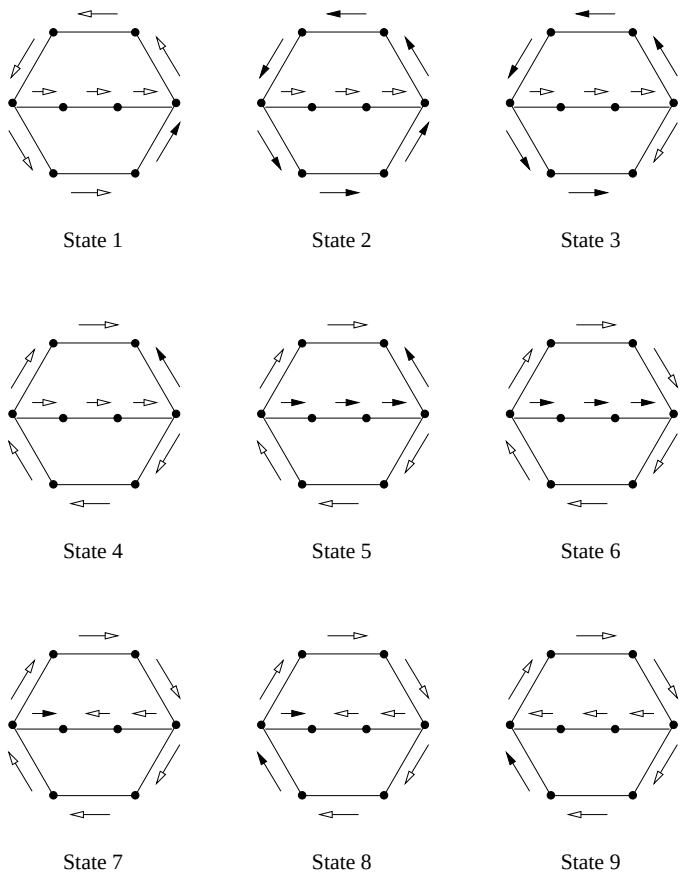

Fig. 5.4. A winning scheduling strategy against the algorithms LR1 and LR2.

Chapter 6

A randomized implementation of the π-calculus with mixed choice

In this chapter we consider the problem of encoding the pi-calculus with mixed choice into the asynchronous pi-calculus via a uniform translation while preserving a reasonable semantics. Since it has been shown that this is not possible with an exact encoding, we define a randomized approach using the probabilistic asynchronous pi-calculus, and we show that our solution is correct with probability 1 under any proper adversary with respect to a notion of testing semantics.

This result establishes the basis for a distributed and symmetric implementation of mixed choice which, differently from previous proposals in literature, does not rely on assumptions on the relative speed of processes and is robust to attacks of proper adversaries.

6.1 An overview of solutions for the binary interaction problem

The distributed implementation of mixed choice, also called the *binary* interaction problem, has been widely investigated, as well as the more general *multiparty* interaction problem. Most of the proposed solutions are asymmetric, see for instance [20, 39, 50], and most of them rely on an ordering among the identifiers of the processes (or equivalently among the nodes of the connection graph). The only symmetric solutions that have been proposed are, not surprisingly, randomized ([10, 43, 19, 18]). However, these algorithms assume a computational model that is different from the asynchronous model used in this work. They rely on assumptions about the relative speed of the processes during the phase in which the processes attempt to establish communication, by implying a global time and assuming that the whole computation time is negligible with respect to the waiting time. Therefore these algorithms work in a partially synchronous model rather than an asynchronous model. Note that these different models also induce a different notion of scheduler.

In the following we briefly recall the algorithm proposed by [10] for CSP-like binary interactions. The one in [19] is an extension of the same algorithm to the case of multiparty interaction. Both algorithms are presented in a shared-memory model where processes communicate by reading from and writing to shared variables. The algorithm of [43] presents a probabilistic distributed solution for the guard scheduling problem, which is a special case of the multiparty interaction scheduling problem where each interaction involves exactly two processes.

We adopt the presentation of [19] which is more rigorous. In the algorithm, each possible binary interaction is associated to a variable which ranges over 0 and 1. The variable can be accessed only by the processes interested in the interaction, via a test-and-set function of the following kind:

$$\texttt{TEST\&SET}(X, op, op')$$

which means: read the value of X. If it is 0, then apply op to X. Otherwise, apply op'. In both cases, return the value of X before the operation. These actions (read and set) are meant to be executed *atomically*, i.e. as an indivisible sequence. Originally, all variables are set to 0.

The code executed by each process interested in interacting is shown in Table 6.1. The idea is that each process P ready to interact chooses randomly one of the possible interactions, and tests the corresponding variable X. If X is 0 then P sets it to 1 and waits for a while. Then P tests X again. If the value has changed (to 0), meaning that the partner has chosen the same interaction, then the interaction is started. Otherwise, P resets X to 0 and tries a new interaction, possibly with a different partner. On the contrary, if at the first test X was 1, then it means that the partner is willing to interact. In this case P sets X to 0 to signal to the partner its positive response, and starts the interaction.

The algorithm in Table 6.1 requires a careful choice of the waiting time a process spends on monitoring the state of its partner. This time, which is called δ, is the time spent by a process in line 5 of the algorithm in Table 6.1 and represents the time a process waits before re-accessing the shared variable to determine if the partner is willing to communicate. In [10] it is assumed that the time to access a shared variable is negligible compared to δ. This assumption is eliminated in [19], but they require that δ

```
1.  while trying do {
2.     randomly choose an interaction. Let X be the associated variable
3.     if  TEST&SET(X, inc, dec) = 1
4.        then participate to the interaction
5.        else { wait for some time;
6.               if  TEST&SET(X, no_op, dec) = 0
7.                  then participate to the interaction
8.                  /* else try another interaction */ }
9.  }
```

Table 6.1. The algorithm for binary interaction proposed in [10]. The notations *inc*, *dec* and *no_op* mean, respectively: add 1, subtract 1, and no operation.

is bounded by some constant, i.e. δ is large enough such that a process will not re-access the shared variable before its partner has a chance to access it too.

Another algorithm for the multiparty interaction scheduling problem is presented in [18]. We refer here to the algorithm TB (Token-Based) of [18]. The second algorithm presented in the same paper is not relevant for this work since it is based on shared-memory.

The algorithm TB uses a mechanism similar to the one in [10, 19] to establish interactions. Each process P_i is associated with a unique token T_i and a set of possible interactions I_i. $P(X)$ denotes the set of processes that can execute an interaction X. A process randomly chooses an interaction X from its set of possible interactions and informs all other processes in $P(X)$ that its choice by sending a copy of T_i tagged with X to each process in $P(X)$. When all recipients have acknowledged the receipt of T_i, P_i waits for some time δ and then checks if it received a copy of T_i tagged with X from each process in $P(X)$. If this is the case, then P_i changes the tags of the tokens to "success" since all processes in $P(X)$ have agreed to execute X. If P_i does not receive tokens tagged with X from all processes in $P(X)$ before δ expires then P_i sends a message "request" to each process in $P(X)$ in order to retrieve its tokens. If any returned token is tagged with "success" then it means that another process agreed to execute X and P_i starts executing X. If none of the returned tokens is tagged with "success" then P_i must give up on X and attempt another interaction.

The algorithm TB of [18] is symmetric and distributed. Unlike the algorithm in [19], the bound for δ is not predefined anymore and this parameter is dynamically adjusted.

Also the solution proposed in [31] can be considered as belonging to the randomized algorithms category, although the randomization is not used explicitly by the process, but assumed implicitly in the scheduler. The idea used by Nestmann in [31] consists in associating a lock l, initially set to *true*, to each choice, and then launch a parallel process for each branch. A process P corresponding to an input branch will try to get both its lock (local lock, l) and the partner's lock (remote lock, r). When P succeeds, it tests the locks: if they are both *true* (meaning that P has won the competition) then P sets the locks to *false* so that all the other processes can *abort*, sends a positive acknowledgment (*true*) to the partner, and proceeds with its continuation. The partner also proceeds when it receives the positive acknowledgment. If the local lock is *false* then P aborts. If the remote lock is *false* then P tells the partner to abort by sending it a negative acknowledgment (*false*).

The problem with the algorithm in [31] is that processes might loop forever in the attempt to get both locks. If the initial situation is symmetric, then it is possible to define a scheduler (even a fair one) which always selects the processes in the same order, and never breaks the symmetry. In [31] it is assumed that the scheduler itself has a random behavior, i.e. it selects at random which process to execute next (and in a way totally independent from the history of the system). As argued in Chapter 1, we believe that it is important to consider stronger schedulers.

6.2 Encoding π into π_{pa}

The main difficulty in encoding the π-calculus into the probabilistic asynchronous π-calculus consists in encoding the choice operator. The other π-calculus operators can be translated homomorphically. The rest of this section will address the encoding of the mixed choice construct.

In order to make the algorithm robust with respect to every scheduler (under an assumption of "proper" behavior), we enhance it with a randomized choice made internally by the processes involved in the synchronization. The idea is similar to the one used by Lehmann and Rabin for solving the dining philosophers problem ([42]). The forks, in this case, are the locks. The idea is to let the process choose randomly the first lock, and wait on it until it becomes available. When the connection graph (where the

forks are the nodes and the philosophers are the arcs) is a simple ring, like in the classic dining philosophers case, this algorithm is deadlock and livelock free with probability 1 under any fair adversary ([42]).

As shown in [37], the problem of the mixed choice however still presents a complication: the connection graph may be rather complex, and in Chapter 5 we showed that the classic algorithm of Lehmann and Rabin does not work for general graphs. For instance, it does not work when a node is part of two or more cycles. In order to cope with this problem, we associate to each choice containing output guards an additional lock h. The processes corresponding to the input branches will first have to compete for the lock h of the partner. Thus, at most one output branch for each choice will be involved at a time in an interaction attempt. This ensures that there will be no connected cycles in the graph representing the interaction attempts, and we will see that this condition is sufficient for the correctness of the algorithm.

For the sake of simplicity we assume that the same channel cannot be used as both input and output guard in the same choice construct.

In the encoding we make use of some syntactic sugar: we assume polyadic communication (i.e. more than one parameter in the communication actions), boolean values $\mathbf{t}$ and $\mathbf{f}$ and an if-then-else construct, which is defined by the structural rules

$$\mathit{if}\ \mathbf{t}\ \mathit{then}\ P\ \mathit{else}\ Q \equiv P \qquad \mathit{if}\ \mathbf{f}\ \mathit{then}\ P\ \mathit{else}\ Q \equiv Q$$

As discussed in [33], these features can be translated into π_a. The encoding of π into π_{pa} is defined in Table 6.2. Note that all the operators are translated homomorphically except for the choice. In the encoding of the choice, l represents the principal lock (corresponding to a fork in the algorithm of Lehmann and Rabin), h represents the auxiliary lock (for ensuring that no more than one output branch for each choice will be involved simultaneously in an interaction attempt). In the encoding of the input prefix, l represents the local principal lock, and r represents the remote principal lock. The name a is used to send an acknowledgment to the partner.

Note that in the encoding of the input-prefix the top-level choice, which represents the arbitrary choice of the first principal lock, is a blind choice ($1/2\ \tau \ldots + 1/2\ \tau \ldots$). This means that the process commits to a lock *before* knowing whether such lock is available. It can be proved that this commitment is essential for the termination of the algorithm. The distribution of the probabilities, on the contrary, is not essential

$$
\begin{array}{rcl}
[\![\nu x P]\!] & = & \nu x [\![P]\!] \\
[\![P_1 \mid P_2]\!] & = & [\![P_1]\!] \mid [\![P_2]\!] \\
[\![X]\!] & = & X \\
[\![rec_X P]\!] & = & rec_X [\![P]\!] \\
\left[\!\!\left[\begin{array}{l} \sum_i \alpha_i.P_i \\ + \\ \sum_j \tau.Q_j \\ + \\ \sum_k \beta_k.R_k \end{array} \right]\!\!\right] & = & \nu l\ (\bar{l}\mathbf{t} \mid \nu h\ (\bar{h} \mid \prod_i [\![\alpha_i.P_i]\!]_{lh})\ \mid\ \prod_j [\![\tau.Q_j]\!]_l\ \mid\ \prod_k [\![\beta_k.R_k]\!]_l) \\
[\![\bar{x}y.P]\!]_{rh} & = & \nu a\ (\bar{x}\langle r, a, h, y\rangle \mid a(b).\ \mathit{if}\ b\ \mathit{then}\ [\![P]\!]\ \mathit{else}\ \mathbf{0}) \\
[\![\tau.Q]\!]_l & = & l(b).(\bar{l}\mathbf{f} \mid\ \mathit{if}\ b\ \mathit{then}\ [\![Q]\!]\ \mathit{else}\ \mathbf{0}) \\
[\![x(y).R]\!]_l & = & rec_X(x(r,a,h,y).h.rec_Y(\ 1/2\ \tau.l(b_L).((1-\varepsilon)\ r(b_R).B + \varepsilon\ \tau.(\bar{l}b_L \mid Y)) \\
 & & \qquad\qquad + \\
 & & \qquad\qquad 1/2\ \tau.r(b_R).((1-\varepsilon)\ l(b_L).B + \varepsilon\ \tau.(\bar{r}b_R \mid Y))\ )) \\
\text{where} & & \\
B & = & \begin{array}{llll} & \mathit{if}\ b_L \wedge b_R & \mathit{then} & \bar{h} \mid \bar{l}\mathbf{f} \mid \bar{r}\mathbf{f} \mid \bar{a}\mathbf{t} \mid [\![R]\!] \\ \mathit{else} & \mathit{if}\ b_L & \mathit{then} & \bar{h} \mid \bar{l}\mathbf{t} \mid \bar{r}\mathbf{f} \mid \bar{a}\mathbf{f} \mid X \\ \mathit{else} & \mathit{if}\ b_R & \mathit{then} & \bar{h} \mid \bar{l}\mathbf{f} \mid \bar{r}\mathbf{t} \mid \bar{x}\langle r, a, h, y\rangle \\ & & \mathit{else} & \bar{h} \mid \bar{l}\mathbf{f} \mid \bar{r}\mathbf{f} \mid \bar{a}\mathbf{f} \end{array}
\end{array}
$$

Table 6.2. The encoding of π into π_{pa}. In the translation of the mixed choice, the α_i's represent output actions, and the β_k's represent input actions. ε stands for a real number in $[0, 1)$.

for termination. However, this distribution affects the efficiency, i.e. how soon the synchronization protocol will converge. It can be proved that in this choice it is better to split the probability as evenly as possible, hence 1/2 and 1/2.

Once the process has obtained the first principal lock, the idea is that it should try to get the second one. If it succeeds, then it should test the locks and proceeds accordingly to the results of the tests as explained at the beginning of this section. Otherwise, it should release both locks and go back to the beginning of the inner loop, where it will make another random draw for selecting the first lock. This conditional behavior would need a priority choice to be expressed, namely a choice in which the first branch would always be selected whenever the corresponding guard is enabled. Such construct does not exist in the (asynchronous) π-calculus, and its introduction would make the semantics rather complicated (although it would be easy to implement it in a language like Java). To overcome the problem, we use a probabilistic choice $((1-\epsilon)\ldots+\epsilon\ldots)$ to approximate a priority choice. Of course, the smaller ϵ is, the tighter the approximation is.

6.3 Correctness of the encoding

In order to assess the correctness of the translation of π into π_{pa}, we consider a probabilistic extension of the notion of testing semantics proposed in [34, 3]. This extension has the advantage of being probabilistically "reasonable", i.e. sensitive to deadlocks and livelocks with non-null probability. Furthermore, in testing semantics all communications are internalized (except the one used by the observer to declare *success*), and this spares us from the problem, discussed in [31], which arises with semantics like bisimulation, barbed bisimulation, and coupled simulation, even in their weak and asynchronous versions. The kind of encoding that we use for choice cannot be correct with respect to these semantics, due to their sensitiveness to the output capabilities. In fact, in the original process the output guards which are not chosen disappear after the choice is made. In the translation, however, a choice is mapped into the parallel composition of the branches, hence an output guard which is not able to interact with a partner will remain present even after some other branch wins the competition, thus causing the presence of a residual output barb. However these barbs are "garbage" by definition, not able to synchronize with any other process at this point (at least, not according to the synchronization protocol of the translated process), so they should not be counted. This sensitivity to the synchronization capabilities is exactly what testing semantics features, differently from bisimulation semantics.

6.3.1 Testing semantics

Let us recall the key concepts of the testing semantics for the π-calculus. An *observer* O is a π-calculus process able to perform a special action ω, denoting success. Usually ω is seen as an output action, but it does not really matter. We assume this action to be different from all those performed by tested processes. Given a π-calculus process P and an observer O, an *interaction* between P and O is a maximal (finite or infinite) sequence of τ transitions starting from $P \mid O$:

$$P \mid O = Q_0 \xrightarrow{\tau} Q_1 \xrightarrow{\tau} Q_2 \xrightarrow{\tau} \dots$$

Maximal means that the sequence is either infinite, or the last state is not able to make any further τ transition.

We say that P **may** O iff there exists an interaction such that $Q_i \xrightarrow{\omega}$ for some i. We say that P **must** O iff for every interaction there exists i such that $Q_i \xrightarrow{\omega}$. Finally, P is *testing equivalent* to Q, notation $P \simeq Q$, if for every observer O, P **may** O iff Q **may** O, and P **must** O iff Q **must** O.

In order to state the correctness of the embedding, we need to extend the notion of testing to the π_{pa}-calculus. We propose the following extension, which, we believe, captures the spirit of testing semantics.

6.3.2 Testing semantics for the π_{pa}-calculus

The natural extension to π_{pa} of the concept of interaction between a process P and an observer O is an execution starting from $P \mid O$, under some adversary ζ, and consisting only of arcs labeled by τ. An interaction is *successful* if it passes trough a state in which an ω step can be performed.

Our intended notion of must testing is that the probability that an interaction is successful is 1. To this end, we need to define $pb(\xi \textit{ is successful} \mid \xi \textit{ is an interaction})$, the probability of successful executions relatively to those executions which are interactions. This notion can be formalized in two different, but equivalent ways:

- Define an interaction as a branch of the execution tree of $P \mid O$ under some ζ, with the property that all arcs of the branch are labeled by τ. Then define $pb(\xi \textit{ is successful} \mid \xi \textit{ is an interaction})$ as the *relative probability* that a branch be successful, given that it is an interaction.

- Restrict the execution tree to contain only arcs labeled by τ and by ω. This can be done by closing the initial process $P \mid O$ with respect to all the free names except ω. Then define $pb(\xi \textit{ is successful} \mid \xi \textit{ is an interaction})$ as the probability of the successful branches of this tree.

The first solution is more elegant, but it is formally more complicated since it involves defining the concept of relative probability in an execution tree. Hence we follow the second approach.

In the sequel we denote by νP the process $\nu x_1 \nu x_2 \ldots \nu x_n P$, where $\nu x_1, \nu x_2, \ldots, \nu x_n$ are all the free names occurring in P. With a slight abuse of notation, we denote the execution tree of the automaton generated by P under the adversary ζ as $etree(P, \zeta)$, and the set of its branches (executions) as $exec(P, \zeta)$.

Let P be a π_{pa} process and let O be a π_{pa} observer. An interaction ξ between P and O is an element of $exec(\nu(P|O), \zeta)$. Given an interaction ξ of the form:

$$\nu(P \mid O) = Q_0 \xrightarrow[p_0]{\tau} Q_1 \xrightarrow[p_1]{\tau} Q_2 \xrightarrow[p_2]{\tau} \ldots,$$

we say that ξ is *successful* if there exist i and p such that $Q_i \xrightarrow[p]{\omega}$. We denote by $sexec(\nu(P|O), \zeta)$ the set $\{\xi \in exec(\nu(P|O), \zeta) \mid \xi \textit{ is successful}\}$. For $i < j$, we say that the node (labeled by) Q_j is a *descendant* of Q_i, and the step $Q_j \xrightarrow[p_j]{\tau} Q_j$ is *subsequent* to Q_i.

The following property is fundamental for defining our notion of testing for π_{pa}:

Proposition 6.1. *Given an adversary ζ, the set $sexec(\nu(P|O), \zeta)$ can be obtained as a countable union of disjoint cones.*

Proof For every $\xi \in sexec(\nu(P|O), \zeta)$, let $\xi' \leq \xi$ be the prefix which ends at the first i such that $Q_i \xrightarrow[p]{\omega}$. We have that $\mathcal{C} = \{C_{\xi'} \mid \xi \in sexec(\nu(P|O), \zeta)\}$ is a set of disjoint cones (see Section 3 for the definition of cone) and $\cup_{C \in \mathcal{C}} C = sexec(\nu(P|O), \zeta)$. Countability follows from the fact that $etree(\nu(P|O), \zeta)$ is finitely branching. □

As a consequence of this proposition, the probability of $sexec(\nu(P|O), \zeta)$ is well defined (cfr. Chapter 3).

Definition 6.1. *Let $\mathcal{A}$ be a class of adversaries. Let P, Q be π_{pa} processes and O be a π_{pa} observer.*

(i) P **may**$_{\mathcal{A}}$ O *iff there exists an adversary* $\zeta \in \mathcal{A}$ *s.t.* $pb(sexec(\nu(P|O), \zeta)) > 0$.

(ii) P **must**$_{\mathcal{A}}$ O *iff for all adversaries* $\zeta \in \mathcal{A}$, $pb(sexec(\nu(P|O), \zeta)) = 1$.

(iii) $P \simeq_{\mathcal{A}} Q$ *iff for every* O, P **may**$_{\mathcal{A}}$ O *iff* Q **may**$_{\mathcal{A}}$ O, *and* P **must**$_{\mathcal{A}}$ O *iff* Q **must**$_{\mathcal{A}}$ O.

Note that, although P **must**$_{\mathcal{A}}$ O implies P **may**$_{\mathcal{A}}$ O (for $\mathcal{A} \neq \emptyset$), must-equivalence does not imply may-equivalence. Hence it makes sense to require both in the definition of $\simeq_{\mathcal{A}}$.

6.3.3 Dining philosophers without the fairness assumption

First of all, we need to make precise what class of adversaries our algorithm can cope with. Clearly, we wish this class to be as large as possible. Yet, we cannot allow just *any* adversary. The problem is related to the output actions: a malicious adversary that never schedules $\bar{l}b_L$ or $\bar{r}b_R$ in the definition of $[\![x(y).P]\!]_l$ will make it impossible for the process to get the lock and therefore will force it to loop forever. Therefore we consider the class of proper adversaries for the encoding of π in π_{pa}.

Note that the definition of proper scheduler in Section 4.3.1 is weaker than the notion of fair scheduler, which requires that *any* process which is ready infinitely often will eventually be scheduled for execution. Clearly, the fairness assumption would be sufficient for our encoding, however it is not necessary. This may seem surprising, since the solution to the dining philosophers proposed in [42] *requires* fairness. However, a careful analysis of the algorithm in [42] reveals that the fairness assumption is used *only* because a philosopher who has committed to a fork enters a *busy waiting* loop, and it remains in the loop until the fork becomes available. An unfair scheduler, hence, could keep scheduling always the same philosopher in a busy waiting loop, thus generating a livelock. If the busy wait is replaced by a suspension command (obliging the scheduler to select another process) then the fairness assumption is not necessary. We prove this result in the following.

Table 6.3 shows a π_{pa}-calculus implementation of the first algorithm in [42] where we replace the busy wait loop on the first fork with a suspend command (i.e. suspend until the fork becomes available) and remove the state in which a philosopher thinks. The reason for dropping the thinking state is that it can be replaced by the state in which a philosopher is hungry. Note that in the original dining philosophers algorithm a philosopher who is thinking can only advance to a state in which he is hungry. Therefore we can use the state in which a philosopher is hungry also to represent the thinking state.

Philosophers are represented by the processes P_i, where $i \in \{0, 1, \ldots, n\}$ and n is the number of philosophers. We use the channels R_i ($i \in \{0, 1, \ldots, n\}$) to denote the resources (forks) and we assume that process $P_{i\oplus 1}$ is on the right of process P_i and that resource $R_{i\oplus 1}$ is between processes P_i and $P_{i\oplus 1}$. We use the symbol $\oplus$ to denote the sum modulo n.

First, let us introduce some notations that will be used to prove that the algorithm is Table 6.3 is deadlock and livelock free with respect to any scheduler. We use W, S, E_F and E_S to denote process states. A process is in state W when it is waiting for its first fork. A process is in state S if it is holding only the first fork. State E_F represents a process that finished eating and it is still holding the two forks, while state E_S indicates that a process that ate but is still holding one fork.

We sometime say that a process P is in state W (resp. S) on fork f to denote the situation in which P chose f as its first fork and is waiting for f (resp. holding f).

We use the definition given in [47] for the set G of good states. Namely, a process is in a good state if it is waiting on or is holding the first fork f (i.e. it is in W or S on f), and its second fork f' is not controlled by the neighbor (i.e. the neighbor is not in state W or S on f'). Figure 6.1 illustrates a process in a good state. The states of G play an important role in the proof since this is a place where the symmetry is broken.

Given the above definition of G, we introduce the notation G_n to denote that the set of states in which the number of processes in a good state is at least n.

Two processes P and Q form a pair of adjacent processes if the following conditions are satisfied:

- P and Q are neighbors
- P is situated on the left side of Q at the round table
- P is committed to or holding its left fork f_P as first fork, i.e. P is in state W or S on f_P

$$
\begin{array}{rcl}
P_i & = & \bar{R}_i\langle b_i\rangle \\
 & & \mid rec_X(\ 1/2\tau.R_i(b_i).(\ (1-\epsilon)R_{i\oplus 1}(b_{i\oplus 1}).S \\
 & & \qquad\qquad + \\
 & & \qquad\qquad \epsilon\tau.(\bar{R}_i\langle b_i\rangle \mid X)) \\
 & & \qquad + \\
 & & \qquad 1/2\tau.R_{i\oplus 1}(b_{i\oplus 1}).(\ (1-\epsilon)R_i(b_i).S \\
 & & \qquad\qquad\qquad + \\
 & & \qquad\qquad\qquad \epsilon\tau.(\bar{R}_{i\oplus 1}\langle b_{i\oplus 1}\rangle \mid X))\ ) \\
\text{where} & & \\
S & = & 1/2\tau.\bar{R}_i\langle b_i\rangle.\bar{R}_{i\oplus 1}\langle b_{i\oplus 1}\rangle \\
 & & + \\
 & & 1/2\tau.\bar{R}_{i\oplus 1}\langle b_{i\oplus 1}\rangle.\bar{R}_i\langle b_i\rangle
\end{array}
$$

Table 6.3. A π_{pa} solution for the dining philosophers problem. Here $i \in \{0, 1, \ldots, n\}$ where n is the number of philosophers and $\oplus$ is the sum modulo n.

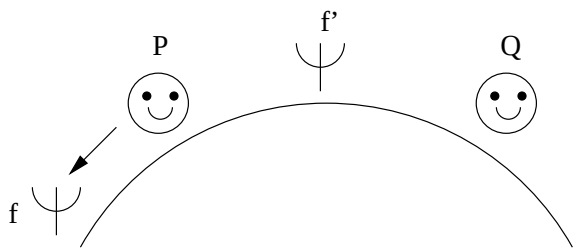

Fig. 6.1. A process P in a good state.

- Q is committed to or is holding its right fork f_Q as first fork, i.e. Q is in state W or S on f_Q

Adjacent processes also represent a significant element in the proof. Note that fork f_{PQ} situated in between the adjacent processes P and Q is not controlled by any process and that the first of two processes who will try to acquire f_{PQ} as second fork will succeed and eat.

Figure 6.2 illustrates a pair of adjacent processes.

We introduce the notation A_n to denote the set of states in which the number of processes which form adjacent pairs is at least n.

Another important observation is that a process can either have both forks controlled or held by the neighbors or at least one of its forks is free. Let BC denote the state in which a process that has both forks controlled or held by the neighbors.

In order to prove the correctness of the π_{pa} solution to the dining philosophers problem we will use the following lemmata. We use the notation Eat to denote the set of states in which at least one process is eating.

Lemma 6.1. *Let n be the number of philosophers. If n is even, then $A_n = G_n \xrightarrow[1]{} Eat$.*

Proof Since we do not have busy waiting, and not all processes can be suspended, some process will eventually try to get the second fork. Since the process will find the second fork free, it will also eat. □

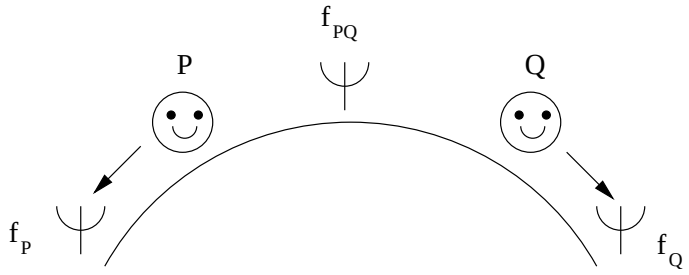

Fig. 6.2. A pair of adjacent processes.

Lemma 6.2. *Let n be the number of philosophers. If n is odd, then* $G_{n-1} \xrightarrow[1/2]{} Eat \cup G_{n-1}$.

Proof If one of the processes in a good state checks for the second fork then it eats. If the process that is not in a good state checks for the second fork, then it either eats, or it finds the fork unavailable. In the second case the process goes back to the state in which randomly chooses the first fork. If the process proceeds, then it will commit to the unavailable fork with probability 1/2, and therefore suspend. At this point, only one of the good processes can move. □

Note that if the number of philosophers is odd then the maximum number of adjacent pairs is $n-1$ and the maximum number of processes in a good state is $n-1$. Furthermore we have that G_{k-1} implies A_{k-1}.

Lemma 6.3. *Each process in state BC, if it is scheduled at least two times and if it does not eat in the interval in between, will reach a state different from BC with probability 1/4.*

Proof Consider the scenario in Figure 6.3, where P is a process in state BC, f_Q is the fork shared with the left neighbor process Q and f_R is the fork shared with the right neighbor process R. If P flips when it has both forks controlled or held by the neighbors, and if P does not eat before flipping again, then the second fork must have been held by the neighbor. Assume without loss of generality that the second fork f_Q is

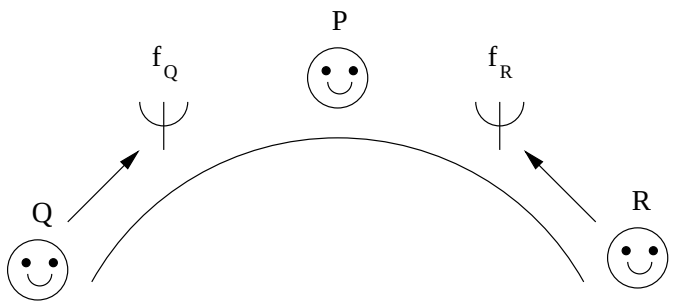

Fig. 6.3. A process P in state BC.

held by Q. When P flips again (for the second time), f_Q can be in one of the following situations:

If f_Q is not held anymore by Q then, with probability at least $1/2$, P is not in BC anymore. This is because if Q has made a random draw before P was scheduled for its second random draw, then Q has committed to the other fork with probability $1/2$. Therefore f_Q is not controlled by Q and P is in BC with probability $1/2$. If P makes the second random draw before Q, then P is already in BC since f_Q is free.

If f_Q is still held when P makes the second random draw, then with probability $1/2$ P will commit to f_Q. Since f_Q is not available P will suspend. If P makes a third random draw, then P must have taken f_Q and then released it because it has not found the second fork f_R available. Therefore f_Q must be released before P's third random draw. If Q makes a random draw before P, then Q will commit to the other fork with probability $1/2$. Hence P moves to a state different from BC with probability $1/2$, which combined with the previous case gives the probability $1/4$. If P makes the third random draw before Q, then P is already in BC since f_Q must be free. □

We are now ready to show the main result of this section.

Theorem 6.2. *Consider the algorithm in Table 6.3 and assume that the scheduler cannot select a suspended process. Then the algorithm is deadlock-free and livelock-free with respect to any scheduler.*

Proof We will show that, from any state, a state in which at least one process is eating is reached with probability 1 under every scheduler. Note that we reason from a global point of view rather than individual, i.e. we consider the state of all processes at the same time.

We first show the following progress statement, which states that from a generic state, with probability $1/2 * 1/4^k$, the system reaches a states in which a process eats or either the number of processes or the number of adjacent pairs increases by 1:

$$G_n \cap A_m \underset{1/2*1/4^k}{\longrightarrow} Eat \cup (G_{n+1} \cap A_m) \cup (G_n \cap A_{m+1}) \tag{6.1}$$

Consider a generic state in $G_n \cap A_m$. First, observe that in every computation the flipping action is repeated infinitely often since there is no busy waiting. Second, if k is the number of philosophers, then between two consecutive random draws performed by processes that are not in BC at the moment of the random draw, if no process has been eating, there can be at most $2 * k$ random draws performed by processes in BC. This is because the calculation in Lemma 6.3 can be repeated up to k times, and according to Lemma 6.3 each process in BC that has not eaten, reaches a state different from BC after two random draws with probability 1/4. This gives the $1/4^k$ component of the probability in the above progress statement.

Next, consider a process P that is not in BC. Let f be the fork shared with the left neighbor Q and f' be the fork shared with the right neighbor R. Since P is not in BC, then at least one of the forks f, f' is free. Assume, without loss of generality, that f is the free fork (note that f' could be free or controlled by R).

Q and R are in one of the following cases when P makes a random draw: both Q and R are not in a good state, one of Q and R is in a good state and the other one is not in a good state or both Q and R are in a good state.

The case in which both Q and R are not in a good state can be further divided in two subcases:

- if R controls f', then, when P makes a random draw, it will commit to f' with probability 1/2. Since P is now in a good state, the total number of processes in a good state increased by 1 (note that the state of Q and R has not changed).

- if R does not control f' then both f and f' are available when P makes the random draw. Hence the size of the set of processes in a good state increases by 1, because P is now in a good state, and the other processes did not change state.

If Q is in a good state but R is not in a good state, then with probability $1/2$ P commits to f' and becomes a process in a good state. Since Q and R maintain their state, the number of good processes increases by 1. Also note that P and Q now form an adjacent pair and the number of adjacent pairs increases by 1.

If Q is not in a good state but R is in a good state, then we have the following subcases:

- if R controls f' then with probability $1/2$ P commits to f' and becomes a process in a good state. Both Q and R maintain their state and the number of good processes increases by 1.

- if R does not control f' then with probability $1/2$ P commits to f and becomes a process in a good state. R is still in a good state because f' is not controlled by P and Q's state remains the same. Hence the number of good processes increases by 1.

The last case that we need to consider is when both Q and R are in a good state. Again we will divide this case in two subcases:

- if R controls f' then with probability $1/2$ P commits to f' and becomes a process in a good state. Both Q and R maintain their state and the number of good processes increases by 1.

- if R does not control f' and R is in a good state, then f' must be R's second fork. When P makes a random draw it will commit to f' with probability $1/2$ and becomes a process in a good state. However, R is not in a good state anymore because its second fork f' is now controlled by P. The number of processes in a good state remains constant in this case. On the other hand, Q is still in a good state, P is now in a good state and the fork f shared with P is free. Therefore P and Q form an pair of adjacent processes and the number of adjacent pairs increases by 1.

We have showed that whenever a process that is not in BC makes a random draw, with probability $1/2$, either the set of processes in a good state or the number of adjacent pairs increases by 1. We also know that each process in BC that is scheduled at least two times and does not eat in the meanwhile, will reach a state different from BC with probability $1/4$. Since this process can be repeated up to k times we get the probability $1/2 * 1/4^k$ for the progress statement (6.1).

Since there are k processes, in the worst case scenario the number of good processes and the number of adjacent pairs need to increase by 1 for k times. Therefore when k is even the system reaches a states in which all processes are in a good state (or all pairs are adjacent) with probability $(1/2 * 1/4^k)^k$. If k is odd a state in which $k-1$ processes are in a good state is reached with the same probability.

By composing the progress statement (6.1) with Lemma 6.2 or Lemma 6.1 we have that from any generic state we can reach *Eat* with positive probability $((1/2 * 1/4^k)^k$ and $1/2 * (1/2 * 1/4^k)^k$ respectively. Furthermore, if *Eat* is not reached then we remain, of course, in a generic state. Hence we can apply Lemma 3.3 (Progress with probability 1) to prove that *Eat* is reached with probability 1. □

Note that this result was also proved independently in [8]. The correctness of the algorithm in [8] is proved by using a method that has been introduced by the same authors in [7] to show the convergence of randomized distributed algorithms with respect to deterministic and memoryless schedulers and arbitrary schedulers.

6.3.4 Correctness of the encoding with respect to testing semantics

It is important to note that π_{pa} (like most process algebra) has a suspension mechanism associated with the communication actions: if a process can proceed only by performing a handshaking, then the process will suspend until the partner is ready. Furthermore the semantics of π_{pa} ensures that a scheduler is obliged to select processes which are not suspended. Note that in $[\![x(y).P]\!]_l$ (Table 6.2) the acquisition of h (auxiliary lock) and of the first lock are done by input prefixes (with no alternatives) and therefore they will suspend if the locks are unavailable. It is easy to implement such suspension mechanism in a language like Java by using the `wait()` and `notify()` primitives.

Another important ingredient of the correctness proof is that at any point of the execution of $[\![P]\!]$ in the graph representing the interaction attempts all cycles are disconnected (i.e. they are not connected to each other by any path). This is fundamental because we showed in Chapter 5 that the algorithm of [42] is not livelock-free if the connection graph (where the forks are the nodes and the philosophers are the arcs) contains two different non-disjoint cycles, not even under the fairness hypothesis.

Lemma 6.4. *Let P be a π-calculus process. Let ζ be any adversary, and let ξ be an execution of $[\![P]\!]$ with respect to ζ. For any point of ξ, consider the graph which has as nodes the principal locks l, and such that there is an arc between l and l' iff there is a process (corresponding to an input branch) for which both input actions on l and l' are*

enabled at some point of the computation. Then all cycles in the graph are disconnected (i.e. for any pair of different cycles there are no paths connecting one node of the first to one node of the second).

Proof We show that the scheduler cannot create a situation where two cycles are connected by a path. We consider all possible configurations in which a node belongs to more than one cycle, and we show that none of these configurations can occur. Note that the graph described in this lemma evolves dynamically because a process may die (thus eliminating the corresponding arc) and because a lock may get communicated to a new process (thus creating a new arc).

We use the following notation: a directed edge from l to l' denotes that l is the local principal lock, l' is the remote principal lock and the process corresponding to the arc between l and l' was able to acquire the auxiliary lock h of the partner. Note that we have a directed graph in which the indegree of every node is at most 1, i.e. the number of arcs coming in to the node from other nodes is at most 1.

Figure 6.4 illustrates the three generic configurations in which a node is part of at least two cycles (more complex configurations can be reduced to these three cases).

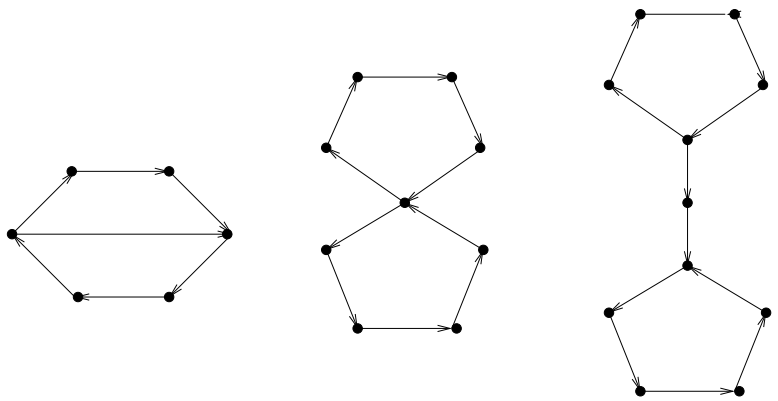

Fig. 6.4. Multiple cycles configurations

Each of the configurations in Figure 6.4 contains at least two different non-disjoint cycles. However, none of these configurations is valid. This is because they all have at least one node with the indegree greater than 1, which means that the auxiliary lock h corresponding to a choice containing output guards was acquired by more than one process. According to the encoding in Table 6.2 this is not possible.

Note that it is possible to create non-disjoint cycles different from the ones in Figure 6.4 if the direction of the arcs is reversed, but the resulting configurations are also not valid. □

We are now ready to state our main result. We begin by showing that, under proper schedulers, the translated processes reflect the may behavior of the original processes, provided that the observer are also translated (Theorem 6.3). To this end, we use the following two lemmata. In the sequel, given a π_{pa} process P, we denote by $\tilde{P}$ the π_a process obtained from P by removing all the probabilities from the choice constructs.

Lemma 6.5. *For every π_{pa} process P and observer O*

$$P \textbf{ may}_{\mathcal{P}} O \textit{ iff } \tilde{P} \textbf{ may } \tilde{O}$$

Proof

only if) Assume $pb(sexec(\nu(P|O), \zeta)) > 0$ for some proper scheduler ζ. Then there exists an execution ξ of $\nu(P \mid O)$ under ζ such that ξ is successful, i.e. ξ is of the form

$$\nu(P \mid O) = Q_0 \xrightarrow[p_0]{\tau} Q_1 \xrightarrow[p_1]{\tau} Q_2 \xrightarrow[p_2]{\tau} \dots$$

and $Q_i \xrightarrow[p]{\omega}$ for some i and p. By eliminating the probabilities from ξ, we obtain a successful interaction of $\tilde{P} \mid \tilde{O}$, namely:

$$\tilde{P} \mid \tilde{O} = \tilde{Q}_0 \xrightarrow{\tau} \tilde{Q}_1 \xrightarrow{\tau} \tilde{Q}_2 \xrightarrow{\tau} \dots$$

and $\tilde{Q}_i \xrightarrow{\omega}$.

if) Assume $\tilde{P}$ **may** $\tilde{O}$. Then there exists an interaction

$$\tilde{P} \mid \tilde{O} = \tilde{Q}_0 \xrightarrow{\tau} \tilde{Q}_1 \xrightarrow{\tau} \tilde{Q}_2 \xrightarrow{\tau} \dots$$

such that, for some i, $\tilde{Q}_i \xrightarrow{\omega}$. Therefore, for suitable probabilities $p_0, p_1, p_2, \dots$ we have an execution fragment ξ' of the form

$$\nu(P \mid O) = Q_0 \xrightarrow[p_0]{\tau} Q_1 \xrightarrow[p_1]{\tau} Q_2 \xrightarrow[p_2]{\tau} \dots Q_i$$

such that, for some p, $Q_i \xrightarrow[p]{\omega}$. Furthermore, since ξ' is finite, we can define a proper scheduler ζ such that ξ' is an execution fragment of $\nu(P \mid O)$ under ζ, from which we derive that $C_{\xi'} \subseteq \mathit{sexec}(\nu(P|O), \zeta)$. Hence we have $pb(\mathit{sexec}(\nu(P|O), \zeta)) \geq pb(\xi') = p_0 p_1 p_2 \dots p_i > 0$. □

Next lemma proves that the may testing is preserved by the translation. From now on, we will assume that ω is the name of the channel on which the action denoting success is performed, i.e. we ignore the number of parameters. In other words, ω is not affected by the translation. This assumption allows us to use the same notion of success for the original and the translated process, thus simplifying the formulation of the correspondence. In the sequel, we use the symbol $\Longrightarrow$ to represent the reflexive and transitive closure of $\xrightarrow{\tau}$.

Lemma 6.6. *For every π process P and observer O*

$$P \textbf{ may } O \textit{ iff } \widetilde{[\![P]\!]} \textbf{ may } \widetilde{[\![O]\!]}$$

Proof

only if) This part trivially follows from the observation that for every π processes Q and Q', if $Q \xrightarrow{\tau} Q'$, then $\widetilde{[\![Q]\!]} \Longrightarrow \widetilde{[\![Q']\!]}$, i.e. there are π_a processes $R_0, R_1, R_2, \dots, R_n$ such that

$$\widetilde{[\![Q]\!]} = R_0 \xrightarrow{\tau} R_1 \xrightarrow{\tau} R_2 \xrightarrow{\tau} \dots R_n = \widetilde{[\![Q']\!]}$$

The additional steps are necessary for performing the synchronization protocol. The processes $R_1, R_2, \ldots$ represent the intermediate states during the execution of the protocol.

if) This part follows from the observation that for every π processes Q, if $\widetilde{[\![Q]\!]} \Longrightarrow R$, then there exists a π process Q' such that $R \Longrightarrow \widetilde{[\![Q']\!]}$. Furthermore, if $R \xrightarrow{\omega}$, then $[\![Q']\!] \xrightarrow{\omega}$. Note that R may not correspond to the translation of any π process because there may be synchronization protocols which have been started but not yet completed in R. By completing them we obtain a process which is a translation of a π process, namely $[\![Q']\!]$. The capability of R of performing an ω step is preserved in $[\![Q']\!]$ because by definition ω is not an internal action of the translation. □

Theorem 6.3 (Correctness of the encoding with respect to may testing). *For every π process P and observer O*

$$P \textbf{ may } O \textit{ iff } [\![P]\!] \textbf{ may}_{\mathcal{P}} [\![O]\!]$$

Proof From Lemma 6.6 we have that P **may** O iff $\widetilde{[\![P]\!]}$ **may** $\widetilde{[\![O]\!]}$. From Lemma 6.5 we have that $\widetilde{[\![P]\!]}$ **may** $\widetilde{[\![O]\!]}$ iff $[\![P]\!]$ **may**$_{\mathcal{P}}$ $[\![O]\!]$. □

We now prove the correctness of the embedding also with respect to must testing (Theorem 6.4). This part more difficult, because the must version of Lemma 6.6 does not hold, due to the possibility of infinite loops generated by the synchronization protocol. We need to show that such loops have probability 0.

Theorem 6.4 (Correctness of the encoding with respect to must testing). *For every π process P, and every observer O*

$$P \textbf{ must } O \textit{ iff } [\![P]\!] \textbf{ must}_{\mathcal{P}} [\![O]\!]$$

Proof

only if) Assume P **must** O. We have to show that $[\![P]\!]$ **must**$_{\mathcal{P}}$ $[\![O]\!]$. Since $[\![P]\!] \mid [\![O]\!] = [\![P \mid O]\!]$, we need to show that for all adversaries $\zeta \in \mathcal{P}$,

$pb(\{\xi \in exec(M_{[\![P|O]\!]}, \zeta) \mid succ(\xi)\}) = 1$. Given an interaction of $P \mid O$, we can mimic the same steps up to the point in which a synchronization involving some choice processes occurs. Suppose that $P_1, \ldots, P_n$ are the processes involved in the synchronization. For each pair P_i, P_j which can synchronize, we know that the interaction (in the original π process) will be successful. The risk is that $[\![P_1]\!], \ldots, [\![P_n]\!]$ will loop forever in the synchronization protocol. This will happen only if none of the processes $[\![P_1]\!], \ldots, [\![P_n]\!]$ will ever be able to acquire both the local and the remote lock. However, we can show that this situation has only probability 0. In fact, after the actions of the form $x(r, a, y)$ (see Table 6.2) have been executed (synchronized with their corresponding output actions), we are in the situation in which several parallel processes compete, pairwise, on the same locks. Consider the graph described in Lemma 6.4. By previous proposition, we know that each connected component contains at most one cycle. The proof then proceeds, for each connected component, like in Theorem 6.2.

if) Assume by contradiction that there exists an interaction between P and O of the form

$$P \mid O = Q_0 \xrightarrow{\tau} Q_1 \xrightarrow{\tau} Q_2 \xrightarrow{\tau} \ldots$$

such that, for all i, $Q_i \not\xrightarrow{\omega}$. We can then construct an interaction between $[\![P]\!]$ and $[\![O]\!]$ of the form

$$[\![P \mid O]\!] = [\![Q_0]\!] \Longrightarrow [\![Q_1]\!] \Longrightarrow [\![Q_2]\!] \Longrightarrow \ldots$$

and for all i, $[\![Q_i]\!] \not\xrightarrow{\omega}$. This contradicts the hypothesis that $[\![P]\!]$ **must**$_{\mathcal{P}}$ $[\![O]\!]$.

□

The above results refer to a notion of correctness which is specifically formulated for testing semantics, so we should justify their generality and strength. A more standard notion of correctness, used in several works about translations (like for instance [32]) is the following: two translated processes are required to be semantically distinguished whenever the original processes are. The following corollary, which is an immediate

consequence of the above theorem, states correctness in the standard process algebra sense.

Corollary 6.1. *For every π-calculus processes P and Q, if $[\![P]\!] \simeq_{\mathcal{P}} [\![Q]\!]$ then $P \simeq Q$.*

Proof Assume $[\![P]\!] \simeq_{\mathcal{P}} [\![Q]\!]$. Then, for every π_{pa} observer O, $[\![P]\!]$ **may** O iff $[\![Q]\!]$ **may** O, and $[\![P]\!]$ **must** O iff $[\![Q]\!]$ **must** O. In particular, this holds for $O = [\![O']\!]$, for any π process O'.

From Theorem 6.3 and Theorem 6.4 we deduce that, for every O', P **may** O' iff Q **may** O', and P **must** O' iff Q **must** O'. □

Note that the viceversa (full abstraction) does not hold: This is due to the fact that, if we allow arbitrary observers in π_{pa}, then we can distinguish $[\![P]\!]$ and $[\![Q]\!]$ by using observers which interact directly with their actions, i.e. without following the synchronization protocol enforced by the algorithm.

Chapter 7

A distributed implementation of π_{pa} in Java

In this chapter, we present an experimental implementation of the *synchronization-closed* π_{pa}-calculus, namely the subset of π_{pa} consisting of processes in which all occurrences of communication actions $x(y)$ and $\bar{x}y$ are under the scope of a restriction operator νx. In other words this means that all communication actions are forced to synchronize. The implementation is compositional with respect to all the operators, and distributed, i.e. homomorphic with respect to the parallel operator. The implementation uses Java as the target language.

We restrict the implementation to the synchronization-closed π_{pa}-calculus because the full calculus is much more abstract than Java. For example in the π_{pa}-calculus there are computations that are virtual and cannot be expressed in a compositional way.

The purpose of this implementation is to prove that π_{pa} is a sensible paradigm for the specification of distributed algorithms, since it can be implemented without loss of expressivity. We can then argue that our encoding fills the gap between the expressiveness of the π-calculus and its distributed implementation.

7.1 General architecture

In the following we describe the high level architecture of the implementation. Implementation details can be found in section 7.2 and Appendix B.

The π_{pa} channels are implemented as Java threads. Each π_{pa} channel manages two lists: availMessages and waitingProcesses. The availMessages list is used by π_{pa} processes to send messages, namely a process executing an output action puts the datum that it wants to send in the availMessages list of the corresponding channel. For example, a process $\bar{x}y$ will insert the Java encoding of the π_{pa} process y in the availMessages list of the π_{pa} channel x. The waitingProcesses list contains requests generated by processes executing guarded choices. A probabilistic choice process inserts a request, which consists in a channel c that is private to the probabilistic choice process, in the waitingProcesses list of each input channel in the probabilistic choice and then waits for

a notification to make a choice. The channel c is used as a buffer where all π_{pa} channels corresponding to the branches of the choice are inserted once they become available for synchronization. A π_{pa} channel processes the request only if it can participate in the handshaking, i.e. the availMessages list is not empty, in which case removes the first element in availMessages, places itself in the availChannels list of channel c and notifies the probabilistic choice that issued the request. When notified, the probabilistic choice process selects with normalized probability one of the π_{pa} channels in c and enables its corresponding branch.

The thread corresponding to a π_{pa} channel is suspended if no requests or data are available. Hence the thread is waken up by a probabilistic choice process or by an output process. Note that c's type is different from the π_{pa} channels.

Channel c also contains a time stamp t which is used to disable the requests of branches belonging to probabilistic choice processes that already made a choice, but still have requests in the waitingProcesses list. Whenever a choice is made the time stamp of the channel c corresponding to the probabilistic choice process is increased, such that only requests having the value of the time stamp equal to this value are valid. The invalid requests are removed from channels' lists when detected.

If a probabilistic choice contains both input guarded and τ-guarded processes, then it first tries to take a local decision. If a τ-guarded branch is selected then its continuation is enabled. No other steps are necessary in this case since a τ-guard is in fact a blind choice. Otherwise a request is inserted in the waitingProcesses list of each input channel in the probabilistic choice and the algorithm proceeds as described in the beginning of this section.

7.2 Implementation detail

As mentioned before, π_{pa} channels are implemented as threads, namely as objects of the `PiChannel` class. Note that the methods that implement the output and the input actions of π_{pa} processes are both *synchronized*, because the placement and removal of a datum must be done atomically. The channels that are private to probabilistic choice processes are implemented as objects of the `Channel` class.

In order to generate the Java implementation of the π_{pa} processes, we chose to use an Extensible Markup Language (XML) representation of the π_{pa} processes. XML is an universal format that is widely used to represent structured data and documents. XML was developed by the W3C consortium. The advantage of using XML for representing

π_{pa} processes is that is platform-independent, easy to parse, and comes with a standard set of functions calls for manipulating XML files from a programming language.

All XML files representing π_{pa} processes are validated against a Document Type Definition (DTD) called `ProbPiProcess.dtd`. The purpose of a Document Type Definition is to define the legal building blocks of an XML file and ensure that XML documents are well-formed. A DTD in an XML document provides a list of the elements, attributes, comments, notes, and entities contained in the document. It also indicates their relationship to one another within the document. In other words, a DTD is the grammar of an XML document. The rules defined by the `ProbPiProcess.dtd` file correspond to the grammar of π_{pa}-calculus. In this way only XML documents representing valid π_{pa} processes are allowed.

The XML representation of π_{pa} processes is parsed using a Document Object Model (DOM) parser. The XML DOM is a programming interface for XML documents which defines the way an XML document can be accessed and manipulated from a programming language. With the XML DOM, a programmer can create an XML document, navigate its structure, and add, modify, or delete its elements. The DOM parser is used to load an XML document into the memory and to retrieve and manipulate the information in the XML document. The DOM represents a tree view of the XML document. The documentElement is the top-level of the tree. This element has one or many childNodes that represent the branches of the tree. A Node Interface is used to access the individual elements in the XML node tree. The W3C site `http://www.w3.org` provides a comprehensive reference of the XML DOM.

For example, a process executing an output action $\bar{x}y$ is represented by the following XML element:

```
<Send>
      <Channel>x</Channel>
      <Channel>y</Channel>
</Send>
```

The DOM parser creates a node for this XML element in the node tree representing the XML document, and two childNodes, one for each channel.

Figure 7.1 shows the XML node tree that is created for the following XML elements representing the process $\nu x(\bar{x}y)$:

```
<Restriction>
    <Channel>x</Channel>
    <Send>
```

```
        <Channel>x</Channel>
        <Channel>y</Channel>
    </Send>
</Restriction>
```

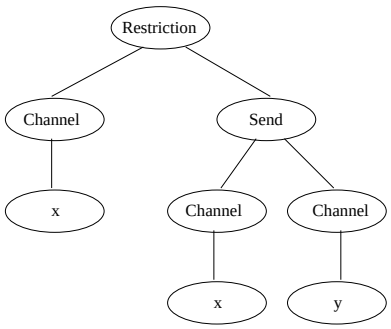

Fig. 7.1. An XML node tree

In order to generate the Java implementation of a π_{pa} process we process each node in the XML node tree representing the process and generate Java source code based on the type of the node.

The code for the Java implementation of π_{pa} is not included here for obvious space reasons, but it is available on the Web from the following URL:

`http://www.cse.psu.edu/~herescu/Implementation/`

In the event that this URL should become unavailable, we recommend to search the Internet for the new location, using the key words "Mihaela Herescu".

An outline of the encoding $[\![\cdot]\!]$ that is used to implement the π_{pa}-calculus is presented in the following.

Probabilistic choice

$$[\![\sum_{i=1}^{n} p_i x_i(y).P_i \,]\!] =$$

```
{ while (availChannels.size() == 0) {
      try {
          wait();
      } catch (InterruptedException e) {}
  }
  timeStamp++;
  Random gen = new Random();
  float pb = 1 - gen.nextFloat();
  float l = 0;
  PiChannel theChannel = (PiChannel)availChannels.get(0);
  float r = (float)(theChannel.prob);
  float normFactor = 0;
  for (int i=0; i < availChannels.size(); i++) {
      normFactor += ((PiChannel)availChannels.get(i)).prob;
  }
  if ((l/normFactor) < pb && pb < (r/normFactor)) {
      availChannels.remove(theChannel);
      return theChannel;
  }
  for (int i=1; i < availChannels.size(); i++) {
      theChannel = (PiChannel)availChannels.get(i);
      l = r;
      r += theChannel.prob;
      if ((l/normFactor) < pb && pb < (r/normFactor)) {
          boolean rem = availChannels.remove(theChannel);
          return theChannel;
      }
  }
}
```

The encoding of the probabilistic choice is executed within a synchronized method. The thread corresponding to the probabilistic choice process suspends when no input guards can be enabled.

Output action

$$[\![\bar{x}y]\!] = \{ \texttt{ x.send(y); } \}$$

Restriction

$$[\![\nu x P]\!] = \{ \texttt{ PiChannel x = new PiChannel(); } [\![P]\!] \}$$

Parallel

If our language is provided with a parallel operator, then we can just have a homomorphic mapping:

$$[\![P_1 \mid P_2]\!] = [\![P_1]\!] \mid [\![P_2]\!]$$

In Java, however, there is no parallel operator. In order to mimic it, a possibility is to define a new class for each process we wish to compose in parallel, and then create and start an object of that class. We use inner classes to define the parallel processes:

```
class processP1 extends Thread {
    public void run() {
            [[ P1 ]]
    }
}
class processP2 extends Thread {
    public void run() {
            [[ P2 ]]
    }
}
```

$$[\![P_1 \mid P_2]\!] = \{ \texttt{ new processP1().start(); new processP2().start(); } \}$$

Recursion

Remember that the process $rec_X P$ represents a process X defined as $X \stackrel{\text{def}}{=} P$, where P may contain occurrences of X. For each such process, we define the following class as an inner class:

```
class P extends Thread {
    public void run() {
            [[ P ]]
    }
}
```

Then define:

$$[\![rec_X P]\!] = \{ \texttt{ new P().start(); } \}$$

$$[\![X]\!] = \{ \texttt{ new P().start(); } \}$$

7.3 Related work

The implementation of different variants of the asynchronous π-calculus has been investigated in the literature since it could provide a very attractive intermediate language for compilers of concurrent languages.

Pict [40] is a strongly-typed concurrent programming language based on the π-calculus. The core language of Pict is an asynchronous, choice-free fragment of the π-calculus enriched with records and pattern matching. The compilation of Pict to C is based on the abstract machine specification defined by David Turner in [51]. The abstract machine is designed for implementation on a uniprocessor system and concurrent execution is simulated by interleaving the execution of processes. Each channel has a channel queue containing readers, writers or replicated readers processes suspended on that channel. Items in a channel queue are ordered since the process at the head of a channel queue is always waken up when a communication becomes possible. A run queue stores those processes which are currently runnable. The process at the head of the run queue is always executed first. In case there are other processes in the run queue when the process that is currently executing terminates, the next process in the run queue is executed. This procedure is repeated as long as the run queue is not empty. In order to

ensure that all runnable processes will eventually be executed, newly created processes are placed on the end on the run queue. Thus fairness is achieved by using FIFO channel queues and a round-robin policy for process scheduling. All channels which have been created are stored in a heap. For every channel, the heap stores all processes that are waiting to communicate on that channel. The parallel composition is encoded using C's sequential operator.

Nomadic Pict [55] is a concurrent programming language based the Nomadic π-calculus. The Nomadic π-calculus is an extension of the choice-free asynchronous π-calculus which provides support for distributed programming. The Nomadic π-calculus formally represents the notion of process mobility and system failure in a distributed network. Nomadic Pict is an extension of Pict enhanced with a notion of locations, agents, migration, distribution, and failures. See [52], [53] for more information about Nomadic π-calculi and Nomadic PICT.

[22] introduces a virtual machine specification and its implementation for a variant of the π-calculus - the TyCO calculus. TyCO is based on the asynchronous π-calculus, featuring built-in objects which interact by sending messages to shared communication channels. The semantics used for the design of the virtual machine is based on Turner's work on the π-calculus, but introduces support for objects, uses recursion rather than replication and explicitly defines the concurrency unit as the thread. The virtual machine is implemented as a byte-code emulator. [22] proposes an orthogonal extension of the TyCO calculus to provide support for distributed computations and code mobility.

Chapter 8

Conclusions and Future work

In this dissertation, we presented a probabilistic extension of the asynchronous π-calculus based on the model of probabilistic automata. We have argued that our calculus is more powerful than the asynchronous π-calculus by showing an example of a distributed problem that cannot be solved with the asynchronous π-calculus but can be solved with the probabilistic asynchronous π-calculus, namely the election of a leader in a symmetric network. We showed that the algorithm we proposed is correct, i.e. that the leader will eventually be elected, with probability 1, under every proper scheduler.

The probabilistic asynchronous π-calculus was then used as an intermediate language for a fully distributed implementation of the π-calculus. To this end, we proposed a uniform, compositional encoding of the π-calculus into the probabilistic asynchronous π-calculus and we showed its correctness with respect to a π_{pa} extension of a notion of testing semantics.

Finally, in order to prove that the probabilistic asynchronous π-calculus is a sensible paradigm for the specification of distributed algorithms, we defined a distributed implementation of π_{pa} in the Java language.

8.1 Future work

There are several possible future direction for the work presented in this dissertation.

One possible direction for this work is the study of formal tools for verification of properties of programs written in the probabilistic asynchronous π-calculus. One natural approach is to extend the Hennessy-Milner logic to the probabilistic automata model, possibly following the lines of Segala and Lynch in [49]. Another direction is the development of a proof system for properties expressed in this logic, namely a system which will allow inferences of the satisfiability relation between a π_{pa} process and a certain property.

For the generalized dining philosophers problem we have focused on the existence of a solution, and we have not address any efficiency issue. Clearly, efficiency is an important attribute for an algorithm. The evaluation of the complexity of our algorithms, and possibly the study of more efficient variants, are open topics for future research. Another open problem that seems worth exploring is the symmetric and fully distributed solution in the even more general case of hypergraphs-like connection structures, in which a philosopher may need more than two forks to eat.

The implementation of π_{pa} uses Java as the target language. This choice was convenient because of the concurrency primitives provided by Java, but a more efficient distributed implementation in a low-level language would be interesting to investigate.

The current implementation of π_{pa} in Java could be enhanced with several features. The encoding generator is currently command line driven. A graphical user interface would be a nice feature to add. Also, the XML representation of the π_{pa} processes could be enhanced by using an XSL stylesheet for formatting the XML documents into a HTML interface. Finally, it would be useful to have more examples of π_{pa} processes that can be used with the encoding generator tool.

References

[1] Roberto M. Amadio, Ilaria Castellani, and Davide Sangiorgi. On bisimulations for the asynchronous π-calculus. *Theoretical Computer Science*, 195(2):291–324, 1998. An extended abstract appeared in *Proceedings of CONCUR '96*, LNCS 1119: 147–162.

[2] J.A. Bergstra and J-W. Klop. Algebra of communicating processes with abstractions. *Theoretical Computer Science*, 33:77–121, 1985.

[3] Michele Boreale and Rocco De Nicola. Testing equivalence for mobile processes. *Information and Computation*, 120(2):279–303, 1995.

[4] Michele Boreale and Davide Sangiorgi. Some congruence properties for π-calculus bisimilarities. *Theoretical Computer Science*, 198(1,2):159–176, 1998.

[5] Gérard Boudol. Asynchrony and the π-calculus (note). Rapport de Recherche 1702, INRIA, Sophia-Antipolis, 1992.

[6] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, October 1971. Reprinted in *Operating Systems Techniques*, C.A.R. Hoare and R.H. Perrot, Eds., Academic Press, 1972, pp. 72–93. This paper introduces the classical synchronization problem of Dining Philosophers.

[7] Marie Duflot, Laurent Fribourg, and Claudine Picaronny. Randomized finite-state distributed algorithms as markov chains. In *Proceedings of 15th International Conference on Distributed Computing (DISC 2001)*, volume 2180, pages 240–254, Lisbon, Portugal, 2001. LNCS, Springer-Verlag.

[8] Marie Duflot, Laurent Fribourg, and Claudine Picaronny. Randomized dining philosophers without fairness assumption. In *Proceedings of 2nd IFIP International Conference on Theoretical Computer Science (TCS 2002)*. Kluver Academic, 2002.

[9] U. Engberg and M. Nielsen. A calculus of communicating systems with label-passing. *Report DAIMI PB-208, Computer Science Department, University of Aarhus*, 1986.

[10] N. Francez and M. Rodeh. A distributed abstract data type implemented by a probabilistic communication scheme. In *Proc. 21st Ann. IEEE Symp. on Foundations of Computer Science*, pages 373–379, 1980.

[11] Oltea Mihaela Herescu and Catuscia Palamidessi. Probabilistic asynchronous π-calculus. In Jerzy Tiuryn, editor, *Proceedings of FOSSACS 2000 (Part of ETAPS 2000)*, Lecture Notes in Computer Science, pages 146–160. Springer-Verlag, 2000.

[12] Oltea Mihaela Herescu and Catuscia Palamidessi. On the generalized dining philosophers problem. In *Proceedings of the Twentieth ACM Symposium on Principles of Distributed Computing (PODC 2001)*, pages 81–89, Newport, RI, 2001.

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

[14] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In Pierre America, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 512 of *Lecture Notes in Computer Science*, pages 133–147. Springer-Verlag, 1991.

[15] He Jifeng, M.B. Josephs, and C.A.R. Hoare. A theory of synchrony and asynchrony. In *Proceedings of IFIP Working Conference on Programming Concepts and Methods*, pages 459–478, 1990.

[16] Bengt Jonsson, Kim Larsen, and Wang Yi. Probabilistic extensions of process algebras. In J. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebras*. Elsevier, 2001.

[17] Bengt Jonsson and Wang Yi. Compositional testing preorders for probabilistic processes. In *Proceedings of the Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 431–441, San Diego, California, 1995. IEEE Computer Society Press.

[18] Yuh-Jzer Joung. Two decentralized algorithms for strong interaction fairness for systems with unbounded speed variability. *Theretical Computer Science*, 243(1-2):307–338, 2000.

[19] Yuh-Jzer Joung and Scott A. Smolka. Strong interaction fairness via randomization. *IEEE Transactions on Parallel and Distributed Systems*, 9(2):137–149, 1998.

[20] F. Knabe. A distributed protocol for channel-based communications with choice. In D. Etiemble and J.-C. Syre, editors, *PARLE '92: Parallel Architectures and Languages Europe*, volume 605. LNCS, Springer-Verlag.

[21] Daniel Lehmann and Michael O. Rabin. On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In *Conference Record of the Eighth annual ACM Symposium on Principles of Programming Languages*, pages 133–138. ACM, ACM, January 1981.

[22] Lus Miguel Barros Lopes. *On the Design and Implementation of a Virtual Machine for Process Calculi*. PhD thesis, Departamento de Cincia de Computadores, Universidade do Porto, December 1999.

[23] Nancy Lynch, Isaac Saias, and Roberto Segala. Proving time bounds for randomized distributed algorithms. In *Symposium on Principles of Distributed Computing (PODC '94)*, pages 314–323, New York, SA, August 1994. ACM Press.

[24] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer-Verlag, New York, NY, 1980.

[25] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[26] Robin Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, 1999.

[27] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40 & 41–77, 1992.

[28] Robin Milner, Joachim Parrow, and David Walker. Modal logics for mobile processes. *Theoretical Computer Science*, 114(1):149–171, 1993.

[29] Uwe Nestmann. What is a 'good' encoding of guarded choice? In Catuscia Palamidessi and Joachim Parrow, editors, *Proceedings of EXPRESS '97: Expressiveness in Concurrency (Santa Margherita Ligure, Italy, September 8–12, 1997)*, volume 7 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 1997. Full version to appear in *Information and Computation*.

[30] Uwe Nestmann. On the expressive power of joint input. In Catuscia Palamidessi and Ilaria Castellani, editors, *EXPRESS '98: Expressiveness in Concurrency*, volume 16.2 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science B.V., 1998.

[31] Uwe Nestmann. What is a 'good' encoding of guarded choice? *Journal of Information and Computation*, 156:287–319, 2000. An extended abstract appeared in the *Proceedings of EXPRESS'97*, volume 7 of *ENTCS*.

[32] Uwe Nestmann and Benjamin C. Pierce. Decoding choice encodings. In Ugo Montanari and Vladimiro Sassone, editors, *Proceedings of CONCUR '96: Concurrency Theory (7th International Conference, Pisa, Italy, August 1996)*, volume 1119 of *Lecture Notes in Computer Science*, pages 179–194. Springer-Verlag, 1996. Full version to appear in *Information and Computation*.

[33] Uwe Nestmann and Benjamin C. Pierce. Decoding choice encodings. *Journal of Information and Computation*, 163:1–59, 2000. An extended abstract appeared in the *Proceedings of CONCUR'96*, volume 1119 of *LNCS*.

[34] Rocco De Nicola and Matthew C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34(1-2):83–133, 1984.

[35] Catuscia Palamidessi. Comparing the expressive power of the synchronous and the asynchronous π-calculi. *Mathematical Structures in Computer Science*. To Appear.

[36] Catuscia Palamidessi. Comparing the expressive power of the synchronous and the asynchronous π-calculus. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 256–265, Paris, France, 1997.

[37] Catuscia Palamidessi and Oltea Mihaela Herescu. A randomized distributed encoding of the π-calculus with mixed choice. In *Proceedings of 2nd IFIP International Conference on Theoretical Computer Science (TCS 2002)*. Kluver Academic, 2002.

[38] Joachim Parrow. Chapter to appear. In Bergstra, Ponse, and Smolka, editors, *Handbook of Process Algebra*. Elsevier.

[39] Joachim Parrow and Peter Sjodin. Multiway synchronization verified with coupled simulation. In Rance Cleaveland, editor, *CONCUR '92: 3rd International Conference on Concurrency Theory*, volume 630, pages 518–533. LNCS, Springer-Verlag, 1992.

[40] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. 1995.

[41] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. The MIT Press, 1998.

[42] Michael O. Rabin and Daniel Lehmann. On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In A. W. Roscoe, editor, *A Classical Mind: Essays in Honour of C.A.R. Hoare*, chapter 20, pages 333–352. Prentice Hall, 1994. An extended abstract appeared in the *Proceedings of POPL'81*, pages 133-138.

[43] John H. Reif and Paul G. Spirakis. Real-time synchronization of interprocess communications. *ACM Transactions on Programming Languages and Systems*, 6(2):215–238, 1984.

[44] W. Reisig. Petri nets. *EATCS Monographs on Theoretical Computer Science*, 1983.

[45] Davide Sangiorgi. π-calculus, internal mobility and agent-passing calculi. *Theoretical Computer Science*, 167(1,2):235–274, 1996.

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

[47] Roberto Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1995. Available as Technical Report MIT/LCS/TR-676.

[48] Roberto Segala. Testing probabilistic automata. In Ugo Montanari and Vladimiro Sassone, editors, *Proceedings of CONCUR '96: Concurrency Theory, 7th International Conference*, volume 1119 of *Lecture Notes in Computer Science*, pages 299–314, Pisa, Italy, 1996. Springer-Verlag.

[49] Roberto Segala and Nancy Lynch. Probabilistic simulations for probabilistic processes. *Nordic Journal of Computing*, 2(2):250–273, 1995. An extended abstract appeared in *Proceedings of CONCUR '94*, LNCS 836: 22–25.

[50] Yih-Kuen Tsay and Rajive L. Bagrodia. Fault-tolerant algorithms for fair interprocess synchronization. *IEEE Transactions on Parallel and Distributed Systems*, 5(7):737–748, 1994.

[51] David N. Turner. *The Polimorphic Pi-calculus: Theory and Implementation*. PhD thesis, University of Edinburgh, 1995.

[52] Asis Unyapoth. *Nomadic Pi Calculi: Expressing and Verifying Infrastructure for Mobile Computation*. PhD thesis, Appeared as Technical Report 514, Computer Laboratory, University of Cambridge, 2001.

[53] Asis Unypoth and Peter Sewell. Nomadic pict: Correct communication infrastructure for mobile computation. In *Proceedings of POPL 2001*, 2001.

[54] Rob J. van Glabbeek, Scott A. Smolka, and Bernhard Steffen. Reactive, generative and stratified models of probabilistic processes. *Information and Computation*, 121(1):59–80, 1995.

[55] Pawel Wojciechowski. *Nomadic Pict: Language and Infrastructure Design for Mobile Computation*. PhD thesis, Appeared as Technical Report 492, Computer Laboratory, University of Cambridge, 2000.

Appendix A

Transition system for the π_{pa}-calculus

Table A.1 presents an equivalent transition system for the π_{pa}-calculus where no assumptions on the bound variables are made. Note that the side condition on the rule SUM is necessary for treating cases like $1/2\ x(y).\mathbf{0} + 1/2\ x(y).\mathbf{0}$. This condition could be eliminated by assuming that the transition groups are multiset instead than sets.

$$\textsc{Sum}\quad \sum_i p_i\alpha_i.P_i\ \{\xrightarrow[p'_i]{\alpha_i} P_i\}_i \qquad p'_i = p_i / \textstyle\sum_{j:\alpha_j=\alpha_i} p_j \equiv P^i\, p_j$$

$$\textsc{Out}\quad \bar{x}y\ \{\xrightarrow[1]{\bar{x}y} \mathbf{0}\}$$

$$\textsc{Open}\quad \frac{P\ \{\xrightarrow[1]{\bar{x}y} P'\}}{\nu y P\ \{\xrightarrow[1]{\bar{x}(y)} P'\}} \qquad x \neq y$$

$$\textsc{Res}\quad \frac{P\ \{\xrightarrow[p_i]{\mu_i} P_i \mid i \in I\}}{\nu y P\ \{\xrightarrow[p'_i]{\mu_i} \nu y P_i \mid i \in I \text{ and } y \notin fn(\mu_i)\}} \qquad \begin{array}{l}\exists i \in I.\ y \notin fn(\mu_i),\\ \forall i \in I.\ y \notin bn(\mu_i), \text{ and}\\ \forall i \in I.\ p'_i = p_i / \sum_{j:y\notin fn(\mu_j)} p_j\end{array}$$

$$\textsc{Par}\quad \frac{P\ \{\xrightarrow[p_i]{\mu_i} P_i\}_i}{P \mid Q\ \{\xrightarrow[p_i]{\mu_i} P_i \mid Q\}_i} \qquad \forall i.\ bn(\mu_i) \cap fn(Q) = \emptyset$$

$$\textsc{Com}\quad \frac{P\ \{\xrightarrow[1]{\bar{x}y} P'\} \qquad Q\ \{\xrightarrow[p_i]{\mu_i} Q_i \mid i \in I\}}{\begin{array}{l}P \mid Q\ \{\xrightarrow[p_i]{\tau} P' \mid Q_i[y/z_i] \mid i \in I \text{ and } \mu_i = x(z_i)\}\\ \quad \cup\{\xrightarrow[p_i]{\mu_i} P \mid Q_i \mid i \in I \text{ and } \mu_i \neq x(z_i)\}\end{array}}$$

$$\textsc{Close}\quad \frac{P\ \{\xrightarrow[1]{\bar{x}(y)} P'\} \qquad Q\ \{\xrightarrow[p_i]{\mu_i} Q_i \mid i \in I\}}{\begin{array}{l}P \mid Q\ \{\xrightarrow[p_i]{\tau} \nu y(P' \mid Q_i[y/z_i]) \mid i \in I \text{ and } \mu_i = x(z_i)\}\\ \quad \cup\{\xrightarrow[p_i]{\tau} P \mid Q_i \mid i \in I \text{ and } \mu_i \neq x(z_i)\}\end{array}}$$

$$\textsc{Cong}\quad \frac{P \equiv P' \qquad P'\ \{\xrightarrow[p_i]{\mu_i} Q'_i\}_i \qquad \forall i.\ Q'_i \equiv Q_i}{P\ \{\xrightarrow[p_i]{\mu_i} Q_i\}_i}$$

Table A.1. Alternative formulation of the probabilistic transition system for the π_{pa}-calculus.

Appendix B

An example of generating a Java implementation for a π_{pa} process

The π_{pa} process used in this example is $P = (\nu x1)(\nu x2)(P_1 \mid P_2 \mid Q)$, where $P_1 = \bar{x_1}y$, $P_2 = \bar{x_2}z$, $Q = 1/3x_1(v_1).0 + 2/3x_2(v_2).R$, $R = \bar{w}q$.

The XML representation of these π_{pa} processes is found in the file `ProbPiProcess.xml`. After running the encoding generator on the XML document, a Java class called

`ProbPiProcess.java` is created. This class contains the Java implementation of the π_{pa} process P.

The following sequence of commands are executed to generate and run the Java implementation of process P. If the `EncodingGenerator` is run without any command line arguments, then a message showing the usage is output. Note that several XML files can processed in the same command.

```
C:\JavaEncoding>javac encoding\java\*.java

C:\JavaEncoding>java encoding.java.EncodingGenerator

Usage:EncodingGenerator -outputDir <outputDirectory> -inputFiles
<inputFilename 1> <inputFilename2> ... <inputFilenameN>

where:

     <outputDirectory> is the relative path from the current
directory where the Java encoding files will be generated

     the arguments <inputFilename> are the names of the xml files
containing the probabilistic pi processes for which the Java
encoding will be generated

C:\JavaEncoding>java encoding.java.EncodingGenerator -outputDir
encoding\samples \output -inputFiles
encoding\samples\ProbPiProcess.xml

C:\JavaEncoding>javac encoding\samples\output\ProbPiProcess.java
```

```
C:\JavaEncoding>java encoding.samples.output.ProbPiProcess

Sent value y on pi channel x1

Sent value z on pi channel x2

Probabilistic choice process added request to pi channel x1

Pi channel x1 processed request

Message y removed from available messages of pi channel x1

Probabilistic choice process added request to pi channel x2

Pi channel x2 processed request

Message z removed from available messages of pi channel x2

Probabilistic choice: selected channel x2 with probability 0.67

Invalid timestamp value detected. Resending consumed message: Sent
value y on pi channel x1

Sent value q on pi channel w
```

Note that process P contains a probabilistic choice. Therefore, if we run the `ProbPiProcess` class several times, then different behaviors are obtained. The other possible outcome for the probabilistic choice in P is shown in the following:

```
C:\JavaEncoding>java encoding.samples.output.ProbPiProcess

Sent value y on pi channel x1

Sent value z on pi channel x2

Probabilistic choice process added request to pi channel x1

Pi channel x1 processed request

Message y removed from available messages of pi channel x1

Probabilistic choice process added request to pi channel x2

Pi channel x2 processed request
```

```
Message z removed from available messages of pi channel x2

Probabilistic choice: selected channel x1 with probability 0.33

Invalid timestamp value detected. Resending consumed message: Sent
value z on pi channel x2
```

Note that the encoding generator uses the Java API for XML Processing (JAXP) and the Crimson implementation of the Java XML parser. JAXP is available to download from `http://java.sun.com/xml/download.html`.

Crimson can be found at `http://xml.apache.org/crimson/`. The following files have to be added to the classpath in order to compile and run the above classes: crimson.jar and jaxp-api.jar.

Vita

Oltea Mihaela Herescu was born in Bucharest, Romania, on January 10, 1975. After graduating from the oldest high-school in Romania, the St. Sava National College, in 1993, she attended the Politehnica University of Bucharest, where she studied computer science and engineering, and obtained her BS in Computer Science in Spring 1998. She started the graduate program at the Pennsylvania State University in Fall 1998, from where she received a Master of Science degree in Summer 2000 and a PhD in Computer Science and Engineering in Fall 2002. Mihaela's research interest is the theory of concurrency and distributed computing.