

Asynchronous Process Calculi for Specification and Verification of Information Hiding Protocols

Romain Beauxis

▶ **To cite this version:**

Romain Beauxis. Asynchronous Process Calculi for Specification and Verification of Information Hiding Protocols. Cryptography and Security [cs.CR]. Ecole Polytechnique X, 2009. English. NNT : . tel-00772693

HAL Id: tel-00772693

https://pastel.hal.science/tel-00772693

Submitted on 11 Jan 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

École Polytechnique
Thèse de Doctorat
Spécialité Informatique

Asynchronous Process Calculi for Specification and Verification of Information Hiding Protocols

Présentée et soutenue publiquement par

Romain Beauxis

le 4 mai 2009

devant le jury composé de

Maurizio	Gabbrielli	Rapporteur
Jean	Goubault-Larrecq	
Michel	Hirshowitz	
Pierre	Lescanne	Rapporteur
Jean-Jacques	Lévy	
Catuscia	Palamidessi	Directeur de thèse
Joachim	Parrow	Rapporteur
Daniele	Varacca	

La Nature est un temple où de vivants piliers
Laissent parfois sortir de confuses paroles ;
L'homme y passe à travers des forêts de symboles
Qui l'observent avec des regards familiers.

Comme de longs échos qui de loin se confondent
Dans une ténébreuse et profonde unité,
Vaste comme la nuit et comme la clarté,
Les parfums, les couleurs et les sons se répondent.

Charles Baudelaire, `Correspondances`
in *Les Fleurs du mal*

Contents

Outline of the thesis **1**

Introduction **3**

Contributions **9**

I Asynchronous communications **11**

1 Concurrent communication models. **17**
1.1 The various π-calculi 18
1.2 Bisimulations and asynchronous communications 26

2 Communicating trough buffers **33**
2.1 Buffers, stacks, queues and bags 34
2.2 The $\pi_{\mathfrak{B}}$-calculus 36
2.3 Encodings from and to the π_a-calculus and $\pi_{\mathfrak{B}}$-calculus 37
2.4 Impossibility result for the other types of buffer 42

II Asynchronous behavior **45**

3 Equivalent behavior in the π-calculus **49**
3.1 The late semantics 50
3.2 Bisimilarities and congruences 52
3.3 The asynchronous case 54

4 Causality and asynchrony: speeding up the scheduler **63**
4.1 Bunched Labeled Transition Systems 64
4.2 Bunched transition system for the π_a-calculus 70
4.3 Further discussions 73

III Asynchronous probabilistic executions 77

5 Probabilistic Concurrent Constraint Programming. 81
5.1 The Constraint System 82
5.2 The language 83
5.3 Application: Crowds anonymous routing 86

6 Denotational semantics for the CCP+P 91
6.1 Valuations 92
6.2 Vector cones of simple valuations 103
6.3 Denotational semantics 108

IV Application: the dinning cryptographers reloaded 111

7 Anonymous protocols: the dining cryptographer 117
7.1 The nondeterministic approach to anonymity 118
7.2 The dining cryptographers' problem 119

8 Toward an automated probabilistic anonymity checker 125
8.1 Probabilistic automata 126
8.2 Our framework for probabilistic anonymity 126
8.3 The various notions of anonymity 130
8.4 Toward an automatic probabilistic checker 139

Outline of the thesis

The work presented in this document in an account of my work as a PhD student at LIX, École Polytechnique, in the COMETE team under the supervision of Catuscia Palamidessi. During these studies, I have been in interested in the various aspects of concurrency covered by the COMETE team activities.

The initial goal of my thesis was to investigate the aspects related to process calculi based formalisms to express and analyze Security Protocols. The ultimate goal was to makes some advances towards the automatic verification of security properties. In particular, I was interested in information-hiding protocols which require no cryptography, but normally use randomized mechanisms and therefore exhibit probabilistic behavior.

Information hiding protocols are used typically in networks, and they are run by parties that reside in different locations of the system, and therefore interact asynchronously.

The first work that I did was to try to give a correct meaning to the various notions of formal asynchronous communications used in various models, in particular between the field of concurrency and the field of distributed computing, where this was a recurrent question. These results are presented in the first part of this document.

Being interested in the formal aspects of information-hiding problems, I took part in the preparation of the journal version of [BP09], and started preparing an automated probabilistic anonymity checker based on the formalism presented in this document. This lead to an initial draft of an implementation presented in `http://vamp.gforge.inria.fr/`. The formalism for this analysis is presented in the fourth part of this document.

Another aspect of the verification of information hiding properties is that it requires to compute the probabilities of the possible outcomes for each scheduler. For this reason, this application quickly turned out to be highly inefficient. However, in an asynchronous system, a lot of transitions are confluent, which means that when evaluating a process, it is only necessary to choose one of the two confluent branches.

Hence, I have worked on formalizing the possible optimizations based on the

possible confluent computations. This work is presented in the second part of the document.

Another interesting aspects of probabilistic protocols is the possibility to consider infinite runs. By doing such consideration, it is possible to verify the correction of some probabilistic protocols. For instance, in the case of the Crowds routing protocol, presented in Section 5.3, the protocol is considered correct because the probability of running into an infinite execution is null, hence the message will eventually be delivered.

For this reason, I got interested in extending the meaning of a asynchronous probabilistic computations to the case of an infinite execution. As a matter of fact, the combination of infinite computation, confluence and probability is not easy to treat in the general case.

The problem of confluence in concurrency is solved in an elegant way in an asynchronous paradigm called Concurrent Constraint Programming (CCP). Hence, I decided to study infinite computations in a probabilistic version of CCP. The problem, however, is that the meaning of the result of an infinite probabilistic computation was still an open problem also in that context.

Hence, I studied a possible way to define this result, using the notion of valuations and sober spaces, and applied it to give a denotational semantics to probabilistic CCP, including infinite computations. This work is presented in the third part of the document.

I have chosen a specific order for the various parts of this document that follows the various formal models that are used, in order to present each result along with the corresponding formalism.

- In the first and second parts, I present the formal concurrent models, and in the particular asynchronous ones.
- In the third part, I present the probabilistic CCP. This part also presents mathematic structures for the representation of infinite probabilistic executions.
- Eventually, an application of both asynchronous and probabilistic models to the case of probabilistic information hiding is presented in the fourth part.

Introduction

Asynchronous communications are characterized by the impossibility for a communicating agent to control both transmission and reception of the messages. In real life communication, one may think for instance of mail communication when one cannot know with certainty when a message will be received and if he will get an answer. This assumption has important consequences in the behavior of the agents, and on the algorithmic possibilities.

The first issue studied in this thesis is the relations between the various possible formal models for representing asynchronous communications: do they all characterize the same notion ? What relations, in terms of expressiveness, can we make between them ? I give an answer to this question in terms of behavioral relations between the algebraic models such as the asynchronous pi-calculus and the models where communication occurs through buffers.

Another interesting phenomenon of concurrent systems is the possibility for two executions to be confluent due to the fact that they represent different interleavings: for any order of execution of the concurrent threads, the result of the computation is the same. I apply this property in order to define conditions for evaluating only one of the possible confluent executions, thus reducing the search space greatly.

The study of information hiding problems requires to extend these models to the case of probabilistic computations, where the model associates probabilities to each elements of the sets of possible transitions. In the case of strong asynchronous confluence, I study the result of an infinite computation, and give general conditions under which this result can be decomposed to elementary probabilities.

At the end of the document, I present an application of probabilistic and asynchronous models to the study of leak of information in an anonymous protocol, the Dining Cryptographers.

Asynchronous communication models

Various models for asynchronous communicating systems have been proposed in the literature, but they do not necessarily coincide. The two most common classes of models consist in:

- Restraining the control over the emission and reception of messages. This is the case for the asynchronous pi-calculus where for instance it is not possible to start the execution of a process exactly when a message has been transmitted.

- Using an asynchronous communication framework. This is the case in the field of distributed computing, where the messages are placed in a buffer that is responsible for their delivery.

The first part of this work tries to bridge the gap between the two classes of models mentioned before. I propose a hybrid version of the pi-calculus (π-calculus, [MPW92]) where the processes are processes of the synchronous π-calculus, but communication is forced to happen through (syntactic) buffers.

The processes of the π-calculus with buffers and those of the asynchronous π-calculus (π_a-calculus, [HT91, Bou92]) are then compared using the notion of bisimilarity. The bisimilarity is used here for searching whether for any process of each language, there exists a process of the other language that can behaves exactly as the other one.

For this correspondence, we use a specific notion of bisimilarity, the asynchronous bisimilarity. This variant of the bisimilarity can be found in [ACS98] and [HT91]. The difference with usual bisimilarity is that this relation does not require a bisimilar process to match all the transitions of the other process, but only those which are relevant in an asynchronous context.

We establish that when adding unordered buffers (aka bags) as communication medium to the π-calculus, the resulting language is asynchronously bisimilar to the π_a-calculus. In particular, the behavior of processes using mixed choice and output prefix in the π-calculus, which are the essential difference with π_a-calculus, can be imitated by processes of the π_a-calculus when communicating through these buffers.

We also show that a similar correspondence, using asynchronous bisimilarity, does not hold when the buffers used for communicating are ordered and follow the FIFO and LIFO strategies. This suggests that the buffers to represented by the communications in the π_a-calculus are the unordered buffers. Communication happening through ordered buffers, namely FIFO and LIFO buffers, should need an intermediate language, in terms of communication mechanisms, between the π-calculus and the π_a-calculus, like the π-calculus with output prefix, or mixed choice.

Asynchronous confluence and optimization

An important aspect of concurrency is the fact that it implies confluent executions. This in increased in the case of asynchronous communications (via bags, i.e. unordered buffers). Intuitively, the order of execution of two send actions does not matter if the communication mean is a bag.

When trying to establish a correspondence between two processes of the π_a-calculus, in terms of traces or bisimilarity, these confluences are very interesting in order to reduce the search space.

For instance, since it is always possible to switch output actions, if a process can do a trace that begins with two such actions, it can also perform all the trace where the output action are switched, and the resulting process is exactly the same.

In the second part of the thesis, I propose a study of the equivalent behaviors under asynchronous communications. In particular, I propose the definition of a bunched transition system for the π_a-calculus, where the evaluation of a process is constrained in order to reduce the interleavings generated by asynchronous execution steps while preserving the relevant information that we want to observe.

We prove that the bisimilarity induced by such a transition system is included into the bisimilarity of the original system. This means that if two processes are bisimilar for the bunched bisimilarity, they are also bisimilar for the original bisimilarity. This implication is proved by establishing a correspondence between the confluent executions and the bunched transitions. It is also proved that if a process succeeds a test for the testing semantics, if and only if the same process also succeeds the test for the bunched transition system.

The results in this part lead to the intuition that in the π_a-calculus, order between output actions if not relevant unless we enforce communication. So, in a sense, a certain sequence of output actions is representative of all the sequences with swapped order. A similar property holds for sequences of output actions followed by input actions, in the sense that an output followed by an input is representative also of the sequence where the order is inverted.

Probabilistic executions, confluence and infinite computations

Another important class of models are those of the probabilistic executions. These models are useful, for instance, for the quantitative analysis of the executions of a process or protocol. This is in particular the case for the study of information hiding problems, as presented in the fourth part of this document.

Another motivation for probabilistic models is the possibility to consider properties of a protocol for infinite computations. This is particularly interesting when the probability of the correct executions of a protocol is one under an infinite computation. In this case, this means that the protocol is correct in the sense that any execution will eventually reach a correct state.

For these reasons, I propose in the third part to study a probabilistic extension of an asynchronous language, the Concurrent Constraint Programming (CCP). This extension adds an internal probabilistic choice, which decides internally a branch of execution among the possible choices. In CCP, the execution of a process uses a constraint store for establishing an asynchronous form of communication where each agent can add constraints to the store or deduce constraints from the current store, but are not able to remove a constraint.

CCP is confluent. One of the results of my investigations is to study the notion of confluence for the probabilistic extension. To this purpose, the results of the executions of a program are represented as sets of probabilistic states. Each state and its associated probability represent the current probability for the program to be in the given state after an execution for a given interleaving.

By mapping these execution states to a set of functions used for measuring the opens of a topological space, it is also possible to define a meaning for the infinite execution of a process. In this part, I give conditions on the constraint system such that this meaning can also be represented as a set of states along with individual probabilities for these states.

Using this decomposition, I propose a denotation semantics for this language, where each program is represented by a vector cone on the vector space of the valuations. These cones can be obtained by a fixed point construction, and represent a linear closure operator on the vector space.

Application: information hiding analysis and the Dining Protocol

In the last part, I propose an illustration of all the considerations and models used in the previous parts. The example is the dining cryptographers protocol, where we want to ensure the probabilistic anonymity of the payer. In this part, I explain the basic models and results about this analysis, and relate the possible use of the results in my thesis as a basis for an automated probabilistic anonymity checker.

This example illustrates the fact that probabilistic asynchronous models are relevant to capture some notions, such as probabilistic anonymity. Furthermore, the scheduler is also identified as a possible vector to leak information. Hence, the need to analyze the possible executions under any scheduler and check the resulting probability distributions, and not only the maximum and

minimum distribution over any scheduler. Finally, the first steps toward an implementation of an automated analysis tool for this purpose showed the need for important optimizations in this exploration, which lead to the results presented in the second part.

Contributions

The structure of this document tries to separate the personal contributions from the results already present in the literature. Each part of the document is divided in two chapters. This first chapter should then present the preliminaries and the background of the work. Then the second chapter presents the new contributions. The two exceptions are in Chapter 3, which presents new results in Section 3.3.2, and the fourth part, where the results are almost all present in the literature. In this part, the results help to show an example of the considerations mentioned in the previous parts.

In details, the major contributions are the following.

In the first part:

- Theorem 2.3.1 and Theorem 2.3.2 show the equivalence between the π_a-calculus and a synchronous π-calculus communicating through (syntactic) unordered buffers.
- Theorem 2.4.1 and Theorem 2.4.2 show the impossibility to perform the same correspondence when trying to encode a synchronous calculus communicating through queues (FIFOs) or stacks (LIFOs).

The results in this part have been published in [BPV08].

In the second part:

- Theorem 3.3.1 and Theorem 3.3.2 prove that when trying to prove the late bisimilarity for the asynchronous π-calculus, the observation of the internal communications that occur on a public channel and the substitution after this reception are not relevant.
- Theorem 4.1.2 proves that, under the conditions for a bunched transition system over an original transition system, the bisimilarity induced by the bunched one implies the bisimilarity on the original one.
- Definition 4.2.1 and Theorem 4.2.1 establish a bunched transition system for the π_a-calculus.

The results of this part are quite recent and will be submitted for publication after the defense.

In the third part:

- Definition 5.2.1 presents a a probabilistic extension of the Concurrent Constraint Programming by adding internal probabilistic choices.
- Definition 6.3.1 presents a denotational semantics for this probabilistic language using linear closure operators on topological simple valuations identified by a vector cone of fixed (or resting) points. Theorem 6.3.1 and Theorem 6.3.2 shows that this semantics is sound and fully abstract with regard to the operational semantics.
- Theorem 6.1.3 proves a very general decomposition result for the valuations based on a quotient of its image and the opens of the topology. Using this result, Theorem 6.1.5 proves that any valuation on a lattice whose order can be extended to a total well-founded order is decomposable as a simple valuation.

The results in this part appeared in [Bea09] and submitted for publication to the Journal of Theory and Practice of Logic Programming.

In the fourth part, the contribution consists mainly in the systematization and clarification of notions and results that already appeared in [BP05], their formal proof, and the illustration of an approach to the automatic verification of the notion of anonymity. Also, this part provides an example for the application of the process-calculi concepts that we have developed in previous parts of the thesis . The content of this part have appeared also in [BCPP08] and [BP09].

Part I

Asynchronous communications

Introduction

Communication, in real life, happens under various forms and since a very long time. Originally, there was oral communication. Then there was written communication mostly in the form of letters and messages sending. One may also think of the various imaginative communications medium or language that have been used in various cultures, such as using smoke or whistles for communicating through long distance, as the native Americans used to do.

Then, with modern technologies, some new forms of communication progressively appeared. First, the telegraphs and phones, followed by radio communications and, more recently e-mails and instant messengers.

When trying to characterize the asynchronism of these various forms of communications, several questions may arise:

- Will the message be received ?
- When will the message be received ?
- Will I know when it is received ?
- Will I get an answer ?

All these questions characterize various form of asynchronism in the communication. All of them are related to the interaction between the two parts of the communication. Hence, when establishing a communication model, one has to consider two *agents*, a *sender* and a *receiver*. Then, an item, called a *message* will be exchanged between the sender and the receiver. This will stand for the communication action.

In the case of an oral conversation, most of the time, the sender and receiver are close to each other, and can see the other part. In such a situation, they both know almost instantaneously that the message have been sent and received. It is even possible to supperpose various messages, in a *full duplex* fashion, so that they are both sender and receiver. This is even the most common form of oral communication. Hence, oral communication defines intuitively the notion of synchronous communication.

Other forms of communications do not usually qualify as being synchronous. But it would be wrong to consider only synchronous or asynchronous communications. Indeed, various degree of asynchrony can be identified. For instance, most of the time, a phone call provides immediate delivery to the receiver. And both the sender and receiver, though usually far away from each other, know that delivery should be instantaneous and expect an acknowledgement from the receiver, either via a simple sound, like “huh”, or via an answer. They may also speak in full-duplex, though it is less easy in this case. However, this form of communication is definitely not synchronous since message delivery can take time, in the case of long distance calls for instance.

On the other hand, when communicating by mails, either paper mail or electronic mail, all the usual synchronous properties are lost. When sending a mail, you cannot know wether it will be received, when it will be received, if it will be read, and if you will get an answer. Instant messengers work as an intermediate between phone and mails. Indeed, when using an instant messenger, each message is sent with a similar synchronism as for electronic mail, but an answer is often expected quite quickly, and the lack of an answer after some short time often means that the receiver was absent. In other words, long delay is assumed to be a loss in the communication. Similarly, most of the instant messenger protocols and clients provide an acknowledgement mechanism, which informs the sender that the receiver has received the message, as well as an alert mechanism which notifies you when the sender is writing a message.

Synchronism can also be studied from the point of the communication medium. Indeed, the communication medium for a phone conversation consists of a path between the two agents. This path is discovered when establishing the communication, and is maintained during the whole conversation. Since all the messages are sent through the same path, one after the other, it can be assumed that the order between them is always preserved.

However, since the delay can be quite long, it is difficult to maintain a synchronism between messages from each other. In case of long delays, it often happens that the conversations splits, when each of the two persons start a new sentence at the same moment, but only get each other's sentence with a delay. In such a situation, it is often convenient to establish a synchronization protocol, which consist of telling the other person when you are done with your sentence and wait for his answer or acknowledgement before going on. This is how citizens' band radio communication work most of the time.

The drawback of the telephone communication medium is that it assumes that each call has its own path, hence needing large communication bandwidth. Multiplexing technologies, that can merge several calls into one single stream exist, but they can only apply to calls that have an important common path. It also needs a centralized network, which is sensitive to failures or destruction of the central nodes.

On the contrary, in the case of internet communications, the protocol consists of packets which are sent through the network. There is no underlying notion of communication path, not even communication streams. Each agent in the network knows some agents to which it can send its packets. And when an agent receives a packet that is not for him, he selects a new node according to the packet's destination and send it to him. In this case, it is not even possible to assume that packets will be received in the same order as they were emitted.

Synchronization mechanisms can be defined on top of this basic medium. The two most commons protocols on the internet are *UDP* and *TCP*. In the case of UDP, also referred as the "send and pray" protocol, messages are simply

delivered to the next communication node, and no information is reported at all. In the case of TCP, each message, equipped with a sequence number, is acknowledged by the receiver. The receiver may also request that a message is sent again, in case it was lost during the communication.

However, the synchronization mechanisms have a price, since a part of the communication bandwidth is used for the acknowledgements. Also, these mechanisms are not a perfect imitation of real synchronous communications: since they include extra communication steps, they also introduce new situations that are not possible in the original case.

Models in mathematics aim at being abstract representations of real life objects and events. Hence, these considerations should be present in the various communication models that we will present. In particular, we have discussed in this section the following issues:

- Reception done in the same order as sending, i.e. the possibility to send sequentially
- Simultaneous conversations without confusion, i.e. the possibility to decide between sending or receiving, as well as knowing when the receiver has received your message.
- Synchronization protocols approximating real synchronism, i.e. the behavioral equivalence between original communication and its approximation.

Both Concurrency Theory and Distributed Computing are concerned with the notion of asynchrony, but they have different approaches. In Concurrency Theory, at least in the most recent proposals, asynchronous communications is characterized by the fact that the send action cannot be prefix of another process, i.e. we cannot specify directly that a certain activity starts after the send action is completed. In Distributed Computing, asynchronous communication is defined by the reception properties, more specifically, by the fact that the bound in time for receiving a message is infinite.

These two notions have been intuited for a long time as being related to the same phenomenon, however, it was never studied formally. In this part, we study the various definitions of synchronous and asynchronous communications, and try to bridge the gap between the two approaches.

In chapter 1, we introduce the notion of synchronous and asynchronous communications, with regard to real life situations, and try to get from this presentation the intuitions about what phenomenon the abstract representation of communications must model in order to be claimed synchronous or not.

In Chapter 2, we prove a correspondence between those two different approaches to asynchronous communications in the case of unordered buffers, which shows that both models represent the same phenomenon. Further investigations showed that there are correspondences in the cases of queues and stacks (FIFO and LIFO buffers) require additional control mechanisms.

Chapter 1

Concurrent communication models.

This chapter introduces a family of concurrent models with different communication mechanisms, namely the π-calculi, and establishes basics results based on the asynchronous properties of these models. We also introduce the basic observational equivalences for these models, namely the notions of bisimulations.

Contents

1.1 The various π-calculi 18
1.1.1 The synchronous π-calculus: π_s 18
1.1.2 The various asynchronous π-calculi. 22
1.1.3 The π_{sc}-calculus 23
1.1.4 The π_{ic}-calculus 25
1.1.5 The π_a-calculus 26
1.2 Bisimulations and asynchronous communications 26
1.2.1 Motivations 26
1.2.2 Strong bisimilarity, weak bisimilarity 28
1.2.3 Asynchronous bisimilarity 29

1.1 The various π-calculi

Introduction

Several models for communication have been proposed in the literature. The kind of models known as Process Calculi consists of formalisms in which processes are specified syntactically by a grammar and semantically by a set of (structural) transition rules which define their operational behavior.

The first process calculi proposed in literature (CSP [BHR84, Hoa85], CCS [Mil80, Mil89], ACP [BK84]) were all based on synchronous communication primitives. This is because synchronous communication was considered somewhat more basic, while asynchronous communication was considered a derived concept that could be expressed using buffers (see, for instance, [Hoa85]). Some early proposals of calculi based purely on asynchronous communication were based on forcing the interaction between processes to be always mediated by buffers [BKT84, dBKP92].

At the beginning of the 90's, asynchronous communication became much more popular thanks to the diffusion of the Internet and the consequent increased interest for widely distributed systems. The elegant mechanism for asynchronous communication (the asynchronous send) proposed in the asynchronous π-calculus [HT91, Bou92] was very successful, probably because of its simple and basic nature, in line with the tradition of process calculi. Thus it rapidly became the standard approach to asynchronism in the community of process calculi, and it was adopted, for instance, also in Mobile Ambients [CG00]. A communication primitive (*tell*) similar to the asynchronous send was also proposed, independently, within the community of Concurrent Constraint Programming [SRP91a].

In the following, we present different variants of the π-calculus, including the synchronous and asynchronous π-calculus, as well as intermediate languages, which use synchronization primitives different from the ones of those two calculi.

1.1.1 The synchronous π-calculus: π_s

The synchronous calculus is the most expressive calculus of the π-calculus family. It was originally introduced by Robin Milner, Joachim Parrow and David Walker. The model is an extension of the Calculus of Communicating Systems [Mil80, Mil89] (CCS), adding the possibility of transmitting a message between two processes trough a communication channel in order to allow dynamic reconfiguration of the network and communication capabilities.

Indeed, in the π-calculus, messages and channels are the same objects, referred as *names*. It is then possible to send to a process a new channel name, allowing

this process to communicate trough this channel. As for the λ-calculus, where everything is a function, in the π-calculus everything is a name.

Definition 1.1.1

Let N be a countable infinite set. We call the elements of N the names. If x is a name, then $\overline{x}$ is the co-name of x.

A process in the π-calculus is built using these names and the following syntax:

Definition 1.1.2

A π-calculus process is an element generated by the following grammar:

$$P, Q, \ldots := 0 \mid \overline{x}z.P \mid x(y).P \mid P + Q \mid \nu x P \mid P \,|\, Q \mid !P$$

The labels for the transitions in the π_a-calculus are:

$$l, m, \ldots := xy \mid \overline{x}y \mid \overline{x}(y) \mid \tau$$

The meaning of the grammar is as follow:

- The 0 process is a process that cannot do anything
- $\overline{x}z.P$ is a process that sends the name z on the channel x, then proceeds by the execution of the process P. It is refered as a sending prefix or sending continuation.
- $x(y).P$ is a process that waits for the reception of a value on the channel x, and then follows with the execution of P, where all occurrences of y have been replaced by the received value. It is refered as a input prefix.
- $P + Q$ is a process that that can proceed either as P or as Q. As soon as the transition from one process is executed, the other is dismissed.
- $\nu x P$ is the process P where all occurrences of x are considered as locally bound. Any other occurrence of x which is not under the scope of this binder cannot interact with these local occurrences.
- $P \,|\, Q$ is a parallel composition of the processes P and Q. The transitions of $P \,|\, Q$ are those of either P or Q, or the result of any communication between them.
- $!P$ is a process that can spawn an unbounded number of parallel copies of P.

The transition labels denotes the various actions that a process can do:

- xy stands for the reception of the value y over the channel x
- $\overline{x}y$ stands for the sending of the value y over the channel x

- $\overline{x}(y)$ stands for the sending of the private value y over the channel x. This action allows a process to communicate a private name to another process, creating a private link between them.

- τ stands for the silent (internal) action

If we consider the process $x(y).P$ or $\nu y P$, the name y in both cases is a local name, bound to the scope of the process itself. If the name y appears in the context of another process, it has a different scope and cannot interact with the name y as referred by the first process. The set of such bound names of a process and the set of its free names are used to discriminate which names can be used to interact with other processes. They are defined as follows:

Definition 1.1.3 (*[SW01], Definition 1.1.2*)

In each of $\nu z P$ and $x(z).P$, the displayed occurence of z is binding with scope P. An occurence of a name in a process is bound if it is, or it lies into the scope of, a binding occurence of a name. An occurence of a name in a process is free if it is not bound. $fn(P)$ (resp. $bn(P)$) stands for the free names of P (resp. bound names of P).

Bound names are names that are binded internaly. Hence, we can apply α-renaming on bounded names. More generally, we define a notion of renaming for the processes of the π-calculus.

Definition 1.1.4 (*[SW01], Definition 1.1.5*)

Let P be a process of the π-calculus.

- *If the name w does not occur in P, then $P[w/z]$ is the process obtained by replacing each free occurence of z by w.*
- *A change of bound name is the replacement of a subterm $x(z).Q$ of P by $x(w).Q[w/z]$, or the replacement of a subterm $\nu z P$ by $\nu w Q[w/z]$, where in each case w does not occur in Q.*
- *Any process Q is an α-renaming of P if it can be obtained by a finite number of changes of bound name in P.*

We want to enforce some symmetries and obvious equivalences between processes. Hence, we define an equivalence relation between processes, to reflect, for instance, the fact that we want: $P \mid Q \equiv Q \mid P$.

Definition 1.1.5 (*structural congruence*)

The relation $\equiv$ is the smallest congruence over processes satisfying:

- *α-conversion on bound names*
- *The commutative monoid laws for parallel and sum composition with 0 as identity*
- $\nu x(P \mid Q) \equiv P \mid \nu x Q$ *when* $x \notin fn(P)$
- $\nu x 0 \equiv 0$
- $\nu x \nu x P \equiv \nu x P$
- $\nu x \nu y P \equiv \nu y \nu x P$
- $\nu x z(t).P \equiv z(t).\nu x P$ *when* $z \neq x$ *and* $t \neq x$

The structural congruence is convenient in order to define the operational semantics of the language, as explained below. The structural congruence represents the static equalities that we want to achieve between processes. This relation is used to match processes that are not syntactically equal, but for which the equality should be implicit, like when renaming bound names. Hence, it is very important that it is decidable since it is used often when computing the possible transitions of a process. In particular, the decidability of this relation is not known when adding the scope extrusion rules, such as $\nu x(P \mid Q) \equiv P \mid \nu x Q$ when $x \notin fn(P)$ and the rule $!P \equiv P \mid !P$. The decidability of the structural congruence when including replication is discussed in details in [EG99]. In particular, it is proved that, when adding also other rules such as $!!P \equiv !P$ or $!(P \mid Q) \equiv !P \mid !Q$, then the structural congruence is decidable. Since we do not need the structural replication rule, we avoid adding any of them in the relation.

The operational semantics of the π-calculus is defined in Table 1.1. It formalizes the meaning of the grammar, as explained above. It is a labeled transition system where labels are the transition labels given in Definition 1.1.

The rule (*open*) formalizes the sending of a private value. Hence, the ν binding is removed since the value has been sent outside of the scope of the process. However, when placed in parallel with another process, this value should not match another value. For instance if $P = \overline{y}t \mid \nu y(\overline{x}y.y(v))$, then we obviously do not want that: $P \xrightarrow{\overline{x}(y)} \overline{y}t \mid y(v)$ since the name y in $\overline{y}t$ should not be the same. Hence, the (*cong*) rule is mandatory. Using it, we should state that: $P \equiv \overline{y}t \mid \nu z(\overline{x}z.z(v))$ and then: $P \xrightarrow{\overline{x}z} \overline{y}t \mid z(v)$. The (*close*) rule is entailed by the rules (*sync*), (ν) and (*cong*), hence it may not always be present in some further operational semantics.

The semantics written in this document is the early semantics, in contrast to the late semantics. In the late semantics, the input process $P = x(y).Q$ only has a single transition $P \xrightarrow{x(y)} Q$ and the name substitution is done in the communication rule, i.e. the rule for the parallel operator. The late semantics is more convenient for implementations of the language since the input process has only one single transition, while in the early semantics, there are infinitely

$$(in)\ \frac{z \notin bn(P)}{x(y).P \xrightarrow{xz} P[z/y]} \qquad (out)\ \frac{}{\overline{x}y.P \xrightarrow{\overline{x}y} P}$$

$$(sync)\ \frac{P \xrightarrow{\overline{x}y} P',\ Q \xrightarrow{xy} Q'}{P \,|\, Q \xrightarrow{\tau} P' \,|\, Q'} \qquad (\nu)\ \frac{P \xrightarrow{\alpha} P',\ a \notin fn(\alpha)}{\nu a P \xrightarrow{\alpha} \nu a P'}$$

$$(open)\ \frac{P \xrightarrow{\overline{x}y} P'\ \ x \neq y}{\nu y P \xrightarrow{\overline{x}(y)} P'} \qquad (close)\ \frac{P \xrightarrow{\overline{x}(y)} P',\ Q \xrightarrow{xy} Q'}{P \,|\, Q \xrightarrow{\tau} \nu y(P' \,|\, Q')}$$

$$(bang)\ \frac{P \,|\, !P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'} \qquad (cong)\ \frac{P \equiv P',\ P' \xrightarrow{\alpha} Q',\ Q \equiv Q'}{P \xrightarrow{\alpha} Q}$$

$$(comp)\ \frac{P \xrightarrow{\alpha} P',\ bn(\alpha) \cap fn(Q) = \emptyset}{P \,|\, Q \xrightarrow{\alpha} P' \,|\, Q} \qquad (sum)\ \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}$$

Table 1.1: Operational semantics for the π-calculus

many of them. On the other hand, reasoning on the transitions generated by the late semantics can be more difficult, since one has to take into account the fact that the received value has not yet been substituted in the resulting process. Finally, the bisimilarities[1] associated to those two semantics are different, the bisimulation associated to the late semantics being more restrictive than the one entailed by the early semantics. This issue is discussed in Section 3.1.

When clear from the context, we may use short-hand notations for some processes, namely: $x(y) = x(y).0$, $\overline{x}y = \overline{x}y.0$, $x.P = x(v).P$, where $v \notin fn(P)$, $\overline{x}.P = \overline{x}v.P$ where $v \notin fn(P)$, and $\tau = \nu x(\overline{x} \,|\, x)$.

1.1.2 The various asynchronous π-calculi.

The π-calculus models communications using a synchronous rule. Indeed, when two processes communicate with each other, the two operations, sending and receiving, are done in a single step. The transition is then atomic and hence synchronous:

$$\overline{x}z.P \,|\, x(y).Q \longrightarrow P \,|\, Q[z/y]$$

As explained in the introduction, this does not apply quite easily to model most of the communications in real life, since for almost all of them, the sending and receiving operations do not happen at the same time. Furthermore, with the synchronous rule the sending process knows precisely when its message has

[1] defined in Section 1.2

been received. In [Pal97], the author shows that there is an expressivity gap, depending on the possibility to communicate synchronously or not.

The π_a-calculus was introduced, simultaneously in [Bou92] and [HT91]. The π_a-calculus restricts the grammar of the π-calculus in order to remove operations that implies some sort of synchronism. The operational semantics remains the same with some small modifications, which are mainly restrictions that reflects the new restricted grammar.

Similarly, the possibility to execute input, output or mixed choices like (resp.) $x(y).P + z(t).Q$, $\overline{x}y + \overline{x}y.Q$ or $\overline{x}y.P + z(t).Q$ can be argued to be a synchronous operation. Consequently, similarly restricted versions of the π-calculus have been proposed to reflect these considerations.

We will present here tree variants of these asynchronous languages, and prove simple properties about them. These properties will be usefull later to study the natural representation of the communications that each calculus model. Each of these properties are inherited from the less restrictive languages by the most restrictive ones.

1.1.3 The π_{sc}-calculus

This is a fragment of the synchronous π-calculus where the general choice is replaced by separate choice: choices are prefixed by a set of actions of the same type, either sending actions or reception actions.

The restriction is motivated by the fact that the possibility, for a process, to choose wether it will receive a new value or send a message is not easy to implement. This is for instance studied in [PH05b]. Intuitively, this relates to the possibility to communicate in full-duplex. As argued in the introduction, this possibility is not always granted, and it is even hardly the case. Hence, this language restricts the communication primitives so as to remove this possibility in the core language.

The syntax is the following:

$$P, Q, \ldots := \sum_{i \in I} x_i(y_i).P_i \;\Big|\; \sum_{i \in I} \overline{x}_i z_i.P_i \;\Big|\; \nu x P \;\Big|\; P \mid Q \;\Big|\; !P$$

Here I is a set of indexes. Note that we have omitted the process 0 since it can be represented as the empty summation.

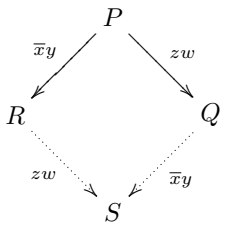
The definition of the transition semantics is the same as the one of the π-calculus (Table 1.1).

The crucial property here is a confluence that holds in the separate-choice π-calculus, as proved in [Pal03](Lemma 4.1). This property, which is present in

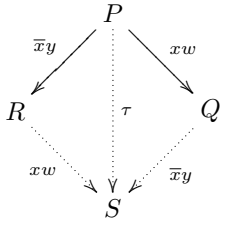
many asynchronous communication models, is the central result for proving the separation between the synchronous π-calculus and the asynchronous one. Indeed, the results is obtained by proving that the application of this confluence gives the possibility to find incorrect asynchronous executions for the problem of distributed symmetric consensus.

Lemma 1.1.1 (*Confluence, [Pal03], Lemma 4.1*)

Let $P \in \pi_a$. Assume that $P \xrightarrow{\overline{x}y} R$ and $P \xrightarrow{zw} Q$.
Then there exists $S \in \pi_a$ such that :



Furthermore, if $x = z$ then:



We can also prove the following extension of the above lemma, illustrated in Figure 1.1.

Lemma 1.1.2 (*Confluence with τ*)

Let $P \in \pi_a$. Assume that $P \xrightarrow{\tau} R$ and $P \xrightarrow{\overline{x}y} Q$. Then, either
1. *$P \xrightarrow{xz}$ for any z, or*
2. *there exists $S \in \pi_{sc}$ such that: $Q \xrightarrow{\tau} S$ and $R \xrightarrow{\overline{x}y} S$.*

Proof: We have to consider the possibility that the transition $P \xrightarrow{\tau} R$ is the result of a synchronization between $P_1 \xrightarrow{xy} Q_1$ and $P_2 \xrightarrow{\overline{x}y} Q_2$, where P_1 and P_2 are parallel subprocesses in P, and the latter transition is the one which induces $P \xrightarrow{\overline{x}y} Q$. If this is the case, then $P \xrightarrow{xz}$ for any z (note that x cannot be bound in P because $P \xrightarrow{\overline{x}y}$). On the other hand, if $P \xrightarrow{\tau} R$ does not involve the transition that induces $P \xrightarrow{\overline{x}y} Q$, then the proof is the same as for Lemma 1.1.1 (see [Pal03], Lemma 4.1). ■

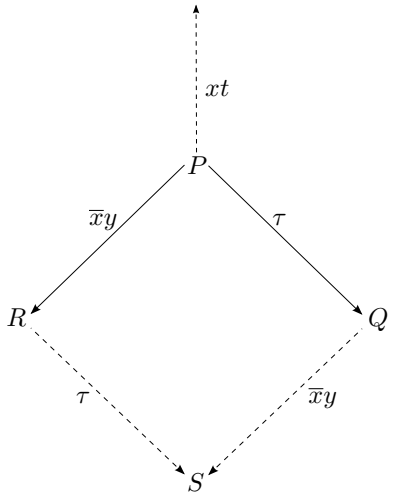


Figure 1.1: Confluence with τ

1.1.4 The π_{ic}-calculus

The π_{ic}-calculus is a restricted version of the π_{sc}-calculus where choices can only be made of input prefixed processes. This is a natural restriction that enforces the fact that the reception cannot be controlled. Not only the resulting processes are unable to choose between receiving or sending messages, but also they cannot decide which sending operation may happen.

Again, this reflects the fact that, if various messages are to be sent by a process, the reception cannot be controlled. Hence, a mutually exclusive choice among various possible send actions is not possible. Furthermore, sending continuation is not relevant either, since the process cannot control the reception.

$$P, Q, \ldots := \sum_{i \in I} x_i(y_i).P_i \,\Big|\, \overline{x}y \,\Big|\, \nu x P \,\Big|\, P \,|\, Q \,\Big|\, !P$$

Since we removed the sending prefix, send actions in this language can always be delayed. Hence, we can prove the following properties:

Lemma 1.1.3 (*[SW01], Lemma 5.3.1*)

If P is a process of the π_{ic}-calculus such that $P \xrightarrow{\overline{x}y} P'$, then $P \equiv \overline{x}y \,|\, P'$.

An important consequence is the following property:

Lemma 1.1.4 (*[SW01], Lemma 5.3.2*)

Let P be a process from the π_{ic}-calculus.
If $P \xrightarrow{\overline{x}y} \xrightarrow{\alpha}$, then $P \xrightarrow{\alpha} \xrightarrow{\overline{x}y}$

1.1.5 The π_a-calculus

The π_a-calculus is the most restrive of the various asynchronous calculi that we present. It follows the same logic as previous restrictions, but also removes the input-prefixed choice.

Definition 1.1.6

A π_a-calculus process is an element generated by the following grammar:

$$P, Q, \ldots := 0 \,\Big|\, \overline{x}z \,\Big|\, x(y).P \,\Big|\, \nu x P \,\Big|\, P \mid Q \,\Big|\, !P$$

This fragment of the π-calculus is the most studied of the various asynchronous calculi. Several encodings of the previous asynchronous calculi have been proposed. In particular, a fairly good encoding of the input prefixed choice is proposed in [NP00], and two variants of the output prefix in [Bou92, HT91].

1.2 Bisimulations and asynchronous communications

1.2.1 Motivations

Now that we have defined the processes from the π-calculus and their possible transitions, even though we have a structural equivalence $\equiv$, there still are processes that we want to identify, but which are not structurally equivalent. More generally, given two arbitrary processes, we would like to know wether they can be considered equivalent or not. In particular, one would like to abstract this equivalence from the internal states of the process, and possibly also internal steps. We then need an observational equivalence, based on the *observations* we can make on the process.

A naive approach could use the notion of morphisms on the graphs generated by the labeled transition system. However, processes would then be equivalent when there is an homeomorphism between their associated graphs. But this notion is too strong, since an homeomorphism means that the graphs are algebraically equivalent, which is not what we were looking for. See Figure 1.2 for an example.

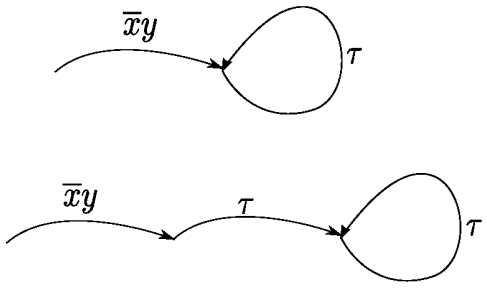


Figure 1.2: Equivalent graphs that are not homeomorphic.

Another approach would be to use (partial) traces. Partial traces are sequences of transition labels as occurring during a partial execution of a process. Two processes are then equivalent when their set of partial traces are the same.

This equivalence is known to be a congruence, or compositional (see Definition 3.2.2). However, when studying partial traces of a process, upon observing for instance the set of partial traces: $\{x; x.\overline{t}; x.\overline{f}\}$, if we observe the partial trace x, we cannot determinate wether $\overline{t}$ can still be observed. See Figure 1.3 for an illustration of this. Similarly, it cannot be known wether a process after a partial trace reached a final state or is simply staled.

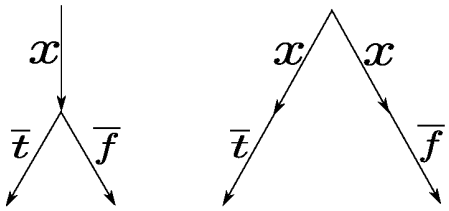


Figure 1.3: Partial trace equivalent processes

In a security context, however, this is not satisfactory. Indeed, when dealing with security concerns, it is very important to know wether a process *can* proceed with a given observable.

Hence, one could use complete traces. In this case, only trace corresponding to maximal computations (possible infinite) are considered. This gives the ability to distinguish final states to partially executed states. But the semantics given by complete traces is not compositional, which makes it difficult to use for modular analysis.

A refinement of the complete trace semantics can be obtained using the testing semantics. Under this semantics, a process is tested by adding a parallel process containing a special symbol. The process passes the test when it can show an execution where the special symbol is observed. Two processes are then

equivalent when they pass the same tests.

This notion is much more satisfactory. In particular it is compositional and fully abstract – that is to say that it contains exactly enough information compared to the observations you can make: if two processes are not testing-equivalent, then there exists a context for which a difference can be observed during the execution.

However, the testing semantics is quite hard to use in a theoretical context, since it requires the need to prove equivalence for every possible test, which is not theoretically easy.

On the other hand, the notions of *bisimulation* and *bisimilarity* are much more interesting for the theoretical proofs. Their co-inductive definition make it much easier to use. Furthermore, as we will see later, they enjoy, in the relevant languages for this study, all the good properties that testing semantics has. We present now the various notions of bisimulations used in the literature.

1.2.2 Strong bisimilarity, weak bisimilarity

The reasons mentioned above show the need for an equivalence in terms of behavior for the processes in the π-calculus. This is achieved with a co-inductive definition of an equivalence relation called the *bisimulation*.

To check wether two processes are equivalent, you may look at any arbitrary action that one can perform, and see if the other one can perform the same action, and become a process that remains equivalent to the resulting process for the first one.

In the following, bisimulation will denote one binary relation with the required properties, while bisimilarity will denote greatest bisimulation relation.

Formally, this gives:

Definition 1.2.1 (*strong bisimilarity*)

Let P and Q be two π-calculus processes. P and Q are bisimilar if and only if there exists a binary relation $\mathcal{R}$ such that $P\mathcal{R}Q$ and:

- *if $P \xrightarrow{\alpha} P'$ and $bn(\alpha) \notin fn(Q)$ then $Q \xrightarrow{\alpha} Q'$ and $P'\mathcal{R}Q'$*
- *if $Q \xrightarrow{\alpha} Q'$ and $bn(\alpha) \notin fn(P)$ then $P \xrightarrow{\alpha} P'$ and $P'\mathcal{R}Q'$*

$\sim$ is the greatest binary relation satisfying these two properties and is referred as the bisimilarity relation.

A particular relation $\mathcal{R}$ following the definition for a bisimulation relation will be referred as a *bisimulation*. The greatest (or union) of all bisimulations will be referred as the *bisimilarity*. Bisimilations and bisimilarities can also be given one or several characterizing adjectives, *late*, *early*, *weak*, *asynchronous*.... When stated without adjective, the underlying bisimulation notion will be the *early* one, which is stated above.

Some examples of bisimilar processes can be:

Example : The following processes are bisimilar

- $\nu x\overline{x}y \sim 0$
- $\overline{x}y.0 \mid t(z).0 \sim \overline{x}y.t(z).0 + t(z).\overline{x}y \ (t \neq x)$

The bisimulation expresses the fact that two processes behave in a similar way. However, when trying to mimic one process, it is sometimes needed to have internal operations in the imitating process, to reflect some internal administrative actions. These actions would be labeled τ. We may not want to discriminate two processes on one of those internal steps, since they should not be distinguishable from an external point of view. Hence, the relaxed notion of weak bisimulation.

In the following, we use the standard notation for weak transitions: $P \stackrel{\alpha}{\Longrightarrow} Q$ stands for $P \stackrel{\tau}{\longrightarrow}^* \stackrel{\alpha}{\longrightarrow} \stackrel{\tau}{\longrightarrow}^* Q$. If $\alpha = \tau$, then the transition may also denote a null transition, which means that the process does nothing and remains the same.

Definition 1.2.2 (*weak bisimilarity*)

Let P and Q be two π-calculus processes. P and Q are weakly bisimilar if and only if there exists a binary relation $\mathcal{R}$ such that $P\mathcal{R}Q$ and:

- *if $P \stackrel{\alpha}{\rightarrow} P'$ and $bn(\alpha) \notin fn(Q)$ then $Q \stackrel{\alpha}{\Rightarrow} Q'$ and $P'\mathcal{R}Q'$*
- *if $Q \stackrel{\alpha}{\rightarrow} Q'$ and $bn(\alpha) \notin fn(P)$ then $P \stackrel{\alpha}{\Rightarrow} P'$ and $P'\mathcal{R}Q'$*

$\approx$ is the greatest binary relation satisfying these two properties.

1.2.3 Asynchronous bisimilarity

When establishing an observational equivalence for processes communicating asynchronously, the emission of a message can always be delayed, hence cannot be tracked down precisely, or shouldn't since the sending process is unable to know when reception is done. In this case, one may want to avoid some distinctions based on this property. A particular case is when a process receives a message, and then send the message again, acting as a proxy for this message. In an asynchronous context, this should not be noticed by an equivalence relation.

For instance, this is the case when studying processes behavior in the asynchronous π-calculus where Lemma 1.1.4 holds, i.e. where there is no output prefix.

If a process P can send the message $\overline{x}y$, since is x necessarily free in P, then, $P \equiv \overline{x}y \mid P'$.

If $Q \stackrel{xy}{\rightarrow} Q'$ and $Q' \equiv \overline{x}y \mid Q''$, which means that Q receives and sends back the message, then:

$$P \mid Q \equiv P' \mid \overline{x}y \mid Q \xrightarrow{\tau} \equiv P' \mid \overline{x}y \mid Q''$$

Hence, Q behaves like a buffer for the message $\overline{x}y$.

If we want to prove that R is bisimilar to Q, then if $R \xrightarrow{\tau} R''$, we have that: $P \mid R = P' \mid \overline{x}y \mid R \xrightarrow{\tau} P' \mid \overline{x}y \mid R''$. Hence the reception action of Q can also be matched by a τ action of R, provided that $R'' \sim Q''$.

This is achieved with the notion of asynchronous bisimulation, which can be found in [ACS98] and [HT91][2].

Definition 1.2.3 (*strong asynchronous bisimilarity*)

A symmetric relation $\mathcal{R}$ is a strong asynchronous bisimulation iff whenever $P \mathcal{R} Q$, then the following holds:

- *If $P \xrightarrow{\alpha} P'$, $bn(\alpha) \notin fn(Q)$, and α is not an input action, then: $Q \xrightarrow{\alpha} Q'$ with: $P' \mathcal{R} Q'$*
- *If $P \xrightarrow{xy} P'$ then*
  - *either $Q \xrightarrow{xy} Q'$ with $P' \mathcal{R} Q'$,*
  - *or $Q \xrightarrow{\tau} Q'$ and $P' \mathcal{R} (Q' \mid \overline{x}y)$.*

We say that P and Q are strongly asynchronously bisimilar, written $P \sim_a Q$, iff there exists $\mathcal{R}$ such that: $P \mathcal{R} Q$.

The asynchronous bisimulation takes into account the considerations explained above. It states that, if P and Q are asynchronously bisimilar, and P receives the message xy, but is then equivalent to $P' \mid \overline{x}y$, then the receiving action corresponding to the message xy can be match by a τ action. This situation is illustrated in terms of transition graph in Figure 1.4.

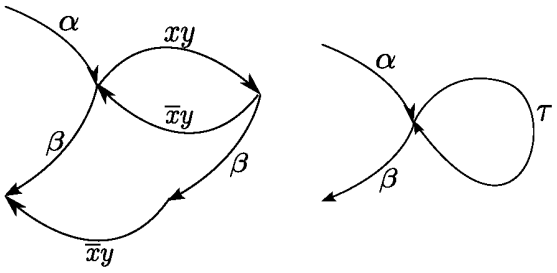


Figure 1.4: Two asynchronously bisimilar graphs.

Several processes which are not equivalent for the bisimilarity are equivalent for asynchronous bisimilarity. For instance $a(b).(\overline{a}b \mid a(b).P) \sim_a a(b).P$.

As we can see, all buffer-like message processing is lumped into 0, which seems a natural requirement for the asynchronous version of the language. But, as for

[2]Some not-so-trivial results, like the fact that asynchronous bisimulations are equivalence relations, as expected, are omitted here and may be found in these papers.

the strong bisimulation, the strong asynchronous bisimulation can be argued to be too strong. In particular, under this relation, $!a(b).\overline{a}b \not\sim_a 0$, since each reception from $!a(b).\overline{a}b$ has to be matched by an action from 0.

This leads to the notion of weak asynchronous bisimulation, where τ transitions are considered internal steps and removed from the observational equivalence.

Definition 1.2.4 (*Weak asynchronous bisimilarity*)

A symmetric relation $\mathcal{R}$ is a weak asynchronous bisimulation iff whenever $P\mathcal{R}Q$, then the following holds:

- *If $P \xrightarrow{\alpha} P'$, $bn(\alpha) \notin fn(Q)$, and α is not an input action, then: $Q \stackrel{\alpha}{\Longrightarrow} Q'$ with: $P' \mathcal{R} Q'$*
- *If $P \xrightarrow{xy} P'$ then*
  - *either $Q \stackrel{xy}{\Longrightarrow} Q'$ with $P' \mathcal{R} Q'$,*
  - *or $Q \stackrel{\tau}{\Rightarrow} Q'$ and $P' \mathcal{R} (Q' \mid \overline{x}y)$.*

We say that P and Q are weakly asynchronously bisimilar, written $P \approx_a Q$, iff there exists $\mathcal{R}$ such that: $P \mathcal{R} Q$.

Under this relation, we have for instance: $!a(b).\overline{a}b \approx_a 0$. The relation in terms of transition graph is illustrated by Figure 1.5.

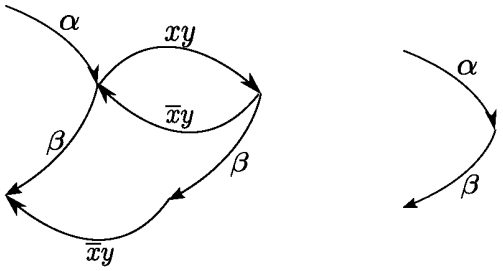


Figure 1.5: Weakly asynchronously bisimilar graphs.

Weak asynchronous bisimulation is weaker than weak bisimulation, and it is weaker than strong asynchronous bisimulation, and is illustrated in Figure 1.6. In this figure $A \rightarrow B$ means that the relation B entails the relation A.

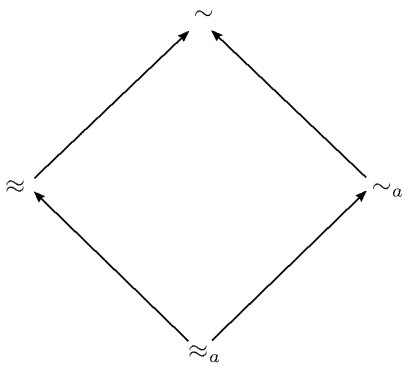


Figure 1.6: Relations between the various bisimilations.

Chapter 2

Communicating trough buffers

After the introduction of the various communication models in the π-calculus family, we present a new result on the correspondence between asynchronous communications as sending primitives without continuation and asynchronous communications as communication happening through buffers. We define a new language using the sending primitive of the π-calculus, but were communication occurs through buffers. We then prove the correspondence using a custom encoding from and to the π_a-calculus and the weak asynchronous bisimulation. Further result show that there are no such encoding and correspondence for the weak asynchronous bisimulation when communication occurs through Queues (aka FIFO) or Stacks (aka LIFO).

These results have been published in [BPV08].

Contents

2.1 Buffers, stacks, queues and bags 34
2.2 The $\pi_{\mathfrak{B}}$-calculus 36
2.3 Encodings from and to the π_a-calculus and $\pi_{\mathfrak{B}}$-calculus 37
2.3.1 From π_a to $\pi_{\mathfrak{B}}$ 38
2.3.2 From $\pi_{\mathfrak{B}}$ to π_a 39
2.4 Impossibility result for the other types of buffer . 42
2.4.1 Impossibility of encoding queues and stacks 43
2.4.2 Impossibility of encoding stacks in the π-calculus without mixed-choice operator 43

Introduction

With the results of this chapter, we aim at explaining what represents the most natural communication mechanism for the asynchronous languages in terms of buffered-communications. It turns out that the π_a-calculus behaves in a very similar way as a synchronous π-calculus where communications happen through unordered buffers. Hence, it can be argued that representing asynchronous communications by the processes' primitives, as for the π_a-calculus, or by the communication mechanisms results in the same expressiveness, hence the same model.

Similarly, we observe that the natural representation of the communication mechanisms happening with sending continuation is the queue, or First In, First Out (FIFO), policy, where messages are sent from the buffer in the same order as they are received. Also, the natural representation of the communication mechanisms happening with sending continuation and mixed choice is the stack, or Least In, Least Out (LIFO), policy, where the latest received message is sent last from the buffer.

More precisely, in the case of sending continuation (resp. sending continuation and mixed choice), we state that the communication mechanisms need to be able to behave like a queue (resp. stack) in order to naturally mimic the processes from the corresponding π-calculi, and that this is not possible with processes of the π_a-calculus (resp. π_{ic}-calculus) up-to weak asynchronous bisimulation.

2.1 Buffers, stacks, queues and bags

A buffer is basically a data structure that accepts messages and resends them later. We consider different types of buffers, depending on the policy used for outputting a previously received message. We focus on the following policies, that can be considered the most common:

- Bag, or unordered policy: any message previously received (and not yet sent) can be sent next.
- Queue, or FIFO policy: only the oldest message received (and not yet sent) can be sent next.
- Stack, or LIFO policy: only the last message received (and not yet sent) can be sent next.

Let us now formally define these three types of buffer. We need to keep the information about the order of reception to decide which message can be sent next. This will be achieved using a common core definition for all kinds of buffers.

We will use $M \in \mathcal{M}$ to denote a message that the buffers can accept.

Definition 2.1.1 (*Buffer*)

A buffer is a finite sequence of messages:
$B = M_1 * ... * M_k$, $k \geq 0$, $M_i \in \mathcal{M}$ *(B is the empty sequence if* $k = 0$*).*

$*$ is a wild card symbol for the three types of buffers. Then, we will use the notation $M_1 \diamond ... \diamond M_k$ for a bag, $M_1 \triangleleft ... \triangleleft M_k$ for a queue, $M_1 \triangleright ... \triangleright M_k$ for a stack.

A reception on a buffer is the same for all kinds of policies:

Definition 2.1.2 (*Reception on a buffer*)

Let $B = M_1 * ... * M_k$*. We write*
$B \xrightarrow{M} B'$ *to represent the fact that B receives the message M, becoming* $B' = M * B = M * M_1 * ... * M_k$.

The emission of a message is different for the three types of buffers:

Definition 2.1.3 (*Sending from a buffer*)

Let $B = M_1 * ... * M_k$*. We write*
$B \xrightarrow{\overline{M}} B'$ *to represent the fact that B sends the message M, becoming* B'*, where:*

- *If* $* = \diamond$ *(bag case) then* $M = M_i$ *for some* $i \in \{1, ..., k\}$ *and* $B' = M_1 \diamond ... \diamond M_{i-1} \diamond M_{i+1} \diamond ... \diamond M_k$.
- *If* $* = \triangleleft$ *(queue case) then* $M = M_k$ *and* $B' = M_1 \triangleleft ... \triangleleft M_{k-1}$.
- *If* $* = \triangleright$ *(stack case) then* $M = M_1$ *and* $B' = M_2 \triangleright ... \triangleright M_k$.

Remark 2.1.1

If B is a buffer such that $B \xrightarrow{\overline{M_1}}$ *and* $B \xrightarrow{\overline{M_2}}$ *with* $M_1 \neq M_2$ *then B must be a bag, i.e. B cannot be a stack or a queue.*

Finally, we introduce here the notion of buffer's content and sendable items.

Definition 2.1.4 (*Buffer's content*)

A buffer's content is the multiset of messages that the buffer has received and has not yet sent:

$$C(M_1 * ... * M_k) = \{\!\{M_1, ..., M_k\}\!\}$$

Definition 2.1.5 (*Buffer's sendable items*)

A buffer's sendable items is the multiset of messages that can be sent immediately:

$$S(M_1 \diamond M_2 \diamond \cdots \diamond M_k) = \{\!\{M_1, M_2, \ldots, M_k\}\!\}$$
$$S(M_1 \triangleleft M_2 \triangleleft \cdots \triangleleft M_k) = \{\!\{M_k\}\!\}$$
$$S(M_1 \triangleright M_2 \triangleright \cdots \triangleright M_k) = \{\!\{M_1\}\!\}$$

Note that $S(B)$ is empty iff $C(B)$ is empty. Furthermore, if B is a bag, then $C(B) = S(B)$.

2.2 The $\pi_{\mathfrak{B}}$-calculus

We define a calculus for asynchronous communications obtained by enriching the synchronous π-calculus with bags, and forcing the communications to take place only between (standard) processes and bags.

We decree that the bag's messages are names. Each bag is able to send and receive on a single channel only, and we write B_x for a bag on the channel x. We use $\{\!\{\}\!\}_x$ to denote an empty bag on channel x, and $\{\!\{y\}\!\}_x$ for the bag on channel x, containing a single message, y.

Definition 2.2.1

The $\pi_{\mathfrak{B}}$-calculus is the set of processes defined by the grammar:

$$P, Q ::= \textstyle\sum_{i\in I} \alpha_i.P_i \;\Big|\; P \,|\, Q \;\Big|\; \nu x\, P \;\Big|\; !P \;\Big|\; B_x$$

where B_x is a bag, I is a finite indexing set and each α_i can be of the form $x(y)$ or $\overline{x}z$. If $|I| = 0$ the sum can be written as 0 and if $|I| = 1$ the symbol "$\sum_{i\in I}$" can be omitted.

Definition 2.2.2 (*structural congruence*)

The relation $\equiv$ is the smallest congruence over processes satisfying:

- *α-conversion on bound names*
- *The commutative monoid laws for parallel and sum composition with 0 as identity*
- *$\nu x(P \,|\, Q) \equiv P \,|\, \nu x Q$ when $x \notin fn(P)$*
- *$\nu x P \equiv 0$ when $fn(P) \subset \{x\}$*
- *$\nu x \nu x P \equiv \nu x P$*
- *$\nu x \nu y P \equiv \nu y \nu x P$*
- *$P \equiv P \,|\, \{\!\{\}\!\}_x$ for all possible x*

The early transition semantics is obtained by redefining the rules *in* and *out* in Table 1.1 and by adding the rules in_{bag} and out_{bag} for bag communication as defined in Table 2.1. Note that they are basically the rules for the (synchronous) π-calculus except that communication can take place only between (standard) processes and bags. In fact, the rule *out* guarantees that a process can only output to a bag. Furthermore the only rule that generates an output transition is out_{bag}, hence a process can only input, via *sync* and *close*, from a bag. We also use the symbol $\xrightarrow{\alpha}_{\mathfrak{B}}$ to denote a transition using these operational rules.

The structural equivalence $\equiv$ consists of the standard rules of Definition 1.1.5, plus $P \equiv P \,|\, \{\!\{\}\!\}_x$. This last rule allows any process to have access to a buffer

even if the process itself is blocked by a binder. A typical example is $P = \nu x(\overline{x}y.x(z).Q)$, which can not execute any action without this rule. Thanks to the rule, we have:

$$P \equiv \nu x(\overline{x}y.x(z).Q \mid \{\!\{\}\!\}_x) \rightarrow_{\mathfrak{B}} \nu x(x(z).Q \mid \{\!\{y\}\!\}_x) \rightarrow_{\mathfrak{B}} \nu x(Q[y/z] \mid \{\!\{\}\!\}_x)$$

Note that we can also restrict the application of $P \equiv P \mid \{\!\{\}\!\}_x$ to the case in which P is a pure process (not containing a bag), i.e. a term of the asynchronous π-calculus). We do not impose this constraint here because it is not necessary, and also because we believe that allowing multiple bags on the same channel name and for the same process is a more natural representation of the concept of channel in distributed systems. Later, when dealing with stacks and queues, we will have to adopt this restriction in order to be consistent with the nature of stacks and queues.

A consequence of the rule $P \equiv P \mid \{\!\{\}\!\}_x$ is that every process P is always *input-enabled*. This property is in line with other standard models of asynchronous communication, for example the Input/output automata (see, for instance, [Lyn96]), the input-buffered agents of Selinger [Sel97] and the Honda-Tokoro original version of the asynchronous π-calculus [HT91].

The basic input and output transitions for bags given by in_{bag} and out_{bag} are defined in terms of receive and send transitions on buffers in Definition 2.1.2 and 2.1.3. The following remark follows trivially from the rules in Table 2.1.

Remark 2.2.1

Let B_x be a bag process. Then $B_x \xrightarrow{y} B'_x$ iff $B_x \xrightarrow{xy}_{\mathfrak{B}} B'_x$. Similarly, $B_x \xrightarrow{\overline{y}} B'_x$ iff $B_x \xrightarrow{\overline{x}y}_{\mathfrak{B}} B'_x$.

The notions of free names and bound names for ordinary processes are defined as usual. For bags, we define them as follows. Recall that C gives the content of a buffer (see Definition 2.1.4).

Definition 2.2.3 (*Bag's free and bound names*)

Let B_x be a bag with content $C(B_x) = \{\!\{y_1, \ldots, y_k\}\!\}$. The free variables fn and the bound variables bn of B_x are defined as $fn(B_x) = \{x, y_1, \ldots, y_k\}$ and $bn(B_x) = \emptyset$.

2.3 Encodings from and to the π_a-calculus and $\pi_{\mathfrak{B}}$-calculus

In this section, we study the relation between the π_a-calculus and the $\pi_{\mathfrak{B}}$-calculus.

$$(in_{bag})\ \frac{B_x \xrightarrow{y} B'_x}{B_x \xrightarrow{xy}_{\mathfrak{B}} B'_x} \qquad (out_{bag})\ \frac{B_x \xrightarrow{\overline{y}} B'_x}{B_x \xrightarrow{\overline{x}y}_{\mathfrak{B}} B'_x}$$

$$(in)\ \frac{\alpha_j = xy}{\sum_{i\in I} \alpha_i.P_i \xrightarrow{xz}_{\mathfrak{B}} P_j[z/y]} \qquad (out)\ \frac{B_x \xrightarrow{xy}_{\mathfrak{B}} B'_x}{(\overline{x}y.P + \sum_{i\in I} \alpha_i.P_i) \mid B_x \xrightarrow{\tau}_{\mathfrak{B}} P \mid B'_x}$$

$$(sync)\ \frac{P \xrightarrow{\overline{x}y}_{\mathfrak{B}} P',\ \ Q \xrightarrow{xy}_{\mathfrak{B}} Q'}{P \mid Q \xrightarrow{\tau}_{\mathfrak{B}} P' \mid Q'} \qquad (\nu)\ \frac{P \xrightarrow{\alpha}_{\mathfrak{B}} P',\ a \notin fn(\alpha)}{\nu a P \xrightarrow{\alpha}_{\mathfrak{B}} \nu a P'}$$

$$(open)\ \frac{P \xrightarrow{\overline{x}y}_{\mathfrak{B}} P',\ \ x \neq y}{\nu y P \xrightarrow{\overline{x}(y)}_{\mathfrak{B}} P'} \qquad (close)\ \frac{P \xrightarrow{\overline{x}(y)}_{\mathfrak{B}} P',\ \ Q \xrightarrow{xy}_{\mathfrak{B}} Q',\ \ y \notin fn(Q)}{P \mid Q \xrightarrow{\tau}_{\mathfrak{B}} \nu y(P' \mid Q')}$$

$$(bang)\ \frac{P \mid !P \xrightarrow{\alpha}_{\mathfrak{B}} P'}{!P \xrightarrow{\alpha}_{\mathfrak{B}} P'} \qquad (cong)\ \frac{P \equiv P'\ P' \xrightarrow{\alpha}_{\mathfrak{B}} Q'\ Q' \equiv Q}{P \xrightarrow{\alpha}_{\mathfrak{B}} Q}$$

$$(comp)\ \frac{P \xrightarrow{\alpha}_{\mathfrak{B}} P',\ bn(\alpha) \cap fn(Q) = \emptyset}{P \mid Q \xrightarrow{\alpha}_{\mathfrak{B}} P' \mid Q}$$

Table 2.1: Transition rules for the π-calculus with bags

We will use the two notions of asynchronous bisimulation introduced in the previous chapter to describe the properties of the encodings from π_a to $\pi_{\mathfrak{B}}$ and from $\pi_{\mathfrak{B}}$ to π_a, respectively. The notion of strong asynchronous bisimulation is almost the same, but not completely, as the one of [ACS98]. The difference is that, in [ACS98], when P performs an input action, Q can either perform a corresponding input action *or a τ step*. The reason for introducing the change is essentially to get the correspondence stated in Theorem 2.3.1. We could have used weak asynchronous bisimulation instead, but we preferred to show how strong the correspondence is. As for the notion of weak asynchronous bisimulation, this is essentially the same as the one introduced by [HT91] (called *asynchronous bisimulation* in that paper). The formulation is different, since the labeled transition system of [HT91] is different from ours, however it is easy to show that the (weak) bisimulations induced by their system, as relations on process terms, coincide with our weak asynchronous bisimulations.

2.3.1 From π_a to $\pi_{\mathfrak{B}}$

We observe that there is a rather natural interpretation of the π_a-calculus into the $\pi_{\mathfrak{B}}$-calculus, formalized by an encoding defined as follows:

Definition 2.3.1

Let $[\![\ ]\!] : \pi_a \hookrightarrow \pi_{\mathfrak{B}}$ *be defined as:*

- $[\![0]\!]_a = 0$
- $[\![\overline{x}y]\!]_a = \{\!\{y\}\!\}_x$.
- $[\![x(y).P]\!]_a = x(y).[\![P]\!]_a$
- $[\![\nu xP]\!]_a = \nu x[\![P]\!]_a$
- $[\![P \mid Q]\!]_a = [\![P]\!]_a \mid [\![Q]\!]_a$
- $[\![!P]\!]_a = !\![\![P]\!]_a$

It is easy to see that there is an exact match between the transitions of P and the ones of $[\![P]\!]_a$, except that $\{\!\{y\}\!\}_x$ can perform input actions on x that the original process $\overline{x}y$ cannot do. This is exactly the kind of situation treated by the additional case in the definition of asynchronous bisimilarity (additional w.r.t. the classical definition of bisimilarity). Hence we have the following result:

Theorem 2.3.1

Let $[\![\ ]\!]_a : \pi_a \hookrightarrow \pi_{\mathfrak{B}}$ *be the encoding in Definition 2.3.1. For every* $P \in \pi_a$, $P \approx_a [\![P]\!]_a$.

Proof: We prove the bisimulation using the following binary (symmetric[1]) relation:

$\mathcal{R} = \bigcup_{P \in \pi_a} \left(\{(P, [\![P]\!]_a\}\right) \cup \bigcup_{\{\!\{y_1,\ldots,y_n\}\!\}_x} \{(\{\!\{y_1,\ldots,y_n\}\!\}_x, \overline{x}y_1 \mid \ldots \mid \overline{x}y_n)\}$

As we can see from the encoding, if P is not an output, then if $P \xrightarrow{\alpha} P'$ then also $[\![P]\!]_a \xrightarrow{\alpha}_{\mathfrak{B}} P''$, and $P' \mathcal{R} P''$ and vice-versa.

Also, if $\overline{x}y_1 \mid \ldots \mid \overline{x}y_n \xrightarrow{\overline{x}y_i} P$ then also $\{\!\{y_1,\ldots,y_n\}\!\}_x \xrightarrow{\overline{x}y_i}_{\mathfrak{B}} B$ with $P \mathcal{R} B$, and vice-versa.

The only remaining case is the reception on a buffer, for which, if:

$\{\!\{y_1,\ldots,y_n\}\!\}_x \xrightarrow{xz}_{\mathfrak{B}} \{\!\{z, y_1,\ldots,y_n\}\!\}_x$

then:

$\{\!\{z, y_1,\ldots,y_n\}\!\}_x \mathcal{R}\ \overline{x}z \mid (\overline{x}y_1 \mid \ldots \mid \overline{x}y_n)$.

Such that the weak asynchronous bisimulation still holds, using the asynchronous case of the definition. ■

The encoding from $\pi_{\mathfrak{B}}$ into π_a is more complicated, but still we can give a rather faithful translation.

2.3.2 From $\pi_{\mathfrak{B}}$ to π_a

Our encoding of the $\pi_{\mathfrak{B}}$-calculus into the π_a-calculus is given below.

[1] For readibility, we only specify one half of the relation in this proof.

Definition 2.3.2

The encoding $[\![\ ]\!]_{\mathfrak{B}} : \pi_{\mathfrak{B}} \hookrightarrow \pi_a$ *is defined as follows:*

$$\begin{aligned}
[\![\textstyle\sum_{i\in I} \alpha_i.P_i]\!]_{\mathfrak{B}} &= \nu(l,t,f)\ (\bar{l}t \,|\, \Pi_{i\in I}[\![\alpha_i.P_i]\!]_{\mathfrak{B},l}) \\
[\![P \,|\, Q]\!]_{\mathfrak{B}} &= [\![P]\!]_{\mathfrak{B}} \,|\, [\![Q]\!]_{\mathfrak{B}} \\
[\![\nu v\ P]\!]_{\mathfrak{B}} &= \nu v\ [\![P]\!]_{\mathfrak{B}} \\
[\![!P]\!]_{\mathfrak{B}} &= !\,[\![P]\!]_{\mathfrak{B}} \\
[\![B_x]\!]_{\mathfrak{B}} &= \Pi_{y_i\in S(B_x)} \overline{x} y_i
\end{aligned}$$

where $[\![\ ]\!]_{\mathfrak{B},l}$ *is given by*

$$\begin{aligned}
[\![x(y).P]\!]_{\mathfrak{B},l} &= x(y).l(\lambda).\Big[(\mathit{if}\ \ \lambda = t\ \ \mathit{then}\ \ [\![P]\!]_{\mathfrak{B},l,x(y)}\ \mathit{else}\ \overline{x}y) \,|\, \bar{l}f\Big] \\
[\![P]\!]_{\mathfrak{B},l,x(y)} &= \nu l' \left[\overline{l'}() \,|\, l'().[\![P]\!]_{\mathfrak{B}} \,|\, l'().\Big(\overline{x}y \,|\, [\![x(y).P]\!]_{\mathfrak{B},l} \,|\, \bar{l}t\Big)\right] \\
[\![\overline{x}y.P]\!]_{\mathfrak{B},l} &= l(\lambda).\Big[(\mathit{if}\ \ \lambda = t\ \ \mathit{then}\ \ \overline{x}y \,|\, [\![P]\!]_{\mathfrak{B}}) \,|\, \bar{l}f\Big]
\end{aligned}$$

In this definition, we use a if-then-else construct in the form if $\lambda = t$ *then* P *else* Q *which is syntactic sugar for* $\overline{\lambda} \,|\, t.P \,|\, f.Q$*. This is correct within the scope of our definition because* λ *can only be* t *or* f*, and* λ*,* t *and* f *are private.*

This encoding of the mixed choice is similar to the first encoding of input guarded choice defined in [Nes00]. The $[\![P]\!]_{\mathfrak{B},l,x(y)}$ is important to establish the bisimilarity result. It consists of a non deterministic choice between going back to initial state, or following with $[\![P]\!]_{\mathfrak{B}}$.

The soundness of the encoding depends crucially on the fact that in the $\pi_{\mathfrak{B}}$-calculus the output of a standard process is non-blocking.

Note that this encoding is not termination preserving. As in [Nes00], this problem could be addressed by removing the backtracking possibility and using a coarser semantic (coupled bisimilarity). Here, we consider the stronger notion of weak asynchronous bisimilarity recalled above.

Theorem 2.3.2

Let $[\![\ ]\!]_{\mathfrak{B}} : \pi_{\mathfrak{B}} \hookrightarrow \pi_a$ *be the encoding in Definition 2.3.2. Then, for every* $P \in \pi_{\mathfrak{B}}$*,* $P \approx_a [\![P]\!]_{\mathfrak{B}}$*.*

Proof: We give the proof only for the non-trivial cases of the encoding, which are:

1. $[\![B_x]\!]_{\mathfrak{B}}$
2. $[\![\sum_{i\in I} \alpha_i.P_i]\!]_{\mathfrak{B}}$

We will show that the above encodings are weakly asynchronous bisimilar to their source processes. For (1), the statement follows from:

- $B_x \xrightarrow{xy}_{\mathfrak{B}} B'_x \Longrightarrow [\![B'_x]\!]_{\mathfrak{B}} = [\![B_x]\!]_{\mathfrak{B}} \,|\, \overline{x}y$
- $B_x \xrightarrow{\overline{x}y}_{\mathfrak{B}} B'_x \Longleftrightarrow [\![B_x]\!]_{\mathfrak{B}} \xrightarrow{\overline{x}y} [\![B'_x]\!]_{\mathfrak{B}}$

Let us now consider the case (2).

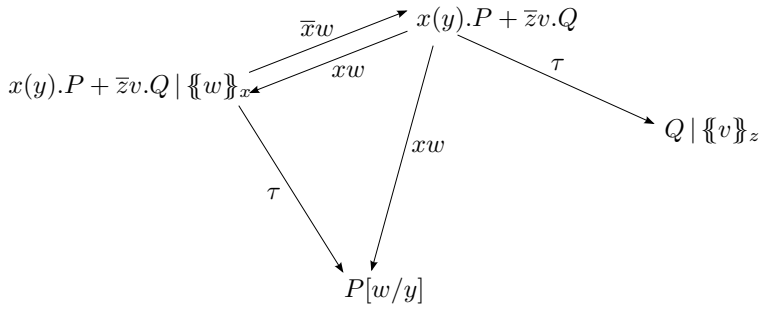


Figure 2.1: Transitions of a $\pi_{\mathfrak{B}}$ sum.

For the sake of simplicity, we outline the proof for a choice construct with only one input-guarded and one output-guarded branches, the proof for a choice with more than two branches can be easily generalized from this case.

There are three possible transitions from this choice[2]:

1. $x(y).P + \overline{z}v.Q \xrightarrow{\tau}_{\mathfrak{B}} Q \mid \{\!\{v\}\!\}_z$
2. $x(y).P + \overline{z}v.Q \xrightarrow{xw}_{\mathfrak{B}} P[w/y]$
3. $x(y).P + \overline{z}v.Q \xrightarrow{xw}_{\mathfrak{B}} (x(y).P + \overline{z}v.Q) \mid \{\!\{w\}\!\}_x$

These transitions are matched by the encoded process in the following way:

1. $[\![x(y).P+\overline{z}v.Q]\!]_{\mathfrak{B}} \xrightarrow{\tau} \nu l(\overline{z}v \mid [\![Q]\!]_{\mathfrak{B}} \mid \bar{l}f \mid [\![x(y).P]\!]_{\mathfrak{B},l}) \equiv \nu l(\bar{l}f \mid [\![x(y).P]\!]_{\mathfrak{B},l}) \mid [\![Q]\!]_{\mathfrak{B}} \mid \overline{z}v$
2. $[\![x(y).P+\overline{z}v.Q]\!]_{\mathfrak{B}} \xrightarrow{xw} \xrightarrow{\tau} \xrightarrow{\tau} \nu l([\![P[w/y]]\!]_{\mathfrak{B}} \mid \bar{l}f \mid [\![\overline{z}v.Q]\!]_{\mathfrak{B},l}) \equiv \nu l(\bar{l}f \mid [\![\overline{z}v.Q]\!]_{\mathfrak{B},l}) \mid [\![P[w/y]]\!]_{\mathfrak{B}}$
3. $(x(y).P + \overline{z}v.Q) \mid \{\!\{w\}\!\}_x \approx_a [\![x(y).P + \overline{z}v.Q]\!]_{\mathfrak{B}} \mid \overline{x}w$

In the case of $\nu l(\bar{l}f \mid [\![x(y).P]\!]_{\mathfrak{B},l})$, the only transitions that can be executed for this process are: $\nu l(\bar{l}f \mid [\![x(y).P]\!]_{\mathfrak{B},l}) \xrightarrow{xz} \xrightarrow{\overline{x}z} \nu l(\bar{l}f)$ Hence, it is weakly asynchronous bisimilar to 0, according the to asynchronous case of the relation.

Also, since $\nu l(\bar{l}f \mid [\![\overline{z}v.Q]\!]_{\mathfrak{B},l})$ can only do a τ transition and become $\nu l(\bar{l}f)$, it is weakly asynchronous bisimilar to 0.

In the other direction, we have the above transitions plus the following one:

$$[\![x(y).P + \overline{z}v.Q]\!]_{\mathfrak{B}} \xrightarrow{xw} R$$

where $R = \nu l(\bar{l}t \mid l(x).((\text{if } x = true \text{ then } [\![P[w/y]]\!]_{\mathfrak{B},l,x(w)} \text{ else } \overline{x}w) \mid \bar{l}f) \mid [\![\overline{z}v.Q]\!]_{\mathfrak{B},l})$. In this case, the choice is not commited: the process can continue with $[\![\overline{z}v.Q]\!]_{\mathfrak{B},l}$ and then release xw, or send xw and come back to its initial state, or receive the value on $x(y).P$. This is matched by the following transition from the original process:

$$x(y).P + \overline{z}v.Q \xrightarrow{xw}_{\mathfrak{B}} (x(y).P + \overline{z}v.Q) \mid \{\!\{w\}\!\}_x$$

Figures 2.1 and 2.2 show the transitions of a typical binary choice and its encoding and how they are related by weak asynchronous bisimilarity. ■

[2]In the third transition, the input could be on a channel different from x. The proof however proceeds in the same way.

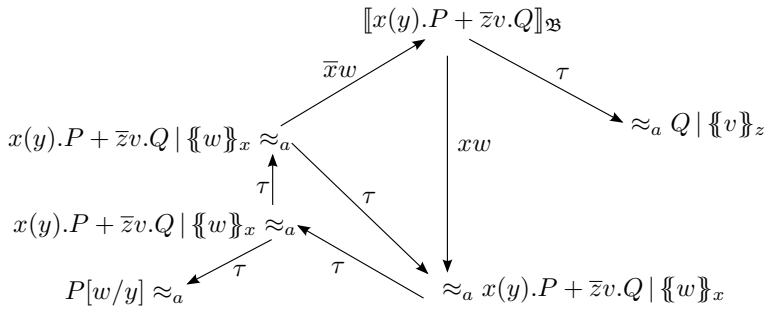


Figure 2.2: Transitions of the π_a encoding of the $\pi_{\mathfrak{B}}$ sum in Figure 2.1.

2.4 Impossibility result for the other types of buffer

In this section, we show the impossibility of encoding other kinds of buffers (i.e. not bags) into the asynchronous π-calculus and into the π-calculus with separate choice. In particular, we show that a calculus with queues and stacks cannot be encoded into π_a up-to weak asynchronous bisimilarity. Then, we show a stronger result for stacks: a calculus with stacks cannot even be encoded, up-to weak asynchronous bisimilarity, in the π-calculus with separated-choice. Note that, since weak bisimilarity is a special case of weak asynchronous bisimilarity, those results also hold up-to weak bisimilarity.

We stress the fact that these results strongly depend on the requirement that a term (in particular a stack or a queue) and its encoding be asynchronously bisimilar. We believe that it is possible to simulate stacks or queues in π_a. Our results only say that it cannot be done via an encoding that satisfies the requirement of translating a process into a weakly asynchronously bisimilar one.

We start by defining the π-calculus with stacks and queues.

Definition 2.4.1

The π-calculus with buffers of type T, written π_T, where T is either $\mathfrak{Q}$ (queues) or $\mathfrak{S}$ (stacks) is the set of processes defined by the grammar:

$$P, Q ::= \textstyle\sum_{i \in I} \alpha_i.P_i \;\Big|\; P \mid Q \;\Big|\; \nu x\, P \;\Big|\; !P \;\Big|\; B_x$$

where B_x represents a buffer of type T.

The operational semantics of π_T is the same as the one defined in Section 2.2, except that the last congruence rule ($P \equiv P \mid \{\!\{\}\!\}_x$) only applies when P is a pure π-calculus process (i.e. not containing a buffer), in order to avoid behaviors that do not represent FIFO or LIFO strategies. Furthermore, the rules for bags (in_{bag} and out_{bag}) should be interpreted as rules for stacks (resp. queues) in

the sense that the transitions in the premises should be those defined for stacks (resp. queues) in Definition 2.1.3.

2.4.1 Impossibility of encoding queues and stacks

In this section we show that it is not possible to find a valid encoding using the π_a-calculus for queues and stacks modulo weak asynchronous bisimilarity.

Theorem 2.4.1

Let $[\![\ ]\!]$ be an encoding from $\pi_{\mathfrak{Q}}$ into π_a (resp. from $\pi_{\mathfrak{S}}$ into π_a). Then there exists $P \in \pi_{\mathfrak{Q}}$ (resp. $P \in \pi_{\mathfrak{S}}$) such that $[\![P]\!] \not\approx_a P$.

Proof: We prove the theorem by contradiction, for $P \in \pi_{\mathfrak{Q}}$. The case of $P \in \pi_{\mathfrak{S}}$ is analogous.

Let P be a queue B_x of the form $\cdots \triangleleft y \triangleleft z$ with $y \neq z$. Then we have:

$$B_x \xrightarrow{\overline{x}z}_{\mathfrak{Q}} \xrightarrow{\overline{x}y}_{\mathfrak{Q}} \tag{2.1}$$

Since we are assuming $[\![B_x]\!] \approx_a B_x$, we also have $[\![B_x]\!] \stackrel{\overline{x}z}{\Longrightarrow}\stackrel{\overline{x}y}{\Longrightarrow}$. By using Lemma 1.1.4 we obtain $[\![B_x]\!] \xrightarrow{\tau} * \xrightarrow{\overline{x}z} \xrightarrow{\overline{x}y} \xrightarrow{\tau} *$. Using Lemma 1.1.4 again we get: $[\![B_x]\!] \xrightarrow{\tau} * \xrightarrow{\overline{x}y} \xrightarrow{\overline{x}z} \xrightarrow{\tau} *$. Since $[\![B_x]\!] \approx_a B_x$ we have $B_x \stackrel{\overline{x}y}{\Longrightarrow}\stackrel{\overline{x}z}{\Longrightarrow}$, and, since a buffer in isolation does not give rise to τ steps, we also have

$$B_x \xrightarrow{\overline{x}y}_{\mathfrak{Q}} \xrightarrow{\overline{x}z}_{\mathfrak{Q}}$$

By the latter, and (2.1), and Remark 2.1.1, we have that B cannot be a queue. ■

Remark 2.4.1

We could give a stronger result, namely that for any encoding $[\![\ ]\!] : \pi_{\mathfrak{Q}} \hookrightarrow \pi_a$ (resp. $[\![\ ]\!] : \pi_{\mathfrak{S}} \hookrightarrow \pi_a$) and any queue (resp. any stack) B_x, $[\![B_x]\!] \not\approx_a B_x$. We leave the proof to the interested reader. The idea is that if B_x contains less than two elements, then we can always make input steps so to get a queue with two elements.

2.4.2 Impossibility of encoding stacks in the π-calculus without mixed-choice operator

In this section we prove that stacks cannot be encoded in π_{sc}-calculus.

Theorem 2.4.2

Let $[\![\ ]\!]$ be an encoding from $\pi_{\mathfrak{S}}$ into π_{sc}. Then there exists $P \in \pi_{\mathfrak{S}}$ such that $P \not\approx_a [\![P]\!]$.

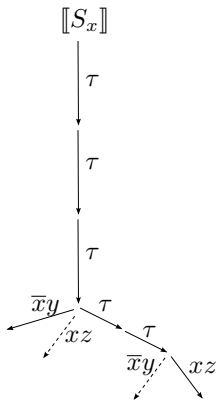


Figure 2.3: Impossibility to encode a stack.

Proof: Let B_x be a stack of the form $y \triangleright \ldots$. Assume by contradiction that $B_x \approx_a [\![B_x]\!]$ (i.e. B_x is *weakly* asynchronously bisimilar to $[\![B_x]\!]$) . Then B_x must be weakly bisimilar to $[\![B_x]\!]$. In fact, if $B_x \xrightarrow{xz}_{\mathfrak{S}} B'_x \approx_a [\![B_x]\!]|\overline{x}z \approx_a B_x|\overline{x}z$, then we would have both $B_x \xrightarrow{\overline{x}y}_{\mathfrak{S}}$ and $B_x \xrightarrow{\overline{x}z}_{\mathfrak{S}}$, which by Remark 2.1.1 is not possible.

Let $z \neq y$. We have $B_x \xrightarrow{\overline{x}y}_{\mathfrak{S}} B'_x$ and $B_x \xrightarrow{xz}_{\mathfrak{S}}$. Since B_x is weakly bisimilar to $[\![B_x]\!]$, we have, for some P, $[\![B_x]\!] \xrightarrow{\tau} * P \xrightarrow{\overline{x}y} \xrightarrow{\tau} *$ and $P \xrightarrow{\tau} * \xrightarrow{xz} \xrightarrow{\tau} *$.

Let us assume that the number of τ steps before P inputs xz is not zero. That is to say, $P \xrightarrow{\tau} P' \xrightarrow{\tau} * \xrightarrow{xz}$. From Lemma 1.1.2, we have that either $P \xrightarrow{xz}$ for any z, or $P' \xrightarrow{\overline{x}y}$. Then, by re-applying this reasoning to each sequence of τ transitions before the input of xz, we eventually get $P \xrightarrow{\overline{x}y}$ and $P \xrightarrow{xz}$. By applying Lemma 1.1.1 we have $P \xrightarrow{\overline{x}y} \xrightarrow{zz} P'$ and $P \xrightarrow{zz} \xrightarrow{\overline{x}y} P'$. From the fact that B_x and $[\![B_x]\!]$ are weakly bisimilar, we get $B_x \xrightarrow{\overline{x}y}_{\mathfrak{S}} \xrightarrow{xz}_{\mathfrak{S}}$ and $B_x \xrightarrow{xz}_{\mathfrak{S}} \xrightarrow{\overline{x}y}_{\mathfrak{S}}$. Finally, we observe that the last sequence is not possible, because after the input action xz a stack can only perform an output of the form $\overline{x}z$.

Figure 2.3 illustrates the fact that the encoded process must have a point where confluence occurs, which is used in this proof. ■

Remark 2.4.2

Also in this case we could give a stronger result, namely that for any encoding $[\![\ ]\!] : \pi_{\mathfrak{S}} \hookrightarrow \pi_{sc}$ *and any stack* B_x, $[\![B_x]\!] \not\approx_a B_x$. *Again the idea is that if* B_x *is not in the right form (i.e. it is empty), then we can make an input step so to get a stack with one element.*

Part II

Asynchronous behavior

Introduction

Asynchronous communication models are based on some properties of the execution of communicating processes. In particular, if a process can do a sending action $\overline{x}y$ and a reception zt, then these two actions confluate, meaning that the resulting process is the same independently from the order in which these two actions are executed. Also, a sending action $\overline{x}y$ may be delayed forever.

When looking at an execution graph of an asynchronous process, wide symmetries appear as the consequence of the confluence properties. For instance, if a process P can do a trace $\xrightarrow{xy}\xrightarrow{zt}$ and another transition $\xrightarrow{ov}$, then it can perform the traces: $\xrightarrow{ov}\xrightarrow{xy}\xrightarrow{zt}$, $\xrightarrow{xy}\xrightarrow{ov}\xrightarrow{zt}$ and $\xrightarrow{xy}\xrightarrow{zt}\xrightarrow{ov}$.

Another interesting property of the asynchronous calculus is that if P is a process such that $P \xrightarrow{\tau}$, then the τ action must be entailed from a communication between two processes in P. Hence, if this communication is done through a channel which is free in P, we may observe the corresponding send and receive actions. Reciprocally, if a process can do a reception on a channel x and an emission in the same channel, then it can do the corresponding τ transition.

Hence, when establishing an observational equivalence between two processes, only the τ actions for which the internal communication occurs on a bound channel should be observed and matched. Indeed, when the internal observation is done through a free channel, then the corresponding send and receive actions are observed too and are sufficient to entail the required τ action.

In other words, the traces $\xrightarrow{\overline{x}y}$ and $\xrightarrow{xz}$ contain enough information, by confluence of the asynchronous semantics, to retrieve the whole transition graph of P, including the possibility to do a τ action. See Figure 2.4 for an illustration of these considerations.

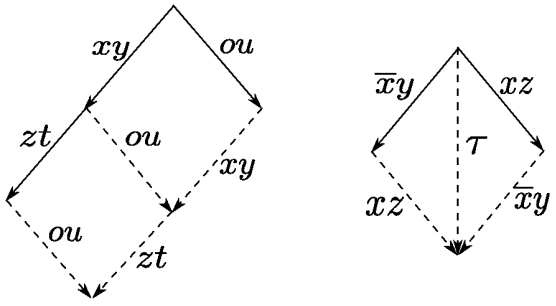


Figure 2.4: Asynchronously confluent processes.

These remarks lead to the intuition that when trying to establish an observational equivalence between two processes, not all observations and executions are relevant. The goal of this part is to characterize some of the relevant classes

of observations, in order to reduce the search space when trying to prove observational equivalence between two processes.

In Chapter 3, we first present some known results about the various usual bisimilarity relations in the various asynchronous calculi. We then present two new results extending the previous work in this topic. In particular, we show that the value received by an asynchronous process is not relevant for the bisimilarity, and that the transition resulting of an internal communication on a free channel need not to be taken into account.

In Chapter 4, we first define a notion of bunched transition system, where transitions are sequences of transitions of the original transition system, preserving confluence and causality. We prove that the bisimilarity relation defined on such a Labeled Transition System is included in the original bisimilarity. We then apply it to the case of the π_a-calculus: using the asynchronous causalities that can be inferred from the grammar of the processes, we define a bunched Labeled Transition System for the π_a-calculus for which the bisimilarity is included into the original bisimilarity.

We conclude with several intuitions about further extensions of this work, as well as details on the relations with the original motivations, which was to speed-up the evaluation, or simulation, of an asynchronous process.

Chapter 3

Equivalent behavior in the π-calculus

Contents

3.1 The late semantics 50
3.1.1 The late transition system 50
3.1.2 Late bisimilarity 51
3.2 Bisimilarities and congruences 52
3.2.1 Congruence under any context 52
3.2.2 Congruent bisimilarities 53
3.3 The asynchronous case 54
3.3.1 Equivalences and congruences 54
3.3.2 Late bisimulation and the input/output π_a-calculus 58

3.1 The late semantics

Introduction

Other operational semantics for the π-calculus than the one described in Definition 1.1 exist. The semantics described in this table is referred as the early semantics. When a process $x(y).P$ receives a value xz, the name y is substituted in every place by the name z. The semantics used in Definition 1.1 presents a transition system where this substitution is done in the (in) reception rule. Although this seems natural, this definition has some drawbacks. In particular, since the process can receive any possible value, each of these possible value entails a transition, leading to an infinite number of possible transitions for the receiving process. This has consequences both in the theoretical models, where infinite branchings can be troublesome, and on practical implementations, where a program should always be represented by a finite object.

Furthermore, the requirement to have a single transition per possible value seems overkill. In particular, we know that if the process $x(y).P$ performs an input transition, then it should become a process where y has been replaced by the received value. Formally speaking, after a reception, all the infinitely many possible cases may all be lumped into a process waiting for the new value.

That is what is achieved with the late semantics and the associated transition system.

3.1.1 The late transition system

Under the late semantics, the (in) rule is replaced by a generic rule of the form:

$$x(y).P \xrightarrow{x(y)}_l P$$

As we can see, no substution is done in this rule. Indeed, substitution happens during the synchronization rule:

$$\frac{P \xrightarrow{\overline{x}z}_l P', \quad Q \xrightarrow{x(y)}_l Q'}{P \mid Q \xrightarrow{\tau}_l P' \mid Q'[z/y]}$$

The receiving process has a single transition and substitution happens when this communication is matched with an available value.

The operational rules of the late transition system are described in Table 3.1. They are almost the same as the early rules, but with the modifications discussed above: the substitution is not longer done in the (in) rule, but in the $(sync)$ one, which is the communication rule, where the actually received value is known. The substitution was not added for the $(close)$ rule. The received value is a bound channel from the sending process. In this case, the value received

$$(in)\ \frac{}{x(y).P \xrightarrow{x(y)}_l P} \qquad (out)\ \frac{}{\overline{x}y.P \xrightarrow{\overline{x}y}_l P}$$

$$(sync)\ \frac{P \xrightarrow{\overline{x}z}_l P',\ Q \xrightarrow{x(y)}_l Q'}{P \mid Q \xrightarrow{\tau}_l P' \mid Q'[z/y]} \qquad (\nu)\ \frac{P \xrightarrow{\alpha}_l P',\ a \notin fn(\alpha)}{\nu a P \xrightarrow{\alpha}_l \nu a P'}$$

$$(open)\ \frac{P \xrightarrow{\overline{x}y}_l P'\ \ x \neq y}{\nu y P \xrightarrow{\overline{x}(y)}_l P'} \qquad (close)\ \frac{P \xrightarrow{\overline{x}(y)}_l P',\ \ Q \xrightarrow{x(y)}_l Q'}{P \mid Q \xrightarrow{\tau}_l \nu y(P' \mid Q')}$$

$$(bang)\ \frac{P \mid !P \xrightarrow{\alpha}_l P'}{!P \xrightarrow{\alpha}_l P'} \qquad (cong)\ \frac{P \equiv P',\ P' \xrightarrow{\alpha}_l Q',\ Q \equiv Q'}{P \xrightarrow{\alpha}_l Q}$$

$$(comp)\ \frac{P \xrightarrow{\alpha}_l P',\ bn(\alpha) \cap fn(Q) = \emptyset}{P \mid Q \xrightarrow{\alpha}_l P' \mid Q} \qquad (sum)\ \frac{P \xrightarrow{\alpha}_l P'}{P + Q \xrightarrow{\alpha}_l P'}$$

Table 3.1: Late operational semantics for the π-calculus

is up to α-conversion. Since the receiving value is also a bound channel, it is also up to α-conversion. Hence, this can be the same value on both sides of the communication, eliminating the need for a substitution in the resulting process.

3.1.2 Late bisimilarity

This transition system entails a different notion of bisimulation. This is due to the fact that if P and Q are bisimilar, and $P \xrightarrow{x(y)}_l P'$, then Q should perform the same action $x(y)$: $Q \xrightarrow{x(y)}_l Q'$, but also, since the actual value is not known, $P'[z/y]$ and $Q'[z/y]$ should be bisimilar for any value z.

Definition 3.1.1 (*strong late bisimilarity*)

Let P and Q be two π-calculus processes. P and Q are late bisimilar if and only if there exists a binary relation $\mathcal{R}$ such that $P\mathcal{R}Q$ and:

- *if $P \xrightarrow{\alpha}_l P'$ and $bn(\alpha) \notin fn(Q)$, then $Q \xrightarrow{\alpha}_l Q'$ and $P'\mathcal{R}Q'$. If $\alpha = x(y)$, then for every possible name z, $P'[z/y]\mathcal{R}Q'[z/y]$*
- *if $Q \xrightarrow{\alpha}_l Q'$ and $bn(\alpha) \notin fn(Q)$, then $P \xrightarrow{\alpha}_l P'$ and $P'\mathcal{R}Q'$. If $\alpha = x(y)$, then for every possible name z, $P'[z/y]\mathcal{R}Q'[z/y]$*

If two processes are bisimilar for the late bisimilarity, we write: $P \sim_l Q$.

In the general case, these bisimilarities are known not to coincide:

Proposition 3.1.1 (*[SW01], Lemma 4.5.3*)

The early bisimilarity is strictly greater than the late bisimilarity: $\sim_l \subsetneq \sim$.

For the late semantics, when a process P performs a transition: $P \xrightarrow{x(y)} P'$ the value y in P' represents a variable that is substituted in the (*sync*) rule, such as: $P \mid \overline{x}z \xrightarrow{\tau} P'[z/y]$. Hence, the name z received in the (*sync*) rule carries a non-deterministic choice and the behaviour of the process $P'[z/y]$ may change depending of this value.

For instance, for the early semantics, when a reception occurs within a choice operator, the choice of the received value is drawn exactly at the same moment as the choice of the continuation, while in the case of the late semantics, the choice of the continuation is resolved before the received variable is known.

Hence, under the early semantics, a process may imitate the behavior of another based on the simultaneous choice of value and continuation, while the late semantics requires a greater set of transitions and states for the imitating process, since it has to match all the possible processes that results of the chosen continuation for any possible received value.

In other words, the correspondence fails because, in the case of the late semantics, there are two different non-deterministic choices, while in the case of the early one, they are lumped into a single choice.

As we will see later, this also relies on the presence of *mixed input and output choice*. When no mixed choice are allowed, then all the usual variants of bisimilarities collapse.

In the following, $\sim$, always denotes the *early bisimilarity*, unless stated explicitly.

3.2 Bisimilarities and congruences

3.2.1 Congruence under any context

A natural requirement for a process equivalence relation is that it is preserved by the application of any context. In particular, if P is equivalent to Q then we may want that any process obtained from the grammar using P would then be equivalent to the same construction using Q. For instance, for any arbitrary process R, $P \mid R$ and $Q \mid R$ are equivalent. Such an equivalence relation is a congruence. As we will see this is unfortunately not true for the strong bisimilarity in the π-calculus.

First, we recall the notion of context for the π-calculus:

Definition 3.2.1

A context in the π-calculus is obtained by the following grammar, where P stands for a regular process:

$$C[.] \;:=\; . \;\Big|\; 0 \;\Big|\; \overline{x}z.C[.] \;\Big|\; x(y).C[.] \;\Big|\; C[.] + P \;\Big|\; P + C[.] \;\Big|\; \nu x C[.] \;\Big|\; C[.] \,|\, P \;\Big|\; P \,|\, C[.] \;\Big|\; !C[.]$$

Where . is a placeholder. For a context $C[.]$ and a process P, $C[P]$ is the process obtained by replacing all occurrences of the . placeholder by P in $C[.]$.
Contexts for the π-calculus variants are defined the same way.

Then, we say that an equivalence relation $\mathcal{R}$ between processes in the π-calculus is a congruence when it is stable under any context. Formally:

Definition 3.2.2

Let $\mathcal{R}$ be an equivalence relation between processes from the π-calculus. $\mathcal{R}$ is a congruence if and only if for any P, Q and any context $C[.]$,

$$P \mathcal{R} Q \Longrightarrow C[P] \mathcal{R} C[Q]$$

3.2.2 Congruent bisimilarities

In the case of the π-calculus, for all possible constructs except input prefix, i.e. $x(y).P$, the bisimilarity is a congruence:

Theorem 3.2.1 (*[SW01], Theorem 2.2.8*)

Let P, Q R be 3 processes from the π-calculus such that P and Q are bisimilar. Then:

- $P \,|\, R \sim Q \,|\, R$
- $\nu x P \sim \nu x Q$
- $P + R \sim P + Q$
- $!P \sim !Q$

However, this is not true for $x(y).P$:

Theorem 3.2.2 (*[SW01], Theorem 2.2.8*)

The strong bisimilarity is not a congruence for the input prefix.

In order to define a congruence from a bisimulation, invariance under renaming needs to be added to restrict the bisimulation relations. A renaming is a function $\sigma : N \longrightarrow N$. For a process P, we write $P\sigma$ to denote the process obtained by replacing all free names in P by their image with σ. For a transition α, $\sigma(\alpha)$ is the transition where free names have been replaced by their image by σ.

Definition 3.2.3

Let $\mathcal{R}$ be a bisimulation relation, either late or early. $\mathcal{R}$ is a congruent bisimulation if and only if for any two processes P, Q such that $P\mathcal{R}Q$, then $P\sigma\mathcal{R}Q\sigma$ for any renaming σ on the free names of P and Q.

It is easy to prove that if $\mathcal{R}$ is a congruent bisimulation, then it is a congruence.

3.3 The asynchronous case

3.3.1 Equivalences and congruences

The considerations above were stated in the general – synchronous – case. However, in the asynchronous case, things are much different. First, most of the issues above came from the non-determinism introduced by the choice operator $+$. The counter examples for the fact that late bisimilarity is more restrictive than early bisimilarity, or that both are not congruences, was built using a mixed choice between an input transition, or an output transition.

In the case where the mixed choice operator is no longer present, one may wonder what happens of the results stated in previous sections. The answer is that they are both invalidated, and it comes from a great difference that arises in the asynchronous case: bisimilarities are preserved under any renaming.

In the following, $\sim$ denotes the early bisimilarity and $\sim_l$ the late bisimilarity.

We first prove the following lemma. A similar lemma can be found in [San95] (Lemma 5.4). The difference is that in this version the renaming is not assumed to be injective, which leads to the second case described below.

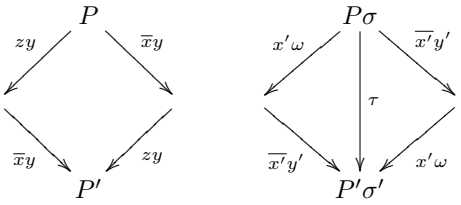
We state and prove this lemma here because it is used in the next chapter to prove the results presented there.

Lemma 3.3.1

Let σ be a renaming, α a transition label and ω a special name used as a variable and not occuring in the processes or transitions.
Let P and P'' be two processes of the π_a-calculus such that $P\sigma \xrightarrow{\alpha} P''$. There exists a renaming σ' such that for any $x \neq \omega$, $\sigma'(x) = \sigma(x)$, and one of the assertions below is true.

1. *There exists γ and P' such that $P \xrightarrow{\gamma} P'$ with $P'' = P'\sigma'$. If $\alpha = \sigma(x)y$, then $\gamma = x\omega$ and $\sigma'(\omega) = y$, otherwise $\alpha = \sigma(\gamma)$ and $\sigma'(\omega) = \omega$.*[1]
2. *There exists x, y, z and x', y' and P' such that:*
   - $\alpha = \tau$
   - $\sigma(\overline{x}y) = \overline{x'}y'$, $\sigma(z) = x'$
   - $\sigma'(\omega) = y'$

   and:



Proof: We prove the lemma by induction on the process' syntax:
Let us consider the various cases:

Input prefix: $P = x(y).P'$
Then $P\sigma = \sigma(x)(y).S$ and the only action that it can perform is: $P\sigma \xrightarrow{\sigma(x)t} S[t/y]$. Then $x(y).P' \xrightarrow{\overline{x}\omega} R$, such that $R\sigma' = S[t/y]$.

Output $P = \overline{x}y$
This case is trivial.

Binding prefix $P = \nu x P'$
In this case, $P\sigma = \nu x(P'\sigma')$, with $\sigma'(x) = x$ and $\sigma'(y) = \sigma(y)$ otherwise, since renaming does not modify bound names. Hence, the transitions of $P\sigma$ are those of $P'\sigma'$ for which x is not a free name. The only particular case is then in the case were assertion 2 holds, with $P'\sigma' \xrightarrow{\overline{x}y} \xrightarrow{xt} P''$. In this case, however, assertion 1 holds for $P\sigma$: since σ' does not modify x, then $P\sigma \xrightarrow{\tau} P''$ and $P \xrightarrow{\tau} P'$ where $P'' = P'\sigma$.

Parallel composition $P = A \mid B$
The only particular case is when there are two original transitions xy and $\overline{z}t$ such that $\sigma(xy) = x'y'$ and $\sigma(\overline{z}t) = \overline{x'}y'$. In this case, then $P\sigma$ has a new τ transition such that $P\sigma \xrightarrow{\tau} P''$. However, since P is a process from the π_a-calculus, then, $P \equiv R \mid S$, where $R \xrightarrow{\overline{z}t} R'$ and $S \xrightarrow{x\omega} S'$. Then $P'' = (R' \mid S')\sigma'$ and assertion 2 is true.

Input/output prefixed choice $P = \sum_{i\in I} x_i(y_i).P_i$ or $P = \sum_{i\in I} \overline{x_i}(y_i).P_i$
In these two cases, the first assertion is obviously true.

Bang composition $P = !P'$
We have: $(!P)\sigma = !(P\sigma)$. If $!(P\sigma) \xrightarrow{\alpha} P'$, then this transition is entailed by a

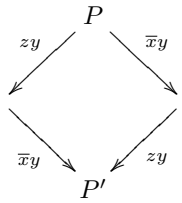
[1] The value in this case does not matter.

finite number of applications of the (*bang*) rule, such that there exists a process $(P\sigma) \mid \ldots \mid (P\sigma)$ so that $(P\sigma) \mid \ldots \mid (P\sigma) \xrightarrow{\alpha} P''$ and $P' = P'' \mid !(P\sigma)$. Hence, the proposition is true since it is true for $(P\sigma) \mid \ldots \mid (P\sigma)$ by a finite number of application of the parallel composition. ■

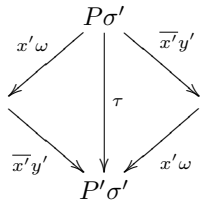
We also prove another lemma, which is specific to the π_a-calculus. This lemma is also used in the next chapter to prove the results preseted there.

Lemma 3.3.2

Let σ be a renaming such that $\sigma(x) = \sigma(z) = x'$, ω a name not occurring elsewhere and P a process of the π_a-calculus. If



Then:



Where $\sigma(y) = y'$, $\sigma'(\omega) = y'$ and $\sigma'(z) = \sigma'(z)$ otherwise.

Proof: By inference on the process' structure, we prove that $P \equiv R \mid S$ where $R \xrightarrow{zy} R'$ and $S \xrightarrow{\overline{x}y} S'$. Hence, $P\sigma' \equiv R\sigma' \mid S\sigma'$ and we obtain the required transitions. ■

Using these lemmas, we can now state the asynchronous cases of the results of previous section. These results are not new, and each of them is linked to its statement in the litterature. However, we prove them again, using the two lemmas that we just stated.

The following proposition is stated in a different form in Lemma 5.3.7 in [SW01]. The statement here is however implied by the lemma of this reference.

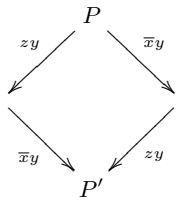
Proposition 3.3.1

Let P and Q be two π_a-calculus processes such that $P \sim Q$, and σ a renaming. Then:

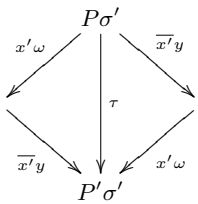
$$P\sigma \sim Q\sigma$$

Proof: Using Lemma 3.3.2, we prove that the relation: $\mathcal{R} = \cup_\sigma \{(P\sigma, Q\sigma) \mid P \sim Q\}$ is a bisimulation relation.

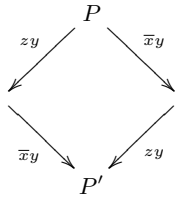
Indeed, if $P\sigma \xrightarrow{\alpha} P''$, then, according to the lemma, either $P \xrightarrow{\gamma} P'$ and $P'\sigma' = P''$, or:



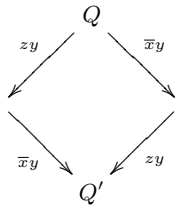
and:



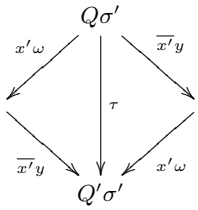
In both cases, since $Q \sim P$, there exists $Q' \sim P'$. Furthermore, in both cases, $Q\sigma \xrightarrow{\alpha} Q'\sigma'$.
In particular, if:



then also:



According to Lemma 3.3.2, then:



Hence, $(P'\sigma', Q'\sigma') \in \mathcal{R}$. ■

Corollary 3.3.1 (*[SW01], Theorem 5.3.11*)

The early and late bisimilarities for the π_a-calculus coincide: $\sim \; = \; \sim_l$.

Proof: We already know that $\sim_l \; \subset \; \sim$.

Let P and Q be two processes for which $P \sim Q$. If $P \xrightarrow{x(y)}_l P'$, then: $Q \xrightarrow{x(y)}_l Q'$, with $P' \sim Q'$. Furthermore, according to the previous result, $P'\sigma \sim Q'\sigma$ for any name renaming σ, so, in particular, $P'[z/y] \sim Q'[z/y]$, for any name z.

Hence, since all the other cases are straight forward, $\sim$ is also a late bisimilarity relation, so $P \sim_l Q$. ■

Corollary 3.3.2 (*[SW01], Theorem 5.3.11*)

The strong bisimilarity is a congruence in the π_a-calculus.

Proof: The only particular case is the input prefix. Let P and Q be two bisimilar processes, then: $x(y).P \xrightarrow{xt} P[t/y]$ and $x(y).Q \xrightarrow{xy} Q[t/y]$. We conclude using Proposition 3.3.1, since $P[t/y] \sim Q[t/y]$. ■

A similar proof can be written for the asynchronous bisimulations. The case of the asynchronous bisimulations is also detailed in [SW01].

3.3.2 Late bisimulation and the input/output π_a-calculus

In this section, we prove a new result about the late bisimilarity. Following the previous results on the (good) relations between the π_a-calculus and bisimulations, we prove that, when working on the late transition system, we do not need to check if the resulting processes remain equivalent under any substitution after the reception of a value. This is because, contrary to the π-calculus, the π_a-calculus behaves much better when renaming free names. In particular, the relations between the transitions before and after the renaming is homogeneous, as detailed in Lemma 3.3.2.

Naive late bisimilarity

We define a late bisimulation without substitution on input transitions. In the following, the transitions follow the Late Transition System, as defined in Table 3.1, and thus we use $\xrightarrow{\alpha}$ for those transitions, without the l subscript.

Definition 3.3.1 (*naive late bisimilarity*)

Let P and Q be two π-calculus processes. P and Q are naively late bisimilar if and only if there exists a binary relation $\mathcal{R}$ such that $P\mathcal{R}Q$ and:

- *if $P \xrightarrow{\alpha} P'$ and $bn(\alpha) \notin fn(Q)$ then $Q \xrightarrow{\alpha} Q'$ and $P'\mathcal{R}Q'$.*
- *if $Q \xrightarrow{\alpha} Q'$ and $bn(\alpha) \notin fn(P)$ then $P \xrightarrow{\alpha} P'$ and $P'\mathcal{R}Q'$.*

If two processes are bisimilar for the naive late bisimilarity, we write: $P \sim_{nl} Q$.

We prove that the naive late bisimilarity coincide with the late bisimilarity for the asynchronous π_a-calculus:

Theorem 3.3.1

The late bisimilarity and the naive late bisimilarity coincide on the π_a-calculus:

$$\sim_l \; = \; \sim_{nl}$$

Proof: By using Lemma 3.3.2, we prove, similarly to the early and late case, that $\mathcal{R} = \cup_\sigma\{(P\sigma, Q\sigma) \mid (P, Q) \in \sim_{nl}\}$ is a naive late bisimulation.

Hence, if $P \sim_{nl} Q$ and $P \xrightarrow{x(y)} P'$ such that $Q \xrightarrow{x(y)} Q'$ and $P' \sim_{nl} Q'$, then $P'[z/y] \sim_{nl} Q'[z/y]$, such that $\sim_{nl} \subset \sim_l$.

It is also clear that $\sim_l \subset \sim_{nl}$. ■

This result has an important consequence for the theory. It states that, in the case of the π_a-calculus, the transition graph generated by the late labeled transition system represents enough information to characterize the late bisimilarity (hence the early, which coincides). Indeed, the late labeled transitions do not represent the substitution after a reception, whereas the early transitions do. Hence, for the case of the π-calculus, there can be two processes which have the same late transition graph but are not late bisimilar.

Input/output bisimilarity

As explained previously, in the π_a-calculus, when a process performs a τ transition, this transition is always related to an internal communication. However, if the communication happens on a channel that is free, we also observe the corresponding sending and receiving actions. Furthermore, the process resulting of the τ transition can also be known by looking at the two processes obtained after the sending and receiving actions.

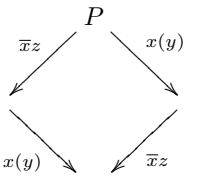
Since, reciprocally, when a process is able to do a sending and the reception on a free channel, then it can do the corresponding internal τ transition, we get the intuition that it is not necessary to observe the τ transitions occurring on a free channel, since we already observe the corresponding sending and receiving actions.

This also is consistent with the results in Chapter 2, since the internal τ transition may be executed through a buffer, which means sending to the buffer and then receiving from it.

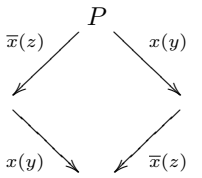
The results in this section formalize this intuition. We define a bisimulation where we do not match the τ transitions resulting from a communication on a free channel, and we prove that this bisimilarity, for the π_a-calculus, coincide with the original bisimilarity. We use the late formalism.

Definition 3.3.2

The input/output π-calculus, written π_{io}-calculus, is the calculus where processes are processes of the π_a-calculus, and operational semantics is given in Table 3.1, except for the (sync) rule, which is replaced by:



$$(sync)\ \frac{P \xrightarrow{\overline{x}z} P' \quad P \xrightarrow{x(y)} P'}{\nu x P \xrightarrow{\tau} \nu x P'[z/y]}$$



$$(sync_b)\ \frac{P \xrightarrow{\overline{x}(z)} P' \quad P \xrightarrow{x(y)} P'}{\nu x P \xrightarrow{\tau} \nu x \nu z P'[z/y]}$$

Also, the rule (close) is removed from the operational semantics.[2]

The operational semantics of the π_{io}-calculus formalizes the considerations mentioned above. Only the (*sync*) communication rule is modified in order to only applies to internal communication, while any public communication must now pass through a sending and a receiving action.

We can now define the input/output bisimulation for this language:

Definition 3.3.3 (*strong input/output late bisimilarity*)

Let P and Q be two π-calculus processes. P and Q are bisimilar if and only if there exists a binary relation $\mathcal{R}$ such that $P\mathcal{R}Q$ and:

- *if $P \xrightarrow{\alpha} P'$ and $bn(\alpha) \notin fn(Q)$ then $Q \xrightarrow{\alpha} Q'$ and $P'\mathcal{R}Q'$.*
- *if $Q \xrightarrow{\alpha} Q'$ and $bn(\alpha) \notin fn(P)$ then $P \xrightarrow{\alpha} P'$ and $P'\mathcal{R}Q'$.*

If two processes are bisimilar for the input/output late bisimilarity, we write: $P \sim_{io} Q$.

Now we state the correspondence:

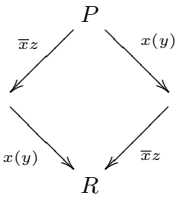
Theorem 3.3.2

The strong late bisimilarity and the input/output bisimilarity coincide:

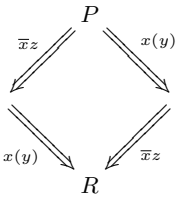
$$\sim_{io} \ = \ \sim_l$$

[2] This rule is redundant with the others so it could also have been removed in the original operational semantics.

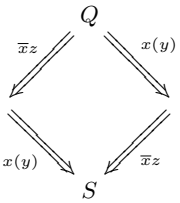
Proof: We prove that the input/output bisimilarity and the naive late bisimilarity coincide. According to Theorem 3.3.1, this is sufficient to prove this result.
We write $P \stackrel{\alpha}{\Longrightarrow} Q$ if P can do an α transition and become Q for the input/output semantics, and $P \stackrel{\alpha}{\rightarrow} Q$ if P can do an α transition and become Q for the naive late semantics.
The only difficult case is to prove that $\sim_{io} \subset \sim_{nl}$. In this situation, the only difficult case is for $P \sim_{io} Q$ such that: $P \stackrel{\tau}{\rightarrow} P'$ for which the (*sync*) (or (*close*)) of the late semantics was applied. Then there exists x, y and z such that:



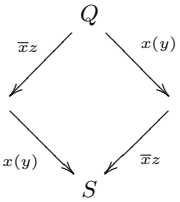
Furthermore, we have: $R[z/y] = P'$. Hence, since the input and output rules are the same for the late π_a-calculus and the π_{io}-calculus, then we also have:



Since P and Q are input/output bisimilar, then there exists S such that $R \sim_{io} S$ and:



Again, since the operational rules in this case are the same, we have:



As usual, we then have: $Q \stackrel{\tau}{\rightarrow} S[z/y] = Q'$.
Since $R \sim_{io} S$, then, by applying the same reasoning as for Theorem 3.3.1, we get: $R[z/y] = P' \sim_{io} S[z/y] = Q'$
Eventually, we get that: $P \stackrel{\tau}{\rightarrow} P'$, and $Q \stackrel{\tau}{\rightarrow} Q'$ with $P' \sim_{io} Q'$. The same reasoning can be also done for the case of rule (*close*) using the ($sync_b$) rule. ■

Chapter 4

Causality and asynchrony: speeding up the scheduler

Contents

4.1 Bunched Labeled Transition Systems 64
4.1.1 Bunched transition systems 64
4.1.2 Bunched traces 66
4.1.3 Bunched Bisimilarity 68
4.2 Bunched transition system for the π_a-calculus . . 70
4.2.1 Bunched transition system 70
4.3 Further discussions 73
4.3.1 Bunched Concurrent Transitions 73
4.3.2 Maximal Bunched Transitions 74
4.3.3 Bunched evaluation strategy and schedulers 76

Introduction

Assume that we have a labeled transition system and consider a second labeled transition system which transitions are sequences of transitions of the first system. In this chapter, we give conditions under which the bisimilarity of the second is a bisimulation of the first. Such a labeled transition system is then called a "bunched labeled transition system".

We then define a new transition system for the π_a-calculus where transitions consist of sequences of transitions from the original transition system. We prove that this labeled transition system is a bunched transition system, for which the induced bisimilarity is a bisimilarity for the original transition system.

Finally, we give intuitions about further extensions on this topic, in particular how to maximize the bunched transition system so that the bisimilarities induced coincide with the original bisimilarity.

The work on this chapter has been inspired by a technique known in logic as focusing. The author believes that the definitions and results are the transposition of this technique to the Labeled Transition Systems and the π_a-calculus. However, the word focusing has been omitted on purpose in order to avoid confusing the reader. The connection with focusing is detailed intuitively in Section 4.3.1.

4.1 Bunched Labeled Transition Systems

In this section, we give the conditions under which a transition system where each transition can be seen as a *bunch* of transitions of another transition system. Intuitively, a bunched labeled transition system is a transition system for which each transition consist of an arbitrary large number of transitions from the original system, with some consistency conditions of confluence. Eventually, we prove that the bisimularity relation induced on the bunched transition system is included in the bisimilarity relation induced on the original one.

The idea behind this is that the large transition of the bunched transition system executes deterministic sequences of computation. Hence, it allows to reduce as much as possible the confluence and interleavings between the various possible transitions, by executing bunched simultaneous executions of several transitions.

4.1.1 Bunched transition systems

First, we define the notion of labeled transition system:

Definition 4.1.1 (*Labeled Transition system*)

Let S and N be two sets. A labeled transition system (LTS) is a set of triplets $LTS \subset S \times S \times N$.
The elements of S are the states of the LTS, and the elements of N are the labels of the LTS. We write "$P \xrightarrow{l} P'$" when $(P, P', l) \in LTS$.

The notion of bisimulation extends to LTSs:

Definition 4.1.2

Let LTS be a labeled transition system whose set of states is X and labels N.
Let $\mathcal{R}$ be a binary relation on X.
$\mathcal{R}$ is a bisimulation relation if and only if for any $P, Q \in \mathcal{R}$,

- *if $P \xrightarrow{l} P'$, then there exist Q' such that $Q \xrightarrow{l} Q'$ and $P', Q' \in \mathcal{R}$*
- *if $Q \xrightarrow{l} Q'$, then there exist P' such that $P \xrightarrow{l} P'$ and $P', Q' \in \mathcal{R}$*

$\sim$ is the greatest binary relation on LTS that satisfies these conditions.

Not all notions of bisimulations coincide with this generalization. For instance, it does not extend to the late bisimulation relations, since in this case, the labeled transition system does not represent all the required information for establishing a correct bisimulation. However, in the case of the π_a-calculus, they coincide, as proved in Section 3.3.1.

We define now the notion of bunched transition system:

Definition 4.1.3

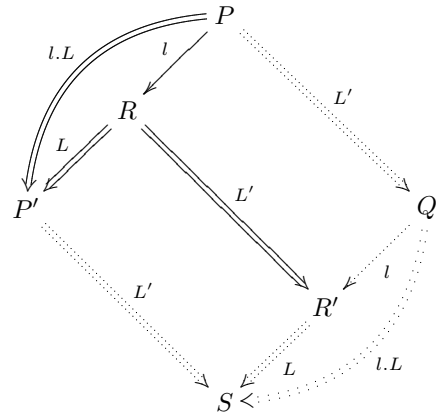
Let LTS be a transition system whose labels are in the set N and LTS_f a transition system whose labels are finite sequences of elements of N and states the same as for LTS. We write $P \xrightarrow{\alpha} P'$ for the transitions of LTS, and $P \stackrel{\alpha}{\Longrightarrow} P'$ for the transitions of LTS_f. When writing $P \stackrel{l.L}{\Longrightarrow} P'$, L can be empty, meaning that $P \stackrel{l}{\Longrightarrow} P'$. When L is an empty sequence we may write: $P \stackrel{L}{\Longrightarrow} P$ for notation simplicity.
LTS_f is a bunched transition system of LTS if and only if:

reciprocity *If $P \stackrel{l.L}{\Longrightarrow} Q$, then there exists P' such that: $P \xrightarrow{l} P'$ and $P' \stackrel{L'}{\Longrightarrow} Q$.*

extensivity *If $P \xrightarrow{l} P'$ then there exists L such that: $P \stackrel{l.L}{\Longrightarrow} P''$ and $P' \stackrel{L}{\Longrightarrow} P''$.*

compatibility *If $P \stackrel{l.L}{\Longrightarrow} Q \stackrel{L'}{\Longrightarrow} S$, where L is not empty, and $P \stackrel{L'}{\Longrightarrow} Q' \stackrel{l.L}{\Longrightarrow} S$, and, by reciprocity, $P \xrightarrow{l} U \stackrel{L}{\Longrightarrow} Q$ and $Q' \xrightarrow{l} V \stackrel{L}{\Longrightarrow} S$ then $U \stackrel{L'}{\Longrightarrow} V$.*

consistency *If $P \stackrel{l.L}{\Longrightarrow} P'$, where L is not empty, with $P \xrightarrow{l} R \stackrel{L}{\Longrightarrow} P'$, and $R \stackrel{L'}{\Longrightarrow} R'$ where $L \neq L'$ or $P' \neq R'$, then there exists Q and S such that:*



The intuition behind a bunched transition system, is that each transition in the bunched system corresponds to a sequence of transitions from the original system, for which there was no interleavings. The *reciprocity* and *extensivity* properties assure the first hypothesis, that each bunched transition corresponds to a sequence of elementary transitions. The *compatibility* and *consistency* properties assure the fact that two different bunched transitions corresponds to two different sequences of original transitions, for which confluence occurred.

In the following, we will write ϵ to denote the empty sequence.

4.1.2 Bunched traces

Traces are sequences of transitions of a LTS. The definition of a bunched transition system implies some correspondences between the original transitions an the

bunched transitions. Even more, a bunched transition is a trace for the original transition system. A natural question that arises then is to compare the traces of the original transitions system with the *bunched traces*. This section shows that the two transition system are equivalent for the testing semantics.

First, we define a trace:

Definition 4.1.4

Let LTS be a transition system. A trace of a state P is a (possibly infinite) sequence L such that for any finite prefix $L' = l_1 \dots l_n$ of L, then there exists P', such that: $P \xrightarrow{l_1} \dots \xrightarrow{l_n} P'$.

A particular way of characterizing a process is the trace semantics. In the trace semantics, a process P is composed in parallel with a test T. The test T is a context obtained with the same grammar as P, plus a special symbol, say θ. Then, we say that the process $P \,|\, T$ passes the test T for the trace L, written $P \vdash_L T$, if and only if there is a trace L starting with $P \,|\, T$ such that L contains θ. In the case of a bunched transition system, we write $P \models_L T$.

We can then prove the following correspondence:

Theorem 4.1.1

Let LTS be a labeled transition system, and LTS_f a bunched transition system for LTS. Let P be a process and T a test.

$$\exists L,\ P \vdash_L T \Leftrightarrow \exists L',\ P \models_{L'} T$$

Proof: By reciprocity, it clear that if $P \models_L T$, then also $P \vdash_L T$.

In order to simplify the notations, for any trace L, we will write $P \xrightarrow{L} P'$ and $P \stackrel{L}{\Longrightarrow} P'$ to specify that P can generate the trace L and become P'.

Reciprocally, we can proceed by induction, with this proposition: $\mathcal{P}(n)$ holds if and only if, for any trace L of length n of any process P, if $P \xrightarrow{L} P'$ and L contains θ, then there exists L' containing θ such that $P \stackrel{L'}{\Rightarrow} P''$.

Let $L = l.L'$ be a trace of length $n+1$ such that $\mathcal{P}(n)$ holds and $P \vdash_L T$. If $l = \theta$, then by extensivity, the result is proved. Otherwise, $P \,|\, T \xrightarrow{l} P'$ and $P' \xrightarrow{L'} P''$ where L' contains θ and is of length n.

According to $\mathcal{P}(n)$, there exists a trace L'' such that $P' \stackrel{L''}{\Longrightarrow} R$ and L'' contains θ. Then, according to extensivity and compatibility, there must exist a trace $S = l.S'$ containing θ such that: $P \,|\, T \stackrel{S}{\Longrightarrow}$ and $\mathcal{P}(n+1)$ holds.

Of course, $\mathcal{P}(1)$ holds by extensivity. ■

4.1.3 Bunched Bisimilarity

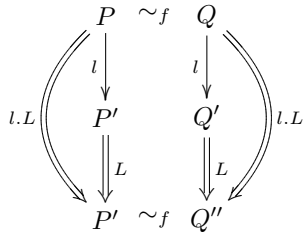
If we have a LTS and a bunched LTS for this LTS, this gives two notions of bisimulation, the first on the original LTS, and the second on the bunched one. Then, one may like to compare the two. The result of this is that the bisimilarity on the bunched transition system is included in the bisimilarity on the original LTS, meaning that two bunched-bisimilar processes are also bisimilar for the original LTS.

In the following, LTS is a labeled transition system, and LTS_f a bunched transition system for LTS. $P \sim Q$ denotes two states bisimilar for LTS, and $P \sim_f Q$ two states bisimilar for LTS_f.

First, we define the notion of one-step extension on a bunched transition system.

Definition 4.1.5 (*One-step extension*)

Let P and Q be two states such that $P \sim_f Q$.
A pair (P', Q') of states extends P and Q by one step if and only if there exists $L \neq \epsilon$ and l such that:



$Ex(P, Q)$ denotes the set of all such pairs of states (P', Q').

Now, we prove the following lemma, which states that bisimulation is stable by one-step transitions:

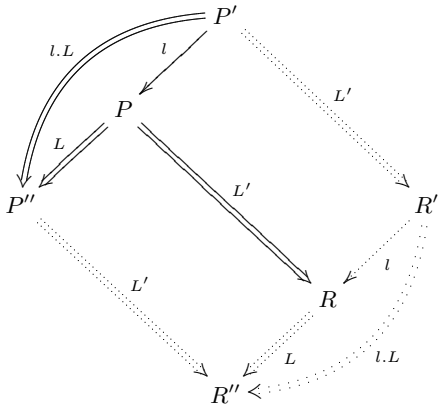
Lemma 4.1.1 (*one-step stability*)

Let $\mathcal{R}$ be a bisimulation relation for LTS_f, and $(P, Q) \in \mathcal{R}$.
Then $\mathcal{S} = \mathcal{R} \cup_{(P,Q)\in\mathcal{R}} Ex(P, Q)$ is also a bisimulation.

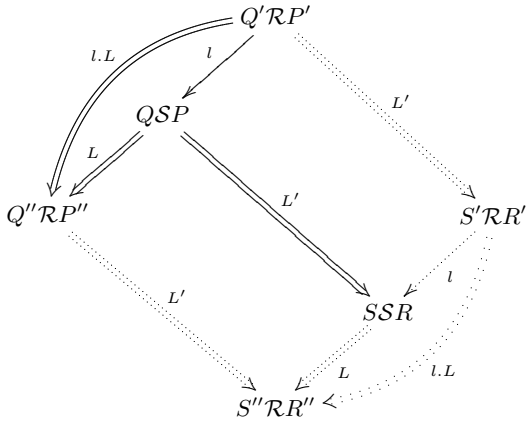
Proof: Let $(P, Q) \in \mathcal{S}$. The only difficult case is when $(P, Q) \notin \mathcal{R}$.
Suppose that $(P, Q) \in \mathcal{S} \setminus \mathcal{R}$.
Then there exists $(P', Q') \in \mathcal{R}$ such that (P, Q) extends (P', Q'), i.e. there exists $L \neq \epsilon$ and l such that:
$P' \stackrel{l.L}{\Longrightarrow} P''$ and $Q \stackrel{l.L}{\Longrightarrow} Q''$, and $P'' \sim_f Q''$.
$P' \stackrel{l}{\rightarrow} P \stackrel{L}{\Longrightarrow} P''$ and $Q' \stackrel{l}{\rightarrow} Q \stackrel{L}{\Longrightarrow} Q''$.
Since $\mathcal{R}$ is a bisimulation relation, we also have: $(P'', Q'') \in \mathcal{R}$.
Suppose that $P \stackrel{L'}{\Longrightarrow} R$.

- If $L' = L$ and $R = P''$, by hypothesis, $Q \stackrel{L}{\Longrightarrow} Q''$, and $(P'', Q'') \in \mathcal{R} \subset \mathcal{S}$.

- Otherwise, by consistency, there exists R, R' such that :



Since $(P', Q') \in \mathcal{R}$, $(P, Q) \in \mathcal{R}$, and $\mathcal{R}$ is a bisimulation relation, we also have:



Indeed, there exists S' such that $Q' \stackrel{L'}{\Longrightarrow} S'$, with $(R', S') \in \mathcal{R}$. Again, since then $R' \stackrel{l.L}{\Longrightarrow} R''$ and $P'' \stackrel{L'}{\Longrightarrow} R''$, then there exists S'' for which $S' \stackrel{l.L}{\Longrightarrow} S''$ and $Q'' \stackrel{L'}{\Longrightarrow} S''$ with $(R'', S'') \in \mathcal{R}$.

By reciprocity, there exists S such that: $S' \stackrel{l}{\rightarrow} S$ and $S \stackrel{L}{\Longrightarrow} S''$.

Since $Q' \stackrel{l.L}{\Longrightarrow} \stackrel{L'}{\Longrightarrow} S''$ and $Q' \stackrel{L'}{\Longrightarrow} \stackrel{l.L}{\Longrightarrow} S''$, by compatibility, $Q \stackrel{L'}{\Longrightarrow} S$.

We conclude by the fact that: $(R, S) \in \mathcal{S}$ since $(R, S) \in Ex(R', S')$. ■

Now, we can state the main result:

Theorem 4.1.2

Let LTS be a transition system, and LTS_f a bunched transition system for LTS. Let $\sim$ (resp. $\sim_f$) be the bisimilarity induced by LTS (resp. LTS_f), then:

$$\sim_f \subset \sim$$

Proof: Let $(P,Q) \in \sim_f$. Suppose that: $P \xrightarrow{l} P'$. We must prove that there exists Q' such that: $Q \xrightarrow{l} Q'$ with $Q \sim_f Q'$.

By extensivity, there exists L such that $P \xrightarrow{l.L} P''$, and $P' \stackrel{L}{\Longrightarrow} P''$.

Since P and Q are bisimilar for LTS_f, there exists Q'' such that: $Q \stackrel{l.L}{\Longrightarrow} Q''$ and $P'' \sim_f Q''$.

By reciprocity, there exists Q' for which $Q \xrightarrow{l} Q'$ and $Q' \stackrel{L}{\Longrightarrow} Q''$.

Then, $(P',Q') \in Ex(P,Q)$.

Since $\sim_f$ is the greatest bisimulation, and since $\sim_f \cup_{(P,Q)\in\sim_f} Ex(P,Q)$ is a bisimulation, $\sim_f \subset \sim_f \cup_{(P,Q)\in\sim_f} Ex(P,Q) \subset \sim_f$.

Hence, $\sim_f = \sim_f \cup_{(P,Q)\in\sim_f} Ex(P,Q)$ and $(P',Q') \in \sim_f$. ■

The converse result is not true in general. The issue being that, if $P \sim Q$, then if $P \stackrel{L}{\Longrightarrow} P'$, with $L = l_1 \dots l_n$, then, by reciprocity, $P \xrightarrow{l_1} \dots \xrightarrow{l_n} P'$ for LTS.

Since Q is bisimilar to P, then $Q \xrightarrow{l_1} \dots \xrightarrow{l_n} Q'$, but it cannot be proved in the general case that then $Q \stackrel{L}{\Longrightarrow} Q'$.

4.2 Bunched transition system for the π_a-calculus

In this section, we use Theorem 4.1.2 to define a bunched transition system for the input/output π_a-calculus. As proved in the previous section, the bisimilarity induced by this transition system is a bisimulation for the original process.

4.2.1 Bunched transition system

When trying to establish a bunched transition system, the main issue that has to be sorted out is how far can the bunched transition go. The general idea is that we would like to go as far as possible while there is only one possible transition, or, alternatively, until there is an execution choice to do between two non-confluent continuations. However, this should also be quite easy to compute, since we aim at using the syntax of the process to infer the bunched transitions.

For the output transitions, there is no continuation, so the real question arises for input and τ transitions. A first approach would be to consider the processes of the form: $x(y).z(t)\dots P$ and perform all the inputs. However, this naive strategy would not group, for instance, $x(y).z(t)$ in $x(y).(0 \mid z(t).P)$. Hence, we will proceed with this idea, but up to any structural equivalence. The structural equivalence being quite simple to compute, we can still claim that we base our bunched transitions on the syntax of the processes.

Secondly, we only observe the internal communications on private channels. Communications occurring internally on a public channel do not need to be observed, according to Theorem 3.3.2.

In the following, we write $\xrightarrow{\alpha}$ for the transitions of the input/outout LTS, and $\stackrel{\alpha}{\Longrightarrow}$ for the transitions of the bunched LTS.

$$(in)\ \frac{P \not\equiv x(y).P',\ L = x_1(y_1)\ldots x_n(y_n)}{x_1(y_1)\ldots x_n(y_n).P \stackrel{L}{\Longrightarrow} P} \qquad (out)\ \frac{}{\overline{x}y \stackrel{\overline{x}y}{\Longrightarrow} 0}$$

$$(sync)\ \frac{P \stackrel{\overline{x}z}{\Longrightarrow} \stackrel{x(y)}{\Longrightarrow} P}{\nu x P \stackrel{\tau}{\Longrightarrow} \nu x(P'[z/y])} \qquad (\nu)\ \frac{P \stackrel{\alpha}{\Longrightarrow} P',\ a \notin fn(\alpha)}{\nu a P \stackrel{\alpha}{\Longrightarrow} \nu a P'}$$

$$(open)\ \frac{P \stackrel{\overline{x}y}{\Longrightarrow} P'\ \ x \neq y}{\nu y P \stackrel{\overline{x}(y)}{\Longrightarrow} P'} \qquad (in_b)\ \frac{P \stackrel{L.x(y).L'}{\Longrightarrow} P'',\ x \notin fn(L),\ P \stackrel{L}{\Longrightarrow} P'}{\nu x P \stackrel{L}{\Longrightarrow} \nu x P'}$$

$$(bang)\ \frac{P \,|\, !P \stackrel{\alpha}{\Longrightarrow} P'}{!P \stackrel{\alpha}{\Longrightarrow} P'} \qquad (cong)\ \frac{P \equiv P',\ P' \stackrel{\alpha}{\Longrightarrow} Q',\ Q \equiv Q'}{P \stackrel{\alpha}{\Longrightarrow} Q}$$

$$(comp)\ \frac{P \stackrel{\alpha}{\Longrightarrow} P',\ bn(\alpha) \cap fn(Q) = \emptyset}{P \,|\, Q \stackrel{\alpha}{\Longrightarrow} P' \,|\, Q} \qquad (sync_b)\ \frac{Q \stackrel{\overline{x}(z)}{\rightarrow} Q',\ P \stackrel{L.x(y).L'}{\Longrightarrow} P'',\ P \stackrel{L}{\rightarrow} P',\ x \notin fn(L)}{P \,|\, Q \stackrel{L}{\Longrightarrow} P' \,|\, Q}$$

Table 4.1: Operational semantics for the bunched π_a-calculus.

Definition 4.2.1

The bunched asynchronous π-calculus, written π_f-calculus is the language where processes are those of the π_a-calculus, and operational semantics is described in Table 4.1.

The meaning of this operational semantics is to group simultaneous sequentialized input. Hence, $x(y).z(t).0$, for instance, has single transition, labeled $x(y).z(t)$. As for the π_{io}-calculus, we do not allow internal communication on a free channel, only in the case of a private channel, as enforced by rule $(sync)$.

The first and immediate result on this language is the following:

Proposition 4.2.1

If $P \xrightarrow{\alpha} P'$ and α is not an input action, then $P \stackrel{\alpha}{\Longrightarrow} P'$.

Proof: This is an immediate consequence of the operational semantics. ■

We can now state the main result of this chapter:

Theorem 4.2.1

The labeled transition system for the π_f-calculus is a bunched labeled transition system of the π_{io}-calculus.

Proof: The proof follows directly from the respective operational semantics.

- If $P \stackrel{l.L}{\Longrightarrow} P''$, where $L = x_1(y_1) \dots x_n(y_n)$, then $l = x(y)$, $P \equiv x(y).P'$ and $P' \equiv x_1(y_1) \dots x_n(y_n).P''$, so reciprocity is true.
- If $P \stackrel{l}{\rightarrow} P'$, then if l is not an input action, $P \stackrel{l}{\Longrightarrow} P'$. Otherwise $l = x(y)$, and let $L = x_1(y_1) \dots x_n(y_n)$ be the maximal sequence (possibly empty) such that: $P \equiv x(y).x_1(y_1) \dots x_n(y_n).P''$. Then, $P \stackrel{x(y)}{\rightarrow} P'$, and $P' \stackrel{L}{\Longrightarrow} P''$. Hence, extensivity is true.
- If $P \stackrel{l.L}{\Longrightarrow} Q \stackrel{L'}{\Longrightarrow} S$ and $P \stackrel{L'}{\Longrightarrow} Q' \stackrel{l.L}{\Longrightarrow} S'$. By a case analysis on the operational semantics, those two transitions must be inferred from two different sub-processes that can only be composed in parallel. Hence, $P \equiv P_1 \mid P_2$ where $P_1 \stackrel{l.L}{\Longrightarrow} P_1''$ and $P_2 \stackrel{L'}{\Longrightarrow} P_2'$. By reciprocity, $P_1 \stackrel{l}{\rightarrow} P_1' \stackrel{L}{\Longrightarrow} P_1''$, and $P \stackrel{l}{\rightarrow} P_1' \mid P_2 \stackrel{L'}{\Longrightarrow} P_1' \mid P_2'$.

  Similarly, $Q' \equiv P_1 \mid P_2' \stackrel{l}{\rightarrow} P_1' \mid P_2' \stackrel{L}{\Longrightarrow} P_1'' \mid P_2'$, and compatibility is true.
- Consistency is the most important and difficult part. If $P \stackrel{l.L}{\Longrightarrow} P'$ where $P \stackrel{l}{\rightarrow} R \stackrel{L}{\Longrightarrow} P'$ by reciprocity, and $R \stackrel{L'}{\Longrightarrow} P''$ with $L' \neq L$ or $P'' \neq P'$. Then, the transitions in L' can only be input transitions. Indeed, in all other cases, the operational semantics breaks the bunched transitions when those actions are just available, thanks to rules (in_b) and ($sync_b$). Hence, as usual, this sequence of actions must be in parallel in the initial process P such that $P \equiv S \mid T$ with $S \stackrel{l.L}{\rightarrow} S'$ and $T \stackrel{L'}{\rightarrow} T$, and consistency is true. ■

Hence, any process that is bisimilar for the bunched labeled transition system is bisimilar for the original bisimulation.

Corollary 4.2.1

Let $\sim_f$ be the bisimilarity induced by the bunched transition system of the π_f-calculus, then: $\sim_f \subset \sim$. Hence, if P and Q are two processes of the π_a-calculus such that $P \sim_f Q$, then $P \sim Q$.

Proof: Since the π_f-calculus is a bunched transition system of the π_{io}-calculus, if $P \sim_f Q$, then $P \sim_{io} Q$, where $\sim_{io}$ is the bisimilarity induced by the input/output transition system.

According to Theorem 3.3.2, then $P \sim_{io} Q$. Eventually, $P \sim Q$, according to Theorem 3.3.2 ■

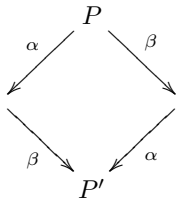
4.3 Further discussions

The results stated in previous section allow to define a bunched transition system for the π_a-calculus. Using this transition system, the number of states for the processes is reduced with regard to the original number of states. Furthermore, we proved that two bisimilar processes for this transition system are bisimilar for the original transition system.

However, this work may be extended and discussed in several ways. We list some of them in this section.

4.3.1 Bunched Concurrent Transitions

In this work, we have forced sequentialized inputs to happen in a single transition. The idea behind is that it didn't change the result of the computation, but only removed some possible interleavings. Similarly, if a process P can do the transitions:



Then it can be claimed that the two transitions are confluent, in the sense that the order in which they appear does not change the overall computation.

Hence, another type of bunched Transition System, say Concurrent Bunched Transition System, could have a single transition in this case: $P \stackrel{\alpha \mid \beta}{\Longrightarrow} P'$. For the bunched π_a-calculus, this would be the case, for instance, with all the possible outputs: $\overline{x}y \mid \overline{z}t \stackrel{\overline{x}y \mid \overline{z}t}{\Longrightarrow} 0$.

When doing both (Sequentialized) Bunched Transitions and Concurrent Bunched Transitions, the transition labels would then be obtained from the following grammar:

$$T ::= l \Big| T.T \Big| T \mid T$$

Where l is a label of the original labeled transition system.

If the notion of Concurrent Bunched Transition System is well defined, then the bisimilarity for this labeled transition system should also imply the bisimilarity on the original one.

The two combination of Sequentialized and Concurrent optimization also relates to the two main property of asynchronous systems, as explained previously:

confluence and commutativity of the outputs.

This intuition is also a very good way to understand the connections of this work with the focusing in logic and, in particular, in linear logic, as defined in [And92]. Focusing is defined as a an alternation between asynchronous and synchronous phases, as explained in [MS07]. During an asynchronous phase, all the actions for which the syntax indicates that they are trivially confluent are executed. Then, during a synchronous phases, all the possible syntactic sequentialized actions are executed. The synchronous phases correspond to the Sequentialized Bunched Transitions, and the asynchronous phases correspond to the Concurrent Bunched Transitions.

4.3.2 Maximal Bunched Transitions

Another question that could be investigated is the fact that we only proved an inclusion of the bunched bisimilarity into the original bisimilarity. Hence, it would be interesting to give conditions under which the two bisimilarities coincide.

Looking at the application to the π_a-calculus, one may notice that this is not an easy issue. For instance, the two processes: $!(x(y).x(y))$ and $!x(y)$ are bisimilar. However, a natural strategy to group transitions based on the syntax of the process would give $x(y).x(y)$ as the possible transitions of the first one, and $x(y)$ for the second. Hence, they would not be bisimilar for the bunched transition system.

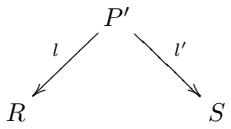
More generally, the bunched bisimilarity introduces distinctions on the processes that are generated from their respective syntax, while the original bisimilarity only considered possible transitions and observable. Hence, it can be expected that the bunched bisimilarity differentiate processes based on syntactic differences, while the original did not.

Then, if we plan to obtain a full correspondence between the bunched bisimilarity and the original bisimilarity, it is necessary to give more informations about the relations between observable behavior and bunched transitions.

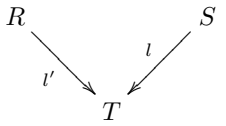
For instance, we can prove the following proposition:

Proposition 4.3.1

Let LTS_f be a bunched transition system of LTS and P a process.
We say that the bunched transition system is maximal for P if:
$P \stackrel{L}{\Longrightarrow} P'$ if and only if there exists no transitions from P' which are not confluent, i.e. for any labels l and l' and processes R and S such that:



There is no transitions of the form:



We say that a process P is recursively maximal if for any bunched trace: $P \stackrel{L}{\Longrightarrow} \ldots \stackrel{L'}{\Longrightarrow} P'$, the bunched transition system is maximal for P'.
Let P and Q be two recursively maximal processes. Then $P \sim_f Q$ if and only if $P \sim Q$.

Proof: We only have to prove that if $P \sim Q$ then $P \sim_f Q$.

Let Q be a process such that $P \sim Q$. Suppose that $P \stackrel{L}{\Longrightarrow} P'$, where $L = l_1 \ldots l_n$. By reciprocity, $P \stackrel{l_1}{\rightarrow} \ldots \stackrel{l_n}{\rightarrow} P'$ and, since Q is bisimilar to P, then also $Q \stackrel{l_1}{\rightarrow} \ldots \stackrel{l_n}{\rightarrow} Q'$, with $P' \sim Q'$.

By hypothesis, since $P \stackrel{L}{\Longrightarrow} P'$, if $P' \stackrel{l}{\rightarrow} R$ and $P' \stackrel{l'}{\rightarrow} S$, then there is no T such that $R \stackrel{l'}{\rightarrow} T$ and $S \stackrel{l'}{\rightarrow} T$. Since P' and Q' are bisimilar, this also stands for Q'.

Again, by hypothesis, this implies that $Q \stackrel{L}{\Longrightarrow} Q'$. Since P and Q play the same role in the hypothesis, and since the two processes are recursively maximal, we conclude. ■

This result states that a bunched transition system for which all the observed transitions are reaching a state where all derivations are not confluent would induce a bisimilarity that coincide with the original bisimilarity. Such a bunched bisimilarity system may then be considered as maximal.

However, this hypothesis cannot be easily granted in general. For instance, the processes $!(x(y).x(y))$ and $!x(y)$ do not have any non confluent continuations. In this case, the bunched system may operate infinitely many transitions, without reaching a point where two non confluent transitions are possible. As a consequence, a maximal bunched transition system should then take into account such processes. A possible solution could be to limit the depth of the bunched transitions to a given value.

Another interesting intuition that arises from this remark is that, in the π_a-calculus, if $P \stackrel{\alpha}{\rightarrow} P'$ and $P \stackrel{\beta}{\rightarrow} P''$ such that the two do not confluate, then $\alpha = \beta = \tau$ and this corresponds to an internal reception on a private channel in the

process P. Then, it can be noticed that in the asynchronous languages, only the internal choices are non-deterministic. Hence, the bunched evaluation should stop only in this case.

As we will see in the next part, it is indeed possible to give a fully deterministic probabilistic semantics to an asynchronous calculus whose only non-deterministic transitions are internal choices, provided we define a probability distribution for the choices.

4.3.3 Bunched evaluation strategy and schedulers

Another way of understanding the bunched transition system could be to formalize it as a focusing function $f(x) \in \{0; 1\}$. Then, upon trying to find a bunched transition for a process P, we would first select one of the possible transitions, such that $P \xrightarrow{l} P'$. We would then apply the function f to P' such that, if $f(P') = 1$, we can select any of the transitions of P' and continue this operation.

We would then require that when repeating this operation, the function f would return 0 at some point. Then, the sequence of transitions we have just executed would be a bunched transition of the process P. For instance, the intuition behind the bunched transition system for the π_a-calculus is: $f(P) = 1$ if and only if $P \equiv x(y).P'$.

This formalization would then be a very interesting way of implementing a program executing and testing processes. Indeed, when computing the transitions and processes of a process, the implementation would need to be able to compute the original transitions and the result of the function f applied to a process. The evaluation would then immediately be optimized without wasting the required informations.

This work has been initiated after a first try in implementing an asynchronous evaluator for the π_a-calculus that would also keep track of the specific scheduler that was used. Indeed, in the case of the π_a-calculus, due to all possible interleavings, the number of schedulers to consider was exploding, leading to a very slow evaluation, even for simple processes. Using the results of this chapter, it is then possible to reduce greatly the number of schedulers to consider, without loosing the information we want to observe.

Part III

Asynchronous probabilistic executions

Introduction

The need to deal with probabilistic phenomena arises in many fields in Computer Science. For instance, a large part of the recent research on security and trust is oriented toward understanding and formalizing probabilistic properties.

The formal treatment of probabilistic phenomena is particularly difficult in the presence of concurrency, as concurrency itself presents many complicated aspects. Among the many formalisms which have been developed for concurrency, Concurrent Constraint Programming (CCP, [Sar89]) is a very elegant one. CCP is a model of computation in which the information available to the process is represented by the notion of constraint. Each process has access to a global store, with respect to which it tests and adds constraints. In [SRP91b], a denotational semantics is defined for the processes in CCP. In this semantics, a process is represented by the input/output relation that the process computes. In the case of CCP, that relation belongs in a particular class of functions called the *closure operators*. These functions can be represented by the set of their fixed points. Hence, a process in this semantics will be represented by its set of fixed points. This set can be computed using a fixed-point construction. The denotational semantics for CCP is a very elegant model for reasoning in a particularly simple and modular way.

Probabilistic extentions of CCP have been explored in several works. In the probabilistic CCP, defined by Vineet Gupta et al. in [GJS97, GJP99], the probability is defined at the level of the constraint system and inconsistent executions are removed at run-time. This models the probabilistic choice as an external process independent from the program. During an execution, the program waits until a probabilistic choice has been drawn, and proceed according to the outcome. When the external choice was inconsistent, the program should detect it and avoid any further computation. In [Pie00, PW98], the CCP is extended with a probabilistic construct on the process' syntax. It models probabilistic executions as the choice of the program. Hence, in this case, probabilistic choices that lead to an inconsistent store should be considered as a failure from the program and as such should not be hidden during the program's execution. In this work, we follow the approach used in [Pie00, PW98] and we address the problem of giving a denotational semantics for infinite executions, which were treated neither in [Pie00, PW98] nor in [GJS97, GJP99]. Also, we extend the lattice of constraints to any well-ordered lattice, whereas in [Pie00, PW98] the lattice had to be finite.

When studying probabilistic phenomena, it is often very important to consider infinite succession of events. Indeed, when observing an infinite sequence of events, one can be in a situation where the probability of a given outcome reaches 0 as the limit of the infinite execution. For instance, in the Crowds routing protocol, as defined in Section 5.3, the probability to not deliver a message is the result of an infinite execution and has a null probability. Hence, it can be stated that a message will always be delivered.

But the difficulty that arise when one wants to define the outcome of an infinite execution is that this outcome should again represent the probabilities associated to the original deterministic states. A natural requirement is that if $p_i(S)$ is the probability of being in the state S in the i^{th} execution step, then the probability of being on the state S after an infinite execution is $\lim_{i\to\infty} p_i(S)$. Furthermore, probabilities measure should be additive, meaning that the probability to observe a set of deterministic events is the sum of the probabilities to observe each individual events. Without any further hypothesis, none of these requirements are met in general. In particular, an infinite sum of limits is not in general to the limit of the infinite sums, i.e. $\sum_{s\in S} \lim_{n\to\infty} p_n(s) \neq \lim_{n\to\infty} \sum_{s\in S} p_n(s)$.

We address these issues using a relevant mathematic framework. We use a topological notion of probabilities called the valuations, which forms a domain usually referred as the Jones powerdomain. This domain has been studied extensively in [Jon90].

We prove that the probabilistic states of a program in our language can be represented by a subset of valuations called the simple valuations, where the measure of a set of states is exactly the sum of the measure of the individual states. We prove that the outcome of an infinite run in our language can also be represented by a valuation. We then characterize some conditions under which all the valuations are simple valuations, which implies that the valuations representing the infinite executions are also a simple valuations.

Eventually, we observe that the input/output relation induced by a process in our language, mapping an initial valuation representing the initial probabilistic state to the valuation representing the (possibly infinite) execution of the process started with this valuation, is a linear closure operator on the vector spaces of valuations. We then define a denotational semantics using a fixed point construction, and prove the correctness with the input/output relation.

Another contribution of this work is the definition of an asynchronous concurrent language with an internal choice where executions are deterministic. Indeed, under asynchronous communications, the only non-determinism is introduced by the internal choice. Hence, when adding probabilities for each branches of each choice, the result becomes deterministic in terms of probabilities. This has important algorithmic properties, since we can ignore some possible computations whose probability is 0. In particular, this allows implementing protocols that are known to be impossible to implement in non probabilistic asynchronous languages, such as the leader election.

Compared to real life communication, this formalizes for instance the following idea: when trying to decide for a meeting agenda, instead of proposing any of his possible dates, each agent decides on the basis of others' answer, reducing its behavior to a probabilistic choice, where the data which matches the greatest number of agents has the greatest probability.

Chapter 5

Probabilistic Concurrent Constraint Programming.

We define in this chapter a probabilistic concurrent constraint language (CCP+P). The language is an extension of the original CCP defined in [Sar89, SRP91b]. The interested reader may find there detailed definitions and properties on which this language is based.

In this language, each program is equipped with a constraint store, which is used to put new constraints and test if one constraint can be entailed by it. We start be defining the constraint system. We then define the syntax of the processes, as well as the operational semantics. We finish with an example of application as a model of the crowds anonymous routing protocol.

This chapter introduces the basics of the language, mainly syntax, operational semantics and observables. These definitions are used in the next chapter to define the denotational semantics of the language and prove the correctness with regard to the operational semantics.

Contents

5.1 The Constraint System 82
5.1.1 Simple constraint system 82
5.1.2 Cylindric constraint system 82
5.2 The language 83
5.2.1 Syntax 83
5.2.2 Operational semantics 84
5.3 Application: Crowds anonymous routing 86
5.3.1 Presentation 86
5.3.2 Implementation and infinite runs 87

5.1 The Constraint System

In the following, if X is a set, then $\mathcal{P}_{\text{fin}}(X)$ is the set of finite subsets of X.

5.1.1 Simple constraint system

Definition 5.1.1 (*Simple Constraint System*)

A simple constraint system is a structure $\langle D, \vdash \rangle$, where D is a non-empty (countable) set of tokens or (primitive) constraints and $\vdash \subseteq \mathcal{P}_{fin}(D) \times D$ is an entailment relation satisfying:

1. *$P \vdash u$ if and only if $u \in P$*
2. *$Q \vdash v$ if and only if $P \vdash v$ and $\forall u \in P$, $Q \vdash u$*

$\vdash$ is extended to $\mathcal{P}_{fin}(D) \times \mathcal{P}_{fin}(D)$ by: $u \vdash v$ iff $u \vdash c$, for all $c \in v$.
We define $u \equiv v$ iff $u \vdash v$ and $v \vdash u$.

An equivalent representation of the constraints is a lattice equipped with the order: $x \leq y$ iff $y \vdash x$ and $c \sqcup d = c \wedge d$.

For two finite sets of primitive constraints $c, d \in \mathcal{P}_{\text{fin}}(D)$, $c \wedge d$ represents the logical conjunction of the two sets, i.e. $c \wedge d = c \sqcup d$. The $\top$ constraint represents the logical *false*: for any constraints c, $\top \vdash c$. $\bot$ corresponds to the logical *true*: $c \vdash \bot \Rightarrow c = \bot$.

5.1.2 Cylindric constraint system

Definition 5.1.2 (*Cylindric constraint system*)

A cylindric constraint system is a structure $\langle D, \vdash, \texttt{Var}, \{\exists_X \mid X \in \texttt{Var}\} \rangle$ such that:

- *$\langle D, \vdash \rangle$ is a simple constraint system*
- *$\texttt{Var}$ is an infinite set of indeterminates or variables*
- *for each variable $X \in \texttt{Var}$, $\exists_X : \mathcal{P}_{fin}(D) \to \mathcal{P}_{fin}(D)$ is an operation satisfying:*
  1. *$u \vdash \exists_X u$,*
  2. *$u \vdash v$ implies $\exists_X u \vdash \exists_X v$,*
  3. *$\exists_X(u \wedge \exists_X v) \equiv \exists_X u \wedge \exists_X v$,and*
  4. *$\exists_X \exists_Y u \equiv \exists_Y \exists_X u$*

We also add diagonal constraints of the form γ_{XY} which are used for parameter parsing, with the following axioms:

Definition 5.1.3 (*Diagonal elements*)

Diagonal elements are constraints of the form γ_{XY} where X and Y are variables. They satisfy the following axioms:

1. $\emptyset \vdash \gamma_{XX}$
2. *if* $X \neq Y$ *then* $\gamma_{XY} \equiv \exists_Z(\gamma_{XZ} \wedge \gamma_{ZY})$
3. $\gamma_{XY} \wedge \exists_X(u \wedge \gamma_{XY}) \vdash u$

5.2 The language

In the following, we may omit, when clear from the context, the sets over which we index families, sequences, sums and unions. These sets are not assumed to be limited to the integers (or ω). The sums over positive reals with indexes greater than the integers are extended by transfinite induction[1].

5.2.1 Syntax

The grammar of the language is as follow:

Definition 5.2.1

The CCP+P language is defined by the following grammar, where the c's and c_i's represent constraints.

$$\begin{array}{lllll} \textit{Agent} & A & ::= & 0 \mid c \mid c \to A \mid A \mid A \mid \exists X.(c, A) \mid \oplus_i(A_i, c_i, p_i) \mid p(X) \\ \textit{Procedure} & D & ::= & \epsilon \mid p(X) :: A \mid D.D \\ \textit{Process} & P & ::= & D.A \end{array}$$

We also require the following:

The guard c in $c \to A$ is finitely algebraic.[2]

All constraints on a probabilistic choice are mutually exclusive pure constraints:

$$P = \oplus_i(A_i, c_i, p_i) \Rightarrow \begin{cases} \forall i \neq j,\ c_i \wedge c_j \vdash \top \\ \exists_X c_i \vdash c_i, \textit{ for all } X \end{cases}$$

The p_is in the above agent are positive reals whose sum is 1.

The *ask* agent c waits for adding the constraint c to the store. If the current store can entail the constraint c, then the *tell* agent $c \to A$ is replaced by the agent A. Otherwise, it does nothing. The algebraic requirement on guards for the $c \to A$ construction will be usefull later. Without this, the programs would

[1] We define the sum of a family of positive real values $(x_i)_{i \in I}$ as the least upper bound of the partial finite sums, that is of the number $\sum_{i \in J}$ where J traverses the finite subsets of I.

[2] For any directed sequence (x_i) such that $\sup_i x_i = x$, there exists a finite i_0 such that $x_{i_0} = x$.

not be idempotent in general. $A \,|\, B$ is the parallel concurrent execution of A and B. Direct communication between A and B is not possible. Communication can only happen through the constraint store, using *tell* and *ask*. The agent $\exists X.(c, P)$ proceeds like P, but considers X as a private variable and c is a private constraint that only this process knows. Hence, the operational semantics takes care of removing any external information on X before computing the possible transitions of P under the current store. Similarly, when a transition has been executed, it removes from the global store information related to X and keeps it private for the process. $p(X) :: A$ stands for the procedure definition. When a program reaches $p(Y)$, it can then start the procedure $A[X/Y]$, allowing recursion and infinite behavior.

All these constructs are present in the original CCP. The additional operator in the CCP+P is the choice operator $\oplus$. It corresponds to a guarded blind choice. When executed at run-time, if none of the branch constraints can be entailed, it selects one of the possible branches, adds the corresponding constraint to the global store and continue with the corresponding agent. If one of the branch constraints can be entailed, then it follows the execution with the corresponding agent. Since the branch constraints in the choice are all mutually exclusives, the behavior in this case is always deterministic.

The guards are used to keep track of resolved choice. From the language point of view, it means that each probabilistic choice is resolved once and for all. Hence, each resolved choice can be retrieved from the current store and is not drawn again if the program is restarted with the same store.

We now introduce the operational semantics, starting with the notion of configurations:

Definition 5.2.2

A CCP+P configuration is a set of elements of the form $C = \cup_i\{(P_i, c_i, p_i)\}$ *where the* P_i *are processes, the* c_i *constraints, and the* p_i *positives reals such that* $\sum_i p_i = 1$.

From a configuration in the CCP+P, we can extract a configuration on the constaints lattice:

Definition 5.2.3

The projection of a configuration $C = \cup\{(P_i, c_i, p_i)\}$ *over the constraint lattice is the set:* $\Pi(C) = \{(c, \sum_{(P_i, c_i = c, p_i)} p_i)\}$.

5.2.2 Operational semantics

We define an operational semantics for the language in terms of transitions between configurations. We first introduce the possible transitions for a singleton, and extend them later.

Definition 5.2.4

The transition relation for CCP+P is defined by the following inductive rules:

$$(tell)\quad \frac{}{\{(c,d,1)\} \to \{(0, d \wedge c, 1)\}} \qquad (ask)\quad \frac{d \vdash c}{\{(c \to P, d, 1)\} \to \{(P, d, 1)\}}$$

$$\frac{\{(A,d,1)\} \to \cup_i \{(A_i, d_i, x_i)\}}{\{(A|B, d, 1)\} \to \cup_i \{(A_i|B, d_i, x_i)\}} \qquad \frac{\{(B,d,1)\} \to \cup_i \{(B_i, d_i, x_i)\}}{\{(A|B, d, 1)\} \to \cup_i \{(A|B_i, d_i, x_i)\}}$$

$$(choice1)\qquad \frac{c \vdash c_i}{\{(\oplus_i (A_i, c_i, p_i), c, 1)\} \to \{(A_i, c, 1)\}}$$

$$(choice2)\qquad \frac{c \nvdash c_i, \quad \forall i}{\{(\oplus_i (A_i, c_i, p_i), c, 1)\} \to \cup_i \{(A_i, c \wedge c_i, p_i)\}}$$

$$\frac{\{(P, c \wedge \exists_X d, 1)\} \to \cup_i \{(P_i, d_i, x_i)\}}{\{(\exists X.(c, P), d, 1)\} \to \cup_i \{\exists X.(d_i, P_i), d \wedge \exists_X d_i, x_i)\}}$$

$$\frac{p(X) := A}{\{(p(Y), c, 1)\} \to \{(A[Y/X], c, 1)\}}$$

$$\frac{(linext) \quad \{(P_{i_0}, d_{i_0}, p_{i_0})\} \to \cup_j \{(P'_j, d'_j, a'_j)\}}{\cup_i \{(P_i, d_i, p_i)\} \to \bigcup_{\substack{(P_i,d_i) \neq (P'_j, d'_j) \\ i \neq i_0}} \{(P_i, d_i, p_i)\} \cup \bigcup_j \{(P'_j, d'_j, p_{i_0} a'_j + \sum_{(P_i,d_i)=(P'_j,d'_j)} p_i)\}}$$

The two operational rules (*choice*1) and (*choice*2) for the probabilistic choice are quite straight forward. (*choice*1) states that if the current store can entail one of the branch constraints, meaning that the choice was already done, then the process continues on this branch without changing the current probabilistic state. Otherwise, (*choice*2) states that it proceeds with each choice, adding constraints to each store in order to keep track of the resolved choices.

The rule (*linext*) allows to extend operational rules based on singletons to any configuration. It states that when an atomic transition is possible from a set of states with probabilities, then the whole set can make the same transition, where the new states and probabilities are obtained by linearity.

Other rules are simply lifted from the original operational semantics for the CCP. For instance, $\{(c \to P, d, p)\}$ may evolve to $\{(P, d, p)\}$ iff $d \vdash c$, just as for CCP.

This language extends the probabilistic CCP defined in [PW98]. Indeed, the grammar and syntax are very similar, in particular the choice operator. However, the constraints lattice has to be finite in [PW98]. We will prove later that in our case, the constraint lattice has to be countable.

Remark 5.2.1

If $\mathcal{C} = \cup_i \{(P_i, c_i, d_i)\}$ is a configuration, and $\mathcal{C}' = \cup_j \{(P'_j, c'_j, d'_j)\}$ a configuration derived from $\mathcal{C}$ using the operational semantics, written $\mathcal{C} \longrightarrow \mathcal{C}'$, then:

$$\sum_i p_i = \sum_j p'_j$$

This property ensures that, when started with a bounded (resp. normed) set of probabilities, it remains bounded (resp. normed) along the execution.

5.3 Application: Crowds anonymous routing

5.3.1 Presentation

We give here an example of a CCP+P program. This program will implement the crowds routing protocol, as defined in [RR99]. Crowds is a probabilistic anonymity protocol. It features a pool of agents, called the forwarders. When an agent wants to send a message to another agent, it sends the message to one forwarder of the pool. Then, the forwarder decides with probability p to deliver the message to its destination, or with probability $1 - p$ to forward it again to another node in the pool. This is illustrated in Figure 5.1.

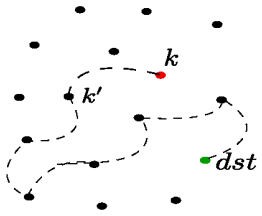


Figure 5.1: The Crowds anonymous routing protocol

At each execution, the forwarder remembers the choice he drew. Hence, when the receiver wants to reply, he can send his message to the last forwarder, and it will be delivered to the originator using the same path. The protocol is correct in the sense that the probability of that the message does not reach its destination is 0.

This protocol can be implemented in the CCP+P. For simplicity, we will only implement the sending part, but not the answer one, where the receiver replies to the sender. However, using the choice constraints, one can remark that each resolved choice is recorded and may be reused later for finding back the answer path.

The constraint system that we consider is made of two predicates. We model the execution as a sequence of messages. The first predicate states that at position i in the sequence, a message with destination *dest* needed to be deliverered to

forwarder id. This predicates is written $elem(i) = (id, dst)$. We also define a per-position choice constraints. The predicates means that the message at position pos was examined by agent id, and its value is the chosen forwarder. It is written $chosen(id, pos) = dst$. This information can be made private using for instance a custom hash of the receiver instead of the receiver, and a local per-agent variable X for which we would add the constraints $(X = hash(dst)) \wedge chosen(id, pos) = X$). Again, this is not detailled here for simplicity and concision.

We consider that sender and receiver are agents of the pool. Hence, the algorithm will stop as soon as the selected forwarder is the receiver. We also consider a uniform probability distribution for the next selected hop. n stands for the cardinality of the pool.

5.3.2 Implementation and infinite runs

A simple implementation of one agent is proposed in Algorithm 1. The implementation works as follow: the messages are sequentialized as a stream. Each agent starts with the first message in the stream. It examines it, and if it is not for him, then proceed with the message at next position. If it is for him, either he is the final receiver, and the program stops, or he selects a new node and add a new constraint representing the new message. The tricky part is at line 3. In this sub-process, the variables i and dst are instantiated as private variable. However, since the process tells to the store $elem(pos) = (i, dst)$ then by unification these private variables' values are necessary those of the public ones. This how values are usually retrieved in CCP.

The algorithm is then started by the instantiation of n parallel agents all waiting on the position 0 on the message stream, equipped with the store $\{elem(0) = (k, k')\}$, for some positive integers k,$k' \leq n$.

Algorithm 1 The Agent process

Agent(id,pos**)** :=
1: $\exists_{i,dst} elem(pos) = (i, dst) \rightarrow$
2: $\exists_{i,dst}$
3: $\mid elem(pos) = (i, dst)$
4: $\mid (id = i) \rightarrow$
5: $\mid (id = dst) \rightarrow 0$
6: $\mid (id \neq dst) \rightarrow$
7: $\mid \oplus_j (elem(pos + 1) = (j, dst), chosen(id, pos) = j, \frac{1}{n})$
8: $\mid$ **Agent(**id,$pos + 1$**)**
9: $\mid (id \neq i) \rightarrow$ **Agent(**id,$pos + 1$**)**

Important studies exist in the literature about the execution of this routing algorithm. In particular, the outcome of an infinite execution is well studied from a security point of view, when the choice for next hop is biased (see [CPP06]

for instance). In the case of this implementation, one may want to study the various configurations that are entailed by the operational semantics and try to retrieve the outcome of an infinite execution. However, this is not really possible in this language with only the operational semantics since the result of an infinite computation cannot be defined.

However, in the case of this implementation, the result of an infinite execution can be intuited quite easily. Indeed, the projection of the initial state is: $\{(elem(0) = (k, k'), 1)\}$. Then, after the next execution step, if $c_{l,m,n} = chosen(l, m) = n \wedge (elem(l + 1) = (m, k')$, the outcome will be:

$$\bigcup \begin{cases} \{(elem(0) = (k, k') \wedge c_{0,k,k'}), \frac{1}{n})\} \\ \cup_{j \neq k'} \{(elem(0) = (k, k') \wedge c_{0,k,j}), \frac{1}{n})\} \end{cases}$$

And then after another step:

$$\bigcup \begin{cases} \{(elem(0) = (k, k') \wedge c_{0,k,k'}, \frac{1}{n})\} \\ \bigcup_{j \neq k'} \{(elem(0) = (k, k') \wedge c_{0,k,j} \wedge c_{1,j,k'}), \frac{1}{n^2})\} \\ \bigcup_{j \neq k'} \bigcup_{i \neq k'} \{(elem(0) = (k, k') \wedge c_{0,k,j} \wedge c_{1,j,i}), \frac{1}{n^2})\} \end{cases}$$

And, after the next step:

$$\bigcup \begin{cases} \{(elem(0) = (k, k') \wedge c_{0,k,k'}, \frac{1}{n})\} \\ \bigcup_{j \neq k'} \{(elem(0) = (k, k') \wedge c_{0,k,j}) \wedge c_{1,j,k'}), \frac{1}{n^2})\} \\ \bigcup_{j \neq k'} \bigcup_{i \neq k'} \{(elem(0) = (k, k') \wedge c_{0,k,j} \wedge c_{1,j,i} \wedge c_{2,i,k'}), \frac{1}{n^3})\} \\ \bigcup_{j \neq k'} \bigcup_{i \neq k'} \bigcup_{l \neq k'} \{(elem(0) = (k, k') \wedge c_{0,k,j} \wedge c_{1,j,i} \wedge c_{2,i,l}), \frac{1}{n^3})\} \end{cases}$$

Hence, the infinite execution leads to the following distribution, where we have removed the infinite sequences, since the associated probability converges to zero. First, let (e_n) be a sequence of identifiers, $(e_n) \in \mathbf{F}_l$ if and only if (e_n) has $l+1$ elements such that for any $i \neq l$, $e_i \neq k'$ and $e_l = k'$. Then, the probability distribution resulting of an infinite execution can be defined as:

$$E = \bigcup_{l \in \mathbb{N}} \bigcup_{(e_n) \in \mathbf{F}_l} \{(elem(0) = (k, k') \bigwedge_{i=0}^{l-1} c_{i,e_i,e_{i+1}}, \frac{1}{n^{l+1}})\}$$

Of course, we need to prove that the sum of the probabilities in E is one. Since the cardinality of $\mathbf{F}_l$ is $(n-1)^l$, we can prove that:

$$\sum_{l=0}^{x} \sum_{(e_n) \in \mathbf{F}_l} \frac{1}{n^{l+1}} = \sum_{l=0}^{x} \frac{(n-1)^l}{n^{l+1}} = \sum_{l=0}^{x} \frac{n(n-1)^l - (n-1)^{l+1}}{n^{l+1}}$$

Hence, $\sum_{l=0}^{x} \sum_{(e_n) \in \mathbf{F}_l} \frac{1}{n^{l+1}} = \sum_{l=0}^{x} (\frac{n-1}{n})^l - (\frac{n-1}{n})^{l+1}$ and we recognize an alternated sum such that:

$$\sum_{l=0}^{x} \sum_{(e_n) \in \mathbf{F}_l} \frac{1}{n^{l+1}} = 1 - (\frac{n-1}{n})^{x+1} \longrightarrow_{x \to \infty} 1.$$

We have then proved that $\sum_{(c,p) \in E} p = 1$

In order to generalize this and properly define the result of an infinite computation for the CCP+P,we introduce a denotation semantics, that represents the program as a function. This is the purpose of the next chapter.

Chapter 6

Denotational semantics for the CCP+P

In this chapter, we give a denotational semantics for the CCP+P. Along the way, we also prove a very general result about the possibility to decompose a valuation. This results allows many powerful applications to semantics and limit of infinite probabilistic executions.

The result in this chapter appeared in [Bea09] and have been submitted for revision to the Journal of Theory and Practice of Logical Programming.

Contents

6.1 Valuations 92
6.1.1 Sober spaces 94
6.1.2 Well-founded orders and lattices 95
6.1.3 Decomposition of valuations 98
6.2 Vector cones of simple valuations 103
6.2.1 Vector cones 103
6.2.2 Linear closure operators 104
6.2.3 Observables for the CCP+P 105
6.3 Denotational semantics 108

Introduction

In this chapter, we give a denotational semantics for the CCP+P. This semantics allows to characterize the processes by a function mapping the initial input to the (possibly infinite) execution it can generate. The interesting consequence is that the (fair) execution of a program in the CCP+P is probabilistically deterministic.

For this semantics, we want to define the maximal probabilistic state as for any finite run, which means of the form $\cup_i\{(c_i, p_i)\}$. We achieve that by mapping each finite execution to a measure function for the opens of the Scott topology, namely a valuation. More precisely, we map the finite executions to a particular type of valuation called simple valuation, of the form $\sum_i p_i\delta_{x_i}$.

We then prove that the sequence of finite executions is directed and reaches a limit. We then introduce a very general result on the structure of the image of the valuation to characterize under which conditions this limit can be also decomposed in the form of a simple valuation $\sum_j p'_j\delta_{c'_j}$. Hence, we define the result of an infinite run as $\cup_j\{(c'_j, p'_j)\}$.

Finally, we also characterize this limit as the image of a closure operator for which we give a recursive definition of its fixed points. The defined semantics is then proved to be sound and fully abstract.

6.1 Valuations

In order to define a denotational semantics, one has to decide which kind of object the denotational meaning of a program will be. Here, we chose to consider programs as mappings from valuations to valuations[1]. Valuations are probability measures for topological spaces. They measure the probability of an open set of atomic states, much like the wave function in quantum mechanics. The execution of a program against an initial probabilistic state returns a new probabilistic state.

Simple constraint valuations are measures that can be decomposed as individual probabilities over constraints. This property is very important in order to achieve some consistency for the result of an infinite computation. Hence, we want the limit state of an execution to be represented as a simple valuations, where each individual state has a probability and the probability of a set of individual states is the sum of their probabilities.

In this section, we study the conditions under which all valuations are simple valuations. An infinite directed sequence of valuations will then have a simple valuation as its limit. That way, every program will be interpreted as a

[1]This approach was inspired by the work done in [Koz81]

linear mapping from simple valuations to simple valuations, even when running an infinite sequence of operations, allowing a denotational semantics to be defined.

The mathematic results on the decomposition of valuations based on the image of the function is new. It implies the important result that any valuation on a totally well-ordered lattice is simple.

In the second section, we define the mathematical structure needed for defining the denotational semantics of the language. It mainly extends some definitions from lattices to vector cones, including the notion of closure operator, namely linear closure operators, will then be the domain of the denotational semantics for the probabilistic CCP.

In the following, we use the Axiom of Dependent Choice. It can be stated as follows: for any nonempty set X and any entire binary relation[2] $\mathcal{R}$ on X, there is a sequence (x_n) in X such that $x_n \mathcal{R} x_{n+1}$ for each n.

Using this constructive axiom, it is possible to build an infinite sequence of elements x_i such that for any i, $x_i \mathcal{R} x_{i+1}$. It means that one can do an infinite sequence of choices. Since each choice depends on the previous one, it is called a dependent choice. It implies the Axiom of Choice over a countable family of sets.

This axiom allows to define the theory using directed sequences instead of directed sets. Indeed, the domain theory can be stated using directed sets, which are subsets where the order is total, or using directed sequences and famillies. It can be proved that the two theories are equivalent under this axiom.

Indexes on directed sequences can be any ordinal. However, the results also hold when replacing transfinite assumptions with only infinite (ω) assumptions.

In the following, we will use directed sequence or sets without any distinction. A directed sequence (x_i) is total if and only if for any x_i, x_j and any x such that $x_i \leq x \leq x_j$, then x is also an element of the sequence.

Definition 6.1.1

Let (X, τ) be a topological space where τ is the set of open sets. A valuation *on X is a function: $\mu : \tau \to \mathbb{R}^+$ with the properties:*

1. $\mu(\emptyset) = 0$
2. $\mu(O) + \mu(U) = \mu(O \cup U) + \mu(O \cap U)$
3. $O \subseteq U \implies \mu(O) \leq \mu(U)$
4. $\mu(X) < +\infty$

We write $\mathcal{V}al(X, \tau)$ for the set of all possible valuations on X. When clear from the context, τ will be omitted.

[2] In this document, an entire binary relation is relation $\mathcal{R}$ such that for any x, there exists at least one y such that $x \mathcal{R} y$.

Definition 6.1.2 (*Simple Valuation*)

The function:

$$\delta_x : O \mapsto \begin{cases} 1, \text{ if } x \in O \\ 0, \text{ otherwise} \end{cases}$$

is a valuation, called Dirac valuation.
A simple (resp. finite) valuation is a countable (resp. finite) linear combination of elementary Dirac valuations:

$$v = \sum_{i \in I} a_i \delta_{x_i}$$

Where the a_i are positive reals such that: $\sum_i a_i < +\infty$.

We define the following order on valuations:

Definition 6.1.3

Let μ and ν be two valuations.

$$\mu \leq \nu \iff \forall U, \mu(U) \leq \nu(U)$$

6.1.1 Sober spaces

We also introduce the topological notion of sober space. A sober space is a topological space where the prime filters characterize the elements of the space. This is very interesting since it allows to identify a point based on a prime filter. We will use this notion to define the elementary decomposition of a valuation to a simple valuation.

Definition 6.1.4

Let (X, τ) be a topological space. A filter on X is a set F of open sets of X such that:

- *F is upward closed: $\forall O \in F$, if $O \subseteq O'$ then $O' \in F$.*
- *F is stable by finite intersections: $\forall O, O' \in F$, $O \cap O' \in F$*

We say that a filter is prime if and only if for any family of open sets $(U_i)_{i \in I}$, we have:

$$\cup_{i \in I} U_i \in F \Rightarrow \exists j \in I \text{ s.t. } U_j \in F$$

The canonical example is the set of open sets containing a point:

Example : Let x be an element of a set X and $F_x = \{O \in \tau \mid x \in O\}$, then F_x is a prime filter.

Proof : • If $x \in O$ and $O \subset O'$ then $x \in O'$

- If $x \in O$ and $x \in O'$ then $x \in O \cap O'$
- If $x \in \cup_{i \in I} O_i$ then $x \in O_{i_0}$ for at least one i_0. ■

A particular application of prime filters is the notion of sober space. For these spaces, all prime filters of open sets are of the form $F_x = \{O \in \tau \mid x \in O\}$.

Definition 6.1.5

Let (X, τ) be a topological space. We say that (X, τ) is sober if and only if for any prime filter F, there exists a point x such that $F = \{O \in \tau \mid x \in O\}$.

Examples of sober spaces are:

- $\mathbb{N} \cup \{+\infty\}$ equipped with the Scott-topology
- Any Hausdorff/T_2 space
- Any domain

This property is very useful in conjunction with valuations: one can retrieve the elements and coefficients of a simple valuation by looking at the values of the function on the open sets.

6.1.2 Well-founded orders and lattices

We will use the notion of well-founded order:

Definition 6.1.6

Let X be a set and $\mathcal{R}$ a binary relation on X. $\mathcal{R}$ is well-founded if there does not exist infinite decreasing sequences, i.e. sequences $(x_i)_{i \in \mathbb{N}}$ such that for any $i \in \mathbb{N}$, $x_{i+1} \mathcal{R} x_i$.

By contradiction, this can also be characterized as:

Proposition 6.1.1

Let X be a set and $\mathcal{R}$ a binary relation on X.
$\mathcal{R}$ is well-founded if and only if for any sequence $(x_i)_{i \in I}$ of elements of X such that $x_{i+1} \mathcal{R} x_i$, then I is a finite set of indexes.

A well-ordered set is a set equipped with an order whose corresponding strict order is well-founded.

Well-ordered sets are those on which transfinite recursion can be proved[3]:

[3]This theorem can also be used to define well-founded orders: a well-founded order is then an order for which all points are recursively enumerable under a transfinite recursion.

Theorem 6.1.1

Let X be a well-ordered set, and $P(\alpha)$ a predicate where $\alpha \in X$.
If when for any $z < x$, $P(z)$ is true then $P(x)$ true,
then $P(x)$ is true for any $x \in X$.

Proof: We prove the result by contradiction. Suppose that $P(x_0)$ is false for a particular x_0, then there must exist a $x_1 < x_0$ such that $P(x_1)$ is not true.
By repeating this sequence, we obtain a sequence of decreasing points x_i for which $P(x_i)$ is false. By assumption, this sequence must reach a minimal element x of the set X.
However, since there is no z such that $z < x$, $P(x)$ is true and we have a contradiction. ■

We give now some interesting results on a special class of well-ordered lattices. In the following, a complete lattice is *meet-continous* if the meet operation is continous, i.e. if x_i is a directed sequence such that $x = \sup_i x_i$, then for any y, $\inf(y, x_i)$ is a directed sequence and $\sup_i(\inf(y, x_i)) = \inf(y, x)$.

Definition 6.1.7

A countable complete meet-continous lattice X, is totally well-ordered, written WOL, if and only if there exists a directed sequence (x_i) such that:

- *(x_i) is well-ordered*
- *$\cup_i\{x_i\} = X$*
- *For any algebraic element x, the index of x in the sequence is finite.*

Such a sequence can be seen as a consistent well-ordered extension of the order on the lattice to a total order. It is illustrated in Figure 6.1.

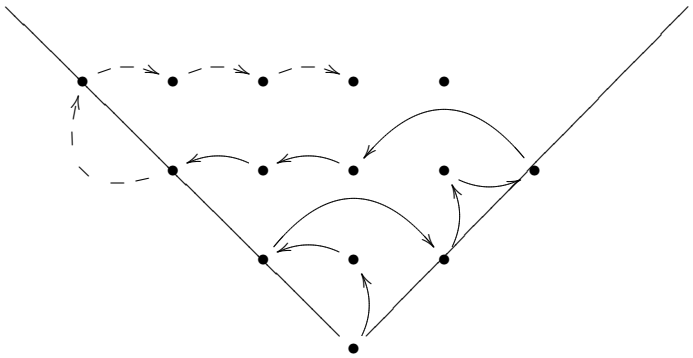


Figure 6.1: An example of a WOL

Hence, we use WOL for the class of countable well-ordered lattices, since they coincide in our case. Now, we prove that any *well-ordered lattice* is sober.

First we introduce the following propositions:

Proposition 6.1.2

Let X be a WOL, and $S \subset X$ such that for any $x \in S$, if $x \leq y$ then $y \in S$.
Then $S = \cup_{x \in \downarrow S} \uparrow x$,
where $\downarrow S = \{x \in S \mid \forall y \in S,\ y \leq x \Rightarrow y = x\}$, and $\uparrow x = \{y \mid x \leq y\}$.

Proof: First, $\cup_{x \in \downarrow S} \subset S$, so we need to prove the converse.
Let $x \in S$. If $x \in \downarrow S$, then $x \in \cup_{x \in \downarrow S} \uparrow x$.
Otherwise, there exists $x_1 \in S$ such that $x_1 \leq x$.
Since the lattice is well-ordered, by repeating this construction, we find a $x_k \in \downarrow S$ such that $x_k \leq x$. Hence, $x \in \cup_{x \in \downarrow S} \uparrow x$. ■

In the following, an element x is algebraic if and only if for any directed sequence (x_i) such that $\sup_i x_i = x$, then there exists i_0 such that $x_{i_0} = x$.

Proposition 6.1.3

For any Scott-open O on X, $O = \cup_{x \in \downarrow O} \uparrow x$, and x is algebraic for any $x \in \downarrow O$.
Reciprocally, for any algebraic element x, $\uparrow x$ is an open.

Proof: For any $x \in \downarrow O$ and any directed sequence x_i such that $x = \sup_i x_i$, since $x \in O$ there exists i_0 such that $x_{i_0} \in O$. Since $x \in \downarrow O$, $x_{i_0} = x$.
Let x_i be a directed sequence such that $\sup_i x_i \in \uparrow x$. Then $\inf(x_i, x)$ is a directed sequence such that $\sup_i \inf(x_i, x) = x$. Hence there exists i_0 such that $\inf(x_{i_0}, x)$, and $x \geq x_{i_0}$ such that $x_{i_0} \in \uparrow x$. ■

Before stating the main result, we need the following Lemma:

Lemma 6.1.1

Let $\mathcal{F}$ be a prime filter on a WOL such that $\cap_{O \in \mathcal{F}} O = \uparrow x$ and $G = \{z \mid \uparrow z \in \mathcal{F}\}$.
Then $\sup G = x$.

Proof: G is not empty since $\bot \in G$.
By contradiction, suppose that $z = \sup G \neq x$. Then there exists $O \in \mathcal{F}$ such that $z \notin O$. Let $y \in G$, then $O' = \uparrow y \cap O \in \mathcal{F}$.
Since $O' = \cup_{v \in \downarrow O'} \uparrow v$ and $\mathcal{F}$ is a prime filter, there exists $v \in \downarrow O'$ such that $\uparrow v \in \mathcal{F}$.
By definition of G, $v \leq z$, hence $z \in O'$ and we have a contradiction. ■

Theorem 6.1.2

A WOL is sober for the Scott-topology.

Proof: Let $\mathcal{F}$ be a prime filter, and $S = \cap_{O \in \mathcal{F}} O$. S is non-empty since it contains the top element $\top$.

According to Proposition 6.1.2, $S = \cup_{x \in \downarrow S} \uparrow x$.

Let $O \in \mathcal{F}$. Similarly, $O = \cup_{x \in \downarrow O} \uparrow x$.

Then, since $\mathcal{F}$ is a prime filter, there exists $x \in \downarrow O$ such that $\uparrow x \in \mathcal{F}$.

Hence, for any $y \in \downarrow S$, $x \leq y$, such that: $x \leq \inf \downarrow S$, and for any open $O \in \mathcal{F}$, $\inf \downarrow S \in O$, such that: $\inf \downarrow S \in S$.

Hence, $S = \uparrow x$ for some x, and for any $O \in \mathcal{F}$, $x \in O$.

Conversely, let I be an open such that $x \in I$ and $G = \{z \mid \uparrow z \in \mathcal{F}\}$.

According to Lemma 6.1.1, $\sup G = x \in I$. Hence, since I is a Scott-open, there exists $y \in G$ such that $y \in I$ and, by definition of G, $\uparrow y \in \mathcal{F}$ and $I \in \mathcal{F}$. ■

6.1.3 Decomposition of valuations

We present a new result extending a theorem first introduced for finite valuations in [Tix95]. Limits of sequence of finite valuations are also studied extensively in [GL05], although this result is not stated there. This result allows the decomposition of valuations under some assumptions. Using this general result, we prove that the decomposition is possible in any WOL. In the terms and definitions used in [GL05], it means that on such a lattice, the classes of semi-simple valuations and discrete valuations are the same, and probably others too.

Definition 6.1.8

Let v be a valuation on a sober space (X, τ).
The set $I(v)$ is the set τ quotiented with the following equivalence relation: $O \equiv O'$ if and only if for any open I, $v(O \cap I) = v(O' \cap I)$.
The relation: $x \sqsubseteq y$ if and only if there exists $O \in x$, and $O' \in y$ such that $O' \subset O$ is an order on $I(v)$.
χ (resp. ϕ) is the equivalence class of X (resp. $\emptyset$).

Proof: We have to prove that the relation $\equiv$ is an equivalence relation. Indeed, it is clearly symmetric, reflexive and transitive.

The relation $\sqsubseteq$ is an ordering relation for $\equiv$:

- $x \sqsubseteq x$
- if $x \sqsubseteq y$ and $y \sqsubseteq z$, there exists $I \in x$, $J \in y$ and $K \in z$ such that: $K \subset J \subset I$. Hence, there exists $I \in x$ and $K \in z$ such that $K \subset I$ and $x \sqsubseteq z$.
- if $x \sqsubseteq y$ and $y \sqsubseteq x$, then let $X \in x$. By definition, there exists $I \in x$, $J \in y$ such that $J \subset I$. For $J \in y$, there exists $K \in x$ such that $K \subset J \subset I$. Hence, for any open O, $K \cap O \subset J \cap O \subset I \cap O$, and $v(K \cap O) \leq v(J \cap O) \leq v(I \cap O)$. By definition, $v(I \cap O) = v(K \cap O) = v(X \cap O)$. Hence, for any open O, $v(J \cap O) = v(X \cap O)$ and $X \in y$. ■

Theorem 6.1.3

If there exists a total directed sequence $(x_i)_{i\in\mathbb{N}}$ of elements of $I(v)$ such that:

- $x_0 = \chi$
- *if $i \neq j$, then $x_i \neq x_j$*
- *for any i, $\bigcup_{O\in x_i} O \in x_i$*

then there exists a simple valuation f such that for any $n \in \mathbb{N}$, any open $O_n \in x_n$, and any open I, $v(I) - f(I) = \inf_n v(O_n \cap I)$.

We prove this theorem using the intermediate results presented now. In the following, X is a sober space and v a valuation on X that satisfies the conditions for Theorem 6.1.3.

First we define a set of opens which will be proved to be a prime filter and used for the initial step of the decomposition:

Definition 6.1.9

Let $O \in x_1$. We define: $\mathcal{F}_O = \{O' \in \tau \mid v(O \cup O') = v(X)\}$.

This set has some nice properties:

Proposition 6.1.4

Let I and J be two opens:

- *$v(O \cup I) \in \{v(O); v(X)\}$*
- *if $v(O \cup I) = v(O)$ and $v(O \cup J) = v(O)$, then $v(O \cup (I \cup J)) = v(O)$.*
- *if $v(O \cup I) = v(X)$ and $v(O \cup J) = v(X)$, then $v(O \cup (I \cap J)) = v(X)$.*

Proof: The first part follows directly from the fact that $v(O \cup I) \geq v(O)$, but the only two possible values are $v(O)$ and $v(X)$ since (x_i) is well-ordered and total.
The second part makes use of the decomposition over union and intersection for valuations: $v(O\cup(I\cup J)) = v((O\cup I)\cup(O\cup J)) = v(O\cup I)+v(O\cup J)-v((O\cup I)\cap(O\cup J))$.
Since $(O\cup I)\cap(O\cup J) = O\cup(I\cap J) \subset O\cup I$, then $v((O\cup I)\cap(O\cup J)) = v(O)$, and:

$$v(O \cup (I \cup J)) = v(O)$$

.
The third part is proved in a similar way:
$v(O\cup(I\cap J)) = v((O\cup I)\cap(O\cup J))$, hence $v(O\cup(I\cap J)) = v(O\cup I)+v(O\cup J)-v(O\cup I\cup J)$.
Since $v(O \cup I \cup J) \geq v(O \cup I) = v(X)$, then $v(O \cup I \cup J) = v(X)$ and:

$$v(O \cup (I \cap J)) = v(X)$$

■

We can also prove that $\mathcal{F}_O$ is a prime filter:

Lemma 6.1.2

$\mathcal{F}_O$ is a prime filter.

Proof: We prove the result using the previous Proposition:

- $v(O \cup X) = v(X)$ hence $X \in \mathcal{F}_O$.
- if $I \in \mathcal{F}_O$ and $I \subset J$, then $v(O \cup J) \geq v(O \cup I) = v(X)$. Hence $v(O \cup J) = v(X)$ and $J \in \mathcal{F}_O$.
- if $I \in \mathcal{F}_O$ and $J \in \mathcal{F}_O$, then, according to previous Proposition, $v(O \cup (I \cap J)) = v(X)$. Hence $I \cap J \in \mathcal{F}_O$.
- Let $(O_i)_{i \in I}$ be a family of opens such that: $\cup_{i \in I} O_i \in \mathcal{F}_O$ and for any $J \subset I$, $U_J = \cup_{j \in J} O_j$. Suppose that for any $i \in I$, $v(O \cup O_i) = v(O)$ or, equivalently, $O \cup O_i \in x_1$, then by assumption, also $v(O \cup (\bigcup_{i \in I} O_i)) = v(\bigcup_{i \in I}(O \cup O_i)) = v(O)$: contradiction. ■

Hence, we obtain a prime filter $\mathcal{F}_O$. Then, since the space is sober, this filter defines a point x_0 so that we can define a new valuation:

Proposition 6.1.5

Let $f = v - (v(X) - v(O))\delta_{x_O}$, then for any open I, $f(I) = v(O \cap I)$.

Proof: Let I be an open, then:

- if $x_O \notin I$, then $f(I) = v(I)$. However, then also: $v(O \cup I) = v(O)$, hence: $v(O) + v(I) - v(O \cap I) = v(O)$, and: $v(I) = v(O \cap I)$, such that $f(I) = v(O \cap I)$.
- if $x_O \in I$, then $v(O \cup I) = v(X)$. Hence, $v(O) + v(I) - v(O \cap I) = v(X)$, such that: $v(I) - v(X) + v(O) = v(O \cap I)$, and: $f(I) = v(O \cap I)$.

In any case, we have that: $f(I) = v(O \cap I)$. ■

Another interesting property of this decomposition is that is invariant for any open O' such that $O \equiv O'$. Hence, the decomposition can be done on (x_i).

We can now prove that this decomposition holds for any x_i. First, we define the proposition that we want to prove. We use a strong induction principle in order to state the unicity of the decomposition.

Definition 6.1.10

For any x_i, $F(x_i)$ is true if and only if there exists a simple valuation f such that for any open I, and any open $O_i \in x_i$, $v(I) - f(I) = v(O_i \cap I)$.
$P(x_i)$ is true if and only if:

- *$F(x_j)$ is true for any $x_j \sqsubseteq x_i$*
- *For any $x_k \sqsubseteq x_j \sqsubseteq x_i$ such that $F(x_j)$ is true for $f_j = \sum_l a_l \delta_{c_l}$ and $F(x_k)$ is true for $f_k = \sum_n a_n \delta_{c_n}$, then $\cup_n \{c_n\} \subset \cup_l \{c_l\}$, and if $c_l = c_n$, $a_l = a_n$.*

Theorem 6.1.4

$P(x_i)$ is true for any $i \in \mathbb{N}$.

Proof: We prove the theorem by recursion on $\mathbb{N}$.
$P(x_0)$ is true according to Proposition 6.1.5.

Suppose that $P(x_n)$ is true for some $n \in \mathbb{N}$. Then there exists a valuation $f_n = \sum_k a_k \delta_{c_k}$ such that for any open I, and any open $O_n \in x_n$, $v(I) - f_n(I) = v(O_n \cap I)$.
We consider the valuation $v' = v - f_n$. Since the sequence x_i is total for v, it is also total for v'.
We can then apply Proposition 6.1.5 to v' and obtain a valuation $a_{n+1}\delta_{x_{n+1}}$ such that for any open I and any open $O_{n+1} \in x_{n+1}$, $v(I) - f(I) - a_{n+1}\delta_{x_{n+1}}(I) = v(O_{n+1} \cap I)$ ■

Now we can prove Theorem 6.1.3:

Proof of Theorem 6.1.3 : The decomposition obtained by Theorem 6.1.4 is unique, so we can take: $C = \cup_n \cup_{k_n} \{c_{k_n}\}$ where for any $n \in \mathbb{N}$, $f_n = \sum_k a_{k_n} \delta_{c_{k_n}}$, and for any $c \in C$ there a unique value a_c. We define: $f = \sum_{c \in C} a_c \delta_c$. If f_n is the decomposition obtained for x_n, then by definition of the sequence (x_i), for any $n \in \mathbb{N}$ any $O_n \in x_n$, $O_{n+1} \in x_{n+1}$ and any open I, we have: $v(O_{n+1} \cap I) \leq f_n(I) = v(O_n \cap I)$. Hence, $f(I) = \inf_n v(O_n \cap I)$. ■

We can apply this result to any well-ordered lattice.

Theorem 6.1.5

Any valuation on a WOL equipped with the Scott topology is a simple valuation.

Proof : Let v be a valuation on a WOL lattice.
According to Theorem 6.1.2, the lattice is sober.
Furthermore, there exists a total well-ordered directed sequence (x_i) with the properties described in Definition 6.1.7.
We construct a sequence of opens $(O_n)_{n \in \mathbb{N}}$ that we use for the initial decomposition.
We consider the minimal index α of a non-algebraic element in the sequence (x_i).
If this element does not exist, then the sequence (x_i) contains only algebraic elements, the lattice has a finite number of elements and we take $O_n = \bigcup_{i \geq n} (\uparrow x_i)$.
Otherwise, we consider the sequence of algebraic elements (y_j) that are below x_α in (x_i) and define $O_n = \bigcup_{\substack{x_k \neq y_j \\ j \leq n}} (\uparrow x_k)$.
We can now consider the sequence of the different equivalence classes ρ_n obtained with the sequence $(O_n)_{n \in \mathbb{N}}$. By construction, this sequence is total, and for any n, $\cup_{O \in \rho_n} O \in \rho_n$.
For this sequence, there exist a simple valuation f_α such that for and any open I, $v(I) - f_\alpha(I) = \inf_n v(O_n \cap I)$.
We finish this initial step by taking $g_\alpha = f_\alpha + x_\alpha.\delta_{x_\alpha}$, where $x_\alpha = f_\alpha(X) - f_\alpha(O)$, with $O = \bigcup_{\substack{i \in \mathbb{N} \\ x_i \not\leq x_\alpha}} (\uparrow x_\alpha)$.
Then, for any open I,

$$v(I) - g_\alpha(I) = v(I \cap (\bigcup_{x_i \not\leq x_\alpha} (\uparrow x_i))$$

We conclude by a transfinite recursion on the non-algebraic elements in the sequence (x_i).

For each non-algebraic element x_s, we build a sequence of algebraic elements (y_j) below x_s.

We take for the sequence (y_j) all the (x_i) that are below x_s and build the next valuation g_s exactly as for the initial step above.

Similary, we have for that:

For any non-algebraic element x_s, let $(s_k)_{k\in K}$ be the set of algebraic elements below x_s in the sequence (x_i), and $X_s = \{x \in (x_i) \mid \forall k \in K,\ x \not\leq s_k\}$ we have:

$$v(I) - g_s(I) = v(I \cap (\bigcup_{x \in X_s} (\uparrow x_i)) \tag{6.1}$$

Eventually, or any non-algebraic element x_s, we build a valuation g_s that satisfies 6.1.

Since the lattice is meet-continous, if y is a non-algebraic element such that for any element x, $x \leq y$, then y is the maximal element of the lattice, and the recursion terminates.

Hence, for the maximal element $\top$, we get a simple valuation $g_\top$ such that, according to 6.1:

$$v(I) - g_\top(I) = v(I \cap (\bigcup_{x \not\leq X_\top} (\uparrow x_i)) = v(\emptyset) = 0$$

Then $v = g_\top$. ■

Example : We define $X = (\mathbb{N} \cup \infty)^2$ ordered by: $(n, m) \leq (k, l)$ iff $n \leq k$ and $m \leq l$. This is a meet-continous well-ordered lattice, and we have a sequence:
$(0,0); (1,0); (0,1); (2,0); (1,1); (0,2); \ldots;$
$(\infty,0); (\infty,1); \ldots;$
$(0,\infty); (1,\infty); \ldots;$
$(\infty,\infty).$

Hence X is a WOL.

The sequences used for the decomposition in Theorem 6.1.5 are:
$(0,0) < (1,0) < \cdots < (\infty,0);$
$(0,0) < (1,0) < (1,1) < \cdots < (\infty,1); \ldots;$
$(0,0) < (0,1) < \cdots < (0,\infty); \ldots$

In the following, we assume that X is a WOL.

Theorem 6.1.6

Let μ_n be a directed sequence of simple valuations all bounded by the same positive real M. Then, the pointwise limit:

$$\mu : O \mapsto \sup_n \mu_n(O)$$

is a simple valuation.

Proof: The sequence $\mu_n(O)$ is bounded by M, hence it has a sup.
We then prove that all the requirements listed in the definition of valuations are met by μ. In particular, $\mu_n(A \cup B) = \mu_n(A) + \mu_n(B) - \mu_n(A \cap B)$ entails, that $\mu(A \cup B) = \mu(A) + \mu(B) - \mu(A \cap B)$[4].
By Theorem 6.1.5, μ is also a simple valuation. ■

This property is very important. It expresses a *closure* property for the space of simple valuations. Hence, we can define a language that operates on simple valuations, and expect a simple valuation as the result of the computation, even when the computation is infinite. This is generally not the case. [GL05] studies extensively these limits in the general case.

6.2 Vector cones of simple valuations

Theorem 6.1.5 shows that if the space X is a WOL, then any valuation on X can be represented as a set of points and positive masses on those points. Hence, we will represent it as a cone on the vector space of absolutely convergent point masses.

6.2.1 Vector cones

Definition 6.2.1

A set of vectors C from a vector space V whose scalars is the field of real numbers $\mathbb{R}$ is a cone iff:

- *I finite, and $\forall i \in I$, $a_i \in \mathbb{R}^+$ such that $\sum_{i\in I} a_i < +\infty$, and $x_i \in C \Rightarrow \sum_{i\in I} a_i x_i \in C$.*
- *$x, -x \in C \Longrightarrow x = 0$*

Valuations are a special case of vector cones:

Proposition 6.2.1

Let V be the set of continuous real functions defined on the open sets of the constraint lattice X
Let also W be the subspace $\{\sum_{i\in\mathbb{N}} a_i\delta_{d_i}$, d constraint, $\sum_i |a_i| < +\infty\}$ of all possible infinite linear combinations of Dirac valuations.
Then $\mathcal{V}al(X)$ is a cone of the vector space V. If X is a WOL, then it is a cone of the subspace W.

Proof: $\mathcal{V}al(X)$ is a cone since any positive linear combination of valuations is again a valuation.
Then, if X is a WOL, all valuations are simple valuations. ■

[4] This would not hold with probabilities over a σ-algebra.

Remark 6.2.1

In the literature, cones of vectors are usually used to define a partial order on the vectors. In this case, we already have a natural order on valuations. Besides, it can be proved that the two orders do not coincide.

In the following, we will assume that X is a WOL.

6.2.2 Linear closure operators

In the case of simple valuations, the pointwise order combined with linearity yields the notions of linear closure operator.

First, we recall the definition and fundamental property of closure operators:

Definition 6.2.2

An operator p on a partially ordered set is a closure operator iff:

- $x \leq p(x)$ *(Extensiveness)*
- $p(p(x)) = p(x)$ *(Idempotence)*
- $x \leq y \Longrightarrow p(x) \leq p(y)$ *(Monotonicity)*

The fundamental property for closure operators is:

Proposition 6.2.2

If p is a closure operator, and P the set of its fixed points, then:

$$p(x) = \min(\uparrow x \cap P)$$

In the original CCP, closure operators are the denotational meaning of the programs. For our language, we will lift this notion to linear operators on the vectors space.

Definition 6.2.3

A continuous linear mapping on W is a linear closure operator iff it is a closure operator on the cone of valuations for the pointwise order.

One example of a linear closure operator for valuations is then the following:

Proposition 6.2.3

Let Θ be an operator on constraints, then the linear continuous function $\ltimes\Theta$ is defined on the Dirac valuations as:

$$\ltimes\Theta(\delta_{d_i}) = \delta_{\Theta(d_i)}$$

If Θ is a closure operator, then so is $\ltimes\Theta$.

Proof: The usual property are proved by the fact that: $d \leq d' \Longleftrightarrow \delta_d \leq \delta_{d'}$:

- $x_i \leq \Theta(x_i) \Rightarrow \sum_i a_i\delta_{x_i} \leq \ltimes\Theta(\sum_i a_i\delta_{x_i}) = \sum_i a_i\delta_{\Theta(x_i)}$

- $\Theta(\Theta(x_i)) = \Theta(x_i) \Rightarrow \ltimes\Theta(\ltimes\Theta(\sum_i a_i\delta_{x_i})) = \sum_i a_i\delta_{\Theta(\Theta(x_i))} = \sum_i a_i\delta_{\Theta(x_i)} = \Theta(\sum_i a_i\delta_{x_i})$
- $x_i \leq y_i \Rightarrow \Theta(x_i) \leq \Theta(y_i)$. Hence, $\ltimes\Theta(\sum_i a_i\delta_{x_i}) = \sum_i a_i\delta_{\Theta(x_i)} \leq \sum_i a_i\delta_{\Theta(y_i)} = \ltimes\Theta(\sum_i a_i\delta_{y_i})$ ■

Since a closure operator is idempotent, linear closure operators is a subset of the linear projections. Even more, since the Dirac valuations are all valuations, their images can be uniquely defined on the cone of valuations, and we then have the following extension of the fundamental property on closure operators:

Proposition 6.2.4

Let $\mathcal{C}^*(B) = \{\sum_i a_i b_i \mid a_i \geq 0, \sum_i a_i < +\infty, b_i \in B\}$ be the set of all possible positive linear combination of vectors from B.
Let p be a linear closure operator and x a valuation, then:

$$p(x) = \min(\uparrow x \cap \mathcal{C}^*\{p(\delta_d) \mid d \text{ constraint}\})$$

Proof: For any valuation $v = \sum_i a_i\delta_{d_i}$ and any continuous linear function f:

$$f(v) = \sum_i a_i f(\delta_{d_i})$$

Hence the image of f on the valuation cone is exactly $\mathcal{C}^*(\{f(\delta_d) \mid d \text{ constraint}\})$.
We conclude using the usual property on closure operators. ■

This results shows that linear closure operators can be identified with the cone of their images on valuations. This is also the set of resting points of the associated closure operator on valuations.

6.2.3 Observables for the CCP+P

In this section, we identify the projection of a configuration on the constraints to a simple valuation on constraints. We then show that the relation that maps the projection of a configuration to the projection of its derivative under the operational semantics is consistent with the order over valuations. We also establish some basic convergence results for this relation.

An execution trace from a configuration of the language will give a directed sequence of projected simple valuations on constraints. This sequence will then converge to a simple valuation, which will be the output of the program.

We eventually show that the input/output relation on valuations entailed by a (possible infinite) run of a process is a linear closure operator.

Definition 6.2.4

Let f be a CCP+P configuration and $\Pi(f) = \{(c, p)\}$ its projection. The valuation associated to $\Pi(f)$ is $P(f) = \sum_{(c,p)\in\Pi(f)} p.\delta_c$.

Ordering on valuations is consistent with the derivation under the operational semantics, as show by the next result.

Theorem 6.2.1

If f and f' are two configurations such that $f \longrightarrow f'$, then $P(f) \leq P(f')$. Furthermore, $P(f)(X) = P(f')(X)$ where X stands for the whole constraint lattice.

Proof: Each individual transition from a Dirac valuation yields a greater valuation. ■

Theorem 6.2.2

If $P(\nu_0) \to \ldots P(\nu_n) \to \cdots \to$ is the projection of an execution trace, then it has a limit which is a simple valuation.

Proof: We apply Theorem 6.1.6 to the directed bounded sequence $(P(\nu_n))_{n\in\mathbb{N}}$ ■

We use the fact that the language is confluent, along with the ordering on valuations on constraints to identify infinite fair executions to dominant traces, and use the limit reached during these executions as the observable for the language.

Theorem 6.2.3

CCP+P is confluent.

Proof: This is an extended version of the original confluence result for CCP.

First we notice that the probabilistic choice $\{(\oplus_i(P_i, c_i, p_i), d)\}$ is deterministic.

For all other constructions from the grammar of the language, confluence occurs over individual processes, and we sum up the simple valuations, so that by commutativity of product and sum, we get the same result with both paths. ■

We will use this result to define a notion of fair execution based on the order on valuations. Those executions are characterized by dominant traces.

Definition 6.2.5

An execution trace $F = f_1 \to f_2 \to \cdots \to f_n \to \ldots$ is dominant if and only if for any other execution trace $G = g_1 \to g_2 \to \cdots \to g_n \to \ldots$, then:

$$\lim_{n\to\infty} P(g_n)(O) \leq \lim_{n\to\infty} P(f_n)(O)$$

Corollary 6.2.1

If $F = f_1 \to f_2 \to \cdots \to f_n \to \ldots$ and $G = g_1 = f_1 \to g_2 \to \cdots \to g_n \to \ldots$ are two dominant execution traces, then:

$$\lim_{n\to\infty} P(g_n) = \lim_{n\to\infty} P(f_n)$$

We need to prove that there exist at least one dominant execution trace.

Theorem 6.2.4

For any configuration in CCP+P, there exist at least one dominant execution trace that starts with that configuration.

Proof: Let f be a configuration, and $T_i = \{t_i\colon\ (t_0 = f) \to \cdots \to t_i\}$ be the set of configurations generated by traces of length i.

Then by confluence, for each T_i, there exist a finitely constructible configuration s_j that is superior to each configuration in T_i:

$$s_i \geq t,\ \forall t \in T_i$$

We can construct a sequence of finite traces from $V_i : v_i \to \cdots \to v_{i+1}$, where $s_i \leq v_i$ for all i:

- $V_0 = f$
- $V_i = v_i \to \cdots \to v_{i+1}$ where v_{i+1} realizes the confluence for v_i and s_{i+1}

Those traces exist by confluence. Then, $S = V_0 \to \cdots \to V_n \to \ldots$ is a dominant execution trace. ■

We define the observable associated to a process P and to a valuation on constraints μ.

Definition 6.2.6

Let $\mu = \sum_{i\in\mathbb{N}} a_i\delta_{c_i}$ be a valuation on constraints, and let P be a process.
The observable associated to (P, μ) are:

$$\mathcal{O}(P,\mu) = \{v_f \mid f \text{ execution trace }\}$$

Where $f : f_0 = \cup_{i\in\mathbb{N}}\{(P, c_i, a_i)\} \to \cdots \to f_n \ldots$ is an execution trace, $v_f : O \mapsto \lim_{n\to\infty} P(f_n)(O)$

Proposition 6.2.5

For every P, μ, the valuation $\max\{\mathcal{O}(P,\mu)\}$ exists and is obtained for a dominant trace. We will denote it as $\mathcal{D}(P,\mu)$

We show the connection between the semantics based on closure operators for the original CCP and the semantics based on vector spaces already used in the literature.

Theorem 6.2.5

The operator $\nu_P : f \mapsto \mathcal{D}(P, f)$ extends to a linear operator on the vector space W.

Proof: ν_P is a linear operator on valuations by definition of the language. So, it can be decomposed into its values on the basis (δ_x). Hence, the extension on the whole space W by linearity.

In order to prove that it is also a closure operator, the main difficulty is idempotency.

The first key property that ensures idempotency is the algebraicity of the guards. This ensures that, by taking the limit of an infinite dominant execution, we do not introduce new guarded constraints that could not be reached during the execution. Hence, when starting the same process with this limit, all enabled transitions are enabled also for the same process started with the original valuation, possibly after a finite sequence of steps.

The other important property is the use of constraint guards on choices. Indeed, when restarted with its maximal configuration, all the previously resolved choice reduce to the exact previous outcome. Hence, idempotency is proved for the probabilistic choice construct. ■

This result shows a strong connection between the linear approach for defining semantics on probabilistic CCP, as used in [PW98] and the semantics using closure operator based on fixed points as defined in [SRP91b].

6.3 Denotational semantics

In the previous part, we have shown that the observable associated to a given process is a linear closure operator on the vector cone of simple valuations.

In this section we propose a denotational semantics for which this observable is obtained by a fixed point construction. We then prove the correspondence with the operational semantics.

In the following, $\mathcal{C}^*B$ is the vector cone generated by all the possible positive linear combinations of the vectors in B. E is the whole cone. We define linear closure operators using their set of fixed points, which is also the cone generated by the images on the Dirac valuations. We also use the operation $\ltimes$ (Proposition 6.2.3) to define linear operators using operators on constraints. If f is an operator on valuations, and X a set of valuations, $f(X)$ will denote the set $\{f(x) \mid x \in X\}$. For simplicity, indexes on sums will be omitted. α is a distinguished variable which cannot be used in the programs.

Definition 6.3.1

Let e be a mapping from procedures to cones of valuations on constraints.

Let $[\![\, . \,]\!]_p$ be the function:

- $[\![0]\!]_p(e) = E$
- $[\![c]\!]_p(e) = \mathcal{C}^*\{\delta_{d \wedge c} \text{ for all possible } d\}$
- $[\![c \to P]\!]_p(e) = \mathcal{C}^*\{\delta_d \mid d \not\geq c\} \cup \mathcal{C}^*\{[\![P]\!]_p(e)(\delta_d) \mid d \geq c\}$
- $[\![A|B]\!]_p(e) = [\![A]\!]_p(e) \cap [\![B]\!]_p(e)$
- $[\![\oplus_i(P_i, c_i, p_i)]\!]_p(e) = \mathcal{C}^*\{\sum_i p_i \ltimes f_{c_i}([\![P_i]\!]_p(e)(\delta_d))\}$ *where* $f_c(x) = x \wedge c$
- $[\![\exists X.A]\!]_p(e) = (\ltimes\Phi)^{-1}([\![A]\!]_p(e))$ *where* $\Phi(x) = \exists_X x$
- $[\![p(X)]\!]_p(e) = \ltimes f(e(p))$ *where* $f(x) = \exists_\alpha(\gamma_{\alpha X} \wedge x)\}$

Let $[\![\, . \,]\!]_d$ be the function,

- $[\![\epsilon]\!]_d(e) = e$
- $[\![p(X) :: A]\!]_d(e) =$

$$p' \mapsto \begin{cases} e(p') \text{ if } p' \neq p \\ \ltimes f([\![A]\!]_p(e)) \text{ otherwise} \end{cases}$$

  where: $f(x) = \exists_X(\gamma_{\alpha X} \wedge x)$
- $[\![D.D]\!]_d(e) = [\![D]\!]_d([\![D]\!]_d(e))$

We define the denotational semantics of a process A in the program D as follows:

$$[\![D.A]\!] = [\![A]\!]_p(\underline{fix}([\![D]\!]_d))$$

We show a strong correspondence between this semantics and the observables based on dominant traces.

Theorem 6.3.1

Let P be a process from the CCP+P, we have: $[\![P]\!] = \nu_P : \lambda f \mathcal{D}(P, f)$

Proof: The proof of this results is in two parts:

1. The function $[\![D]\!]_d$ is continuous and monotone, such that the least fixed point exists.
2. The least fixed point $\underline{fix}([\![D]\!]_d)$ composed with $[\![A]\!]_p$ is equal to $\lambda f \mathcal{D}(P, f)$

First, the domain $\mathcal{D}$ that we consider is the domain of mappings from procedure names to cones on valuations. The order of the domain is: $f \leq g$ if and only if for any procedure name p: $g(p) \subset f(p)$. The bottom element $\bot$ for this lattice if $\lambda x.E$, where E is the whole space. This defines a complete lattice for which any directed sequence has a limit, hence it is a domain.

The continuity and monotony of the function $[\![D]\!]_d$ over $\mathcal{D}$ is straight forward. For a directed sequence of elements e_i, the sequence $[\![D]\!]_d(e_i)$ is directed and its limit is $\lambda x. \cap_i [\![D]\!]_d(e_i)(x)$.

The second part is less trivial. First, we introduce the usual property for the domains, which is that $\underline{fix}([\![D]\!]_d) = \sup_n [\![D]\!]_d^n(\bot)$. By continuity of $[\![A]\!]_p$, we also have that: $[\![A]\!]_p(\underline{fix}([\![D]\!]_d)) = \sup_n [\![A]\!]_p([\![D]\!]_d^n(\bot))$.

We then remark that, for any integer n and any constraint c, $[\![A]\!]_p([\![D]\!]^n_d(\bot))(c)$ represents the result of a partial (finite) computation where the recursive procedures have been substituted n times with their corresponding agents. This results is proved by induction on n and the syntax of the language.

Eventually, the limit $\sup_n [\![A]\!]_p([\![D]\!]^n_d(\bot))$ is then exactly the outcome of an infinite maximal computation, as defined in 6.2.5. ■

Our semantics is fully abstract, as stated by the following result.

Theorem 6.3.2

For any context $\mathcal{C}[.]$ and any processes P and Q,
$[\![P]\!] = [\![Q]\!]$ *if and only if* $\lambda f.\mathcal{D}(\mathcal{C}[P], f) = \lambda f.\mathcal{D}(\mathcal{C}[Q], f)$

Proof: The proof of this results derives directly from the fact that the denotational semantics is compositional. The semantics of $\mathcal{D}(\mathcal{C}[P], f)$ is determinitically obtained using $[\![P]\!]$. Hence the equality when $[\![P]\!] = [\![Q]\!]$. ■

Conclusion

This work extends previous linear approaches to probabilistic extensions of the CCP to infinite constraints. The use of valuations has given us a more general framework than previous approaches. Indeed, it is much more appropriate to compute a probabilistic measure on an open, which takes into account all branches contributing to the measure, whereas by following each probabilistic branch, one might be in the situation where the probability of each branch reaches zero but the limit of the sum is not null. This framework is also a good starting point for an extension to continuous probability measures.

The fact that any valuation on a WOL is simple is very important, and assures a very general definition of the CCP+P. The fact that the denotational semantics is deterministic for maximal computations is also a great achievements in the purpose of establishing solid foundations in the semantics of infinite concurrent probabilistic asynchronous programs.

We plan to continue this work by applying our result to other probabilistic protocols. The semantics defined here allows to consider probabilistic properties of a protocol or a program, like the possibility to reach a given constraint representing an outcome of the protocol

Part IV

Application: the dinning cryptographers reloaded

Introduction

The concept of *anonymity* comes into play in those cases in which we want to keep secret the identity of the agents participating to a certain event. There is a wide range of situations in which this property may be needed or desirable; for instance: voting, anonymous donations, and posting on bulletin boards.

Anonymity is often formulated in a more general way as an information-hiding property, namely the property that a part of information relative to a certain event is maintained secret. One should be careful, though, not to confuse anonymity with other properties that fit the same description, notably *confidentiality* (aka *secrecy*). Let us emphasize the difference between the two concepts with respect to sending messages: confidentiality refers to situations in which the content of the message is to be kept secret; in the case of anonymity, on the contrary, it is the identity of the originator, or of the recipient, that has to be kept secret. Analogously, in voting, anonymity means that the identity of the voter associated with each vote must be hidden, and not the vote itself or the candidate voted for. A discussion about the difference between anonymity and other information-hiding properties can be found in [HO03].

An important characteristic of anonymity is that it is usually relative to the capabilities of the observer. In general the activity of a protocol can be observed by diverse kinds of observers, differing in the information they have access to. The anonymity property depends critically on what we consider as observables. For example, in the situation of an anonymous bulletin board, a posting by one member of the group is kept anonymous to the other members; however, it may be possible that the administrator of the board has access to some privileged information that may allow him to infer the member who posted it.

In general anonymity may be required for a subset of the agents only. In order to completely define anonymity for a protocol it is therefore necessary to specify which set(s) of members has to be kept anonymous. A further generalization is the concept of *group anonymity*: the members are divided into a number of sets, and it is revealed which one, among the groups, is responsible for an event, but the information as to which particular member has performed the event remains hidden. In this work, however, we only consider the case of a single group of anonymous users.

Various formal definitions and frameworks for analyzing anonymity have been developed in literature. They can be classified into approaches based on process-calculi [SS96, RS01], epistemic logic [SS99, HO03], and "function views" [HS04]. In this work, we focus on the approach based on process-calculi.

The framework and techniques of process calculi have been used extensively in the area of security, to formally define security properties, and to verify cryptographic protocols. See, for instance, [AG99, Low97, Ros95, Sch96, AL00]. The common denominator is that the various parties involved in the protocol are specified as concurrent processes and present typically a nondeterministic

behavior. In [SS96, RS01], the nondeterminism plays a crucial role in the definition of the concept of anonymity, definition which is based on the so-called "principle of confusion": a system is anonymous if the set of the possible observable outcomes is saturated with respect to the intended anonymous users. More precisely, if in one computation the *culprit* (the user who performs the action) is i and the observable outcome is o, then for every other agent j there must be a computation where j is the culprit and the observable is still o.

The principle of anonymity described above is elegant and general, however it is limited in that it does not cope with quantitative information. Now, many protocols for anonymity use random mechanisms, see, for example, Crowds [RR98], Onion Routing [SGR97], and Freenet [CSWH00]. The probability distribution of these may be known or become known through statistical experiments. From this knowledge, and the observables, one may be able to differentiate the agents quantitatively, namely to deduce that one agent is more likely (has higher probability) to be the culprit than the others. This means that we don't have perfect anonymity. However the definition of the non-deterministic approach (in which of course the random mechanisms are approximated by nondeterministic mechanisms) may still be satisfied, as long as it is possible for each of the other agents to be the culprit, even with very low probability. In other words, the approach in [SS96, RS01], is based on set-theoretic notions, and it is therefore only able to detect the difference between possible and impossible, and not the quantitative differences.

Another advantage in taking into account probabilistic information is that it allows to classify various notions of anonymity according to their strength. See for instance the hierarchy proposed by Reiter and Robin [RR98]. In this work we explore a notion of anonymity which corresponds to the strongest one in [RR98], namely *beyond suspicion*[5]: from the observables all agents appear equally likely to be the culprit.

A probabilistic notion of anonymity was developed (as a part of a general epistemological approach) in [HO03]. The approach there is purely probabilistic, in the sense that both the protocol and the users are assumed to act probabilistically. In particular the emphasis is on the probability of the users to be the culprit.

In this work, we take the opposite point of view, namely we assume that we may know nothing about the users. They may be totally unpredictable, and change attitude every time, so that the choice of being the culprit cannot be quantified probabilistically, not even by repeating statistical observations. Namely, it is a typical nondeterministic choice[6]. We regard this as a special case, though: In

[5] To be more precise we should say that *we think that it corresponds to the intended notion of beyond suspicion in* [RR98]. We cannot prove this correspondence because the definition there is given only informally.

[6] Some people consider nondeterministic choice as a probabilistic choice with unknown probabilities. Our opinion is that the two concepts are different: the notion of probability implies that we can gain knowledge of the distribution by repeating the experiment under

general, we assume that the behavior of the users may be in part probabilistic and in part nondeterministic. As for the protocol, it may use mechanisms like coin tossing, or random selection of a nearby node, which are supposed to exhibit a certain regularity and obey a probabilistic distribution. On the other hand, also the protocol can behave nondeterministically in part, due, for instance, to the (unpredictable) interleaving of the parallel components.

In summary, we investigate a notion of anonymity which combines both probability and nondeterminism, and which is suitable for describing the general situation in which both the users and the protocol can exhibit a combination of probabilistic and nondeterministic behavior. We also investigate the properties of the definition for the particular cases of purely nondeterministic users and purely probabilistic users.

One of the results of our investigation is that the property of anonymity does not depend on the probabilities of the users. We consider this a fundamental property of a good notion of anonymity. In fact, a protocol for anonymity should be able to guarantee this property for every group of users, no matter what is their probability distribution for being the culprit.

In order to define the notion of probability we need, of course, a model of computation able to express both probabilistic and nondeterministic choices. This kind of systems is by now well established in literature, see for instance the probabilistic automata of [SL95], and has been provided with solid mathematical foundations and sophisticated tools for verification. For expressing the protocols, we will use the probabilistic asynchronous π-calculus introduced in [HP00, PH05a], whose semantics is based on a model similar to [SL95].

Some of the results of this document have appeared initialy in [BP05]. These results were then completed and generalized in [BP09], including detailed proofs and comments about the possible implementation of an automated verification of these properties.

---

the same conditions and by observing the frequency of the outcomes. In other words, from the past we can predict the future. This prediction element is absent from the notion of nondeterminism.

Chapter 7

Anonymous protocols: the dining cryptographer

We present in this chapter a particular application of probabilistic and asynchronous anonymity protocol, the Dining Cryptographers.

In Section 7.1 we recall the nondeterministic approach of [SS96, RS01] to the notion of anonymity.

In Section 7.2 we recall the dining cryptographers' Problem by Chaum [Cha88], which will serve as a running example, and we motivate the necessity of copying with probabilities.

Contents

7.1 The nondeterministic approach to anonymity . . 118
7.2 The dining cryptographers' problem 119
7.2.1 Nondeterministic dining cryptographers 119
7.2.2 Limitations of the nondeterministic approach 121
7.2.3 Probabilistic dining cryptographers 122

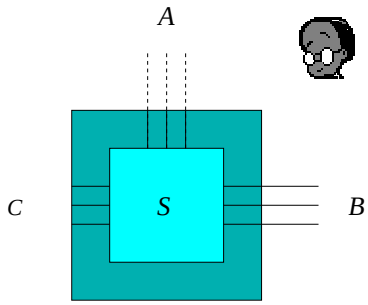


Figure 7.1: Classification of the actions in an anonymous system (cfr. [RS01]).

7.1 The nondeterministic approach to anonymity

In this section we briefly recall the approach in [SS96, RS01]. In these works, the actions of a system S are classified into three sets (see Figure 7.1):

- A: the actions whose performer is intended to remain anonymous for the observer,
- B: the actions that are intended to be completely visible to the observer,
- C: the actions that are intended to be hidden from the observer.

Typically the set A consists of actions of the form $a(i)$, where a is a fixed "abstract" action (the same for all the elements of A), and i represents the identity of an anonymous user. Hence:

$$A = \{a(i) \,|\, i \in I\},$$

where I is the set of all the identities of the anonymous users.

Consider a dummy action d (different from all actions in S) and let f be the function on the actions of $A \bigcup B$ defined by $f(\alpha) = d$ if $\alpha \in A$, and $f(\alpha) = \alpha$ otherwise. Then S is said to be (strongly) anonymous on the actions in A if

$$f^{-1}(f(S\backslash C)) \sim_T S\backslash C,$$

where, following the CSP notation [Hoa85], $S\backslash C$ is the system resulting from hiding C in S, $f(S')$ is the system obtained from S' by applying the relabeling f to each (visible) action, f^{-1} is the relation inverse of f, and $\sim_T$ represents trace equivalence[1].

Intuitively, the above definition means that for any action sequence $\vec{\alpha} \in A^*$, if an observable trace t containing $\vec{\alpha}$ (not necessarily as a consecutive sequence)

[1]The definition given here corresponds to that in [SS96]. In [RS01] the authors use a different (but equivalent) definition: they require $\rho(S\backslash C) \sim_T S\backslash C$ for every permutation ρ in A.

is a possible outcome of $S\backslash C$, then, any trace t' obtained from t by replacing $\vec{\alpha}$ with an arbitrary $\vec{\alpha}\,' \in A^*$ must also be a possible outcome of $S\backslash C$.

We now illustrate the above definition on the example of the dining cryptographers.

7.2 The dining cryptographers' problem

This problem, described by Chaum in [Cha88], involves a situation in which three cryptographers are dining together. At the end of the dinner, each of them is secretly informed by the master whether he should pay the bill or not. So, either the master will pay, or he will ask one of the cryptographers to pay. The cryptographers, or some external observer, would like to find out whether the payer is one of them or the master. However, if the payer is one of them, they also wish to maintain anonymity over the identity of the payer. Of course, we assume that the master himself will not reveal this information, and also we want the solution to be distributed, i.e. communication can be achieved only via message passing, and there is no central memory or central 'coordinator' which can be used to find out this information.

A possible solution to this problem, described in [Cha88], is the following: Each cryptographer tosses a coin, which is visible to himself and to his neighbor to the right. Each cryptographer then observes the two coins that he can see, and announces *agree* or *disagree*. If a cryptographer is not paying, he will announce *agree* if the two sides are the same and *disagree* if they are not. However, if he is paying then he will say the opposite. It can be proved that if the number of *disagrees* is even, then the master is paying; otherwise, one of the cryptographers is paying. Furthermore, if one of the cryptographers is paying, then neither an external observer nor the other two cryptographers can identify, from their individual information, who exactly is paying.

7.2.1 Nondeterministic dining cryptographers

In the approach of [SS96, RS01] the dining cryptographers are formalized as a purely nondeterministic system: the coins are approximated by nondeterministic coins, and the choice on who pays the bill is also nondeterministic.

The specification of the solution can be given in a process calculus style as illustrated below. In the original works [SS96, RS01] the authors used CSP [Hoa85]. For the sake of uniformity here we use the π-calculus [MPW92]. We recall that $+$ ($\sum$) is the nondeterministic sum and $|$ (Π) is the parallel composition. 0 is the empty process. τ is the silent (or internal) action. $\overline{c}m$ and $c(x)$ are, respectively, send and receive actions on channel c, where m is the message being transmitted and x is the formal parameter. ν is an operator that, in the π-calculus, has multiple purposes: it provides abstraction (hiding), enforces synchronization,

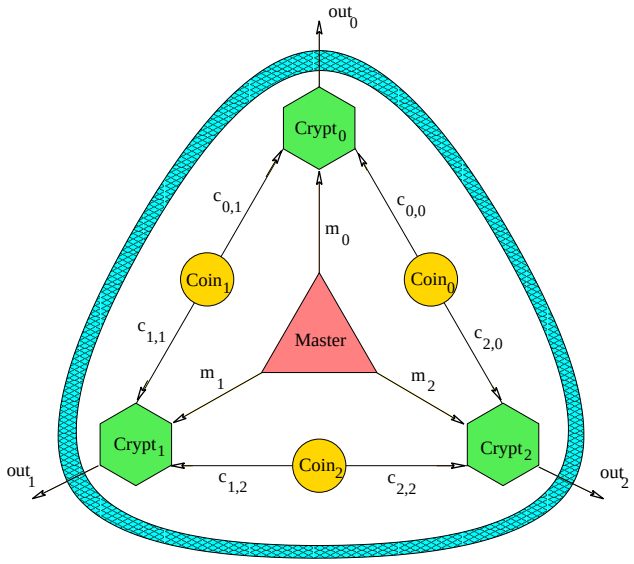


Figure 7.2: Chaum's protocol for the dining cryptographers [Cha88, RS01].

and generates new names. For more details on the π-calculus and its semantics, we refer to [BCPP08].

In the code below, $\oplus$ and $\ominus$ represent the sum and the subtraction modulo 3. Messages `p` and `n` sent by the master are the requests to pay or to not pay, respectively. $\overline{pay}_i$ is the action of paying for cryptographer i.

We remark that we do not need all the expressive power of the π-calculus for this program. In particular, we do not need guarded choice (all the choices are internal because they start with τ), and we do not need neither name-passing nor scope extrusion, thus ν is used just like the restriction operator of CCS [Mil89].

$$
\begin{array}{rcl}
Master & = & \sum_{i=0}^{2} \tau \,.\, \overline{m}_i \mathsf{p} \,.\, \overline{m}_{i\oplus 1} \mathsf{n} \,.\, \overline{m}_{i\oplus 2} \mathsf{n} \,.\, 0 \\
 & & \quad + \quad \tau . \overline{m}_0 \mathsf{n} \,.\, \overline{m}_1 \mathsf{n} \,.\, \overline{m}_2 \mathsf{n} \,.\, 0 \\
Crypt_i & = & m_i(x) \,.\, c_{i,i}(y) \,.\, c_{i,i\oplus 1}(z) \,. \\
 & & \quad \text{if } x = \mathsf{p} \\
 & & \quad\quad \text{then } \overline{pay}_i \ \text{. if } y = z \\
 & & \quad\quad\quad \text{then } \overline{out}_i\, disagree \\
 & & \quad\quad\quad \text{else } \overline{out}_i\, agree \\
 & & \quad\quad \text{else if } y = z \\
 & & \quad\quad\quad \text{then } \overline{out}_i\, agree \\
 & & \quad\quad\quad \text{else } \overline{out}_i\, disagree \\
Coin_i & = & \tau \,.\, Head_i + \tau \,.\, Tail_i \\
Head_i & = & \overline{c}_{i,i} head \,.\, \overline{c}_{i\ominus 1,i} head \,.\, 0 \\
Tail_i & = & \overline{c}_{i,i} tail \,.\, \overline{c}_{i\ominus 1,i} tail \,.\, 0 \\
DCP & = & (\nu \vec{m})(Master \\
 & & \quad | \quad (\nu \vec{c})(\Pi_{i=0}^{2} Crypt_i \ | \ \Pi_{i=0}^{2} Coin_i) \ )
\end{array}
$$

Let us consider the point of view of an external observer. The actions that are to be hidden (the set C) are the communications of the decision of the master and the results of the coins ($\vec{m}$, $\vec{c}$). These are already hidden in the definition of the system DCP. The anonymous users are of course the cryptographers, and the anonymous actions (the set A) is constituted by the $\overline{pay}_i$ actions, for $i = 0, 1, 2$. The observable actions (the set B) is constituted by those of the form $\overline{out}_i\, agree$ and $\overline{out}_i\, disagree$, for $i = 0, 1, 2$.

Let f be the function $f(\overline{pay}_i) = \overline{pay}$ and $f(\alpha) = \alpha$ for all the other actions. It is possible to check that $f^{-1}(f(DCP))) \sim_T DCP$, where we recall that $\sim_T$ stands for trace equivalence. Hence the nondeterministic notion of anonymity, as defined in Section 7.1, is satisfied.

7.2.2 Limitations of the nondeterministic approach

As a consequence of approximating the coins by nondeterministic coins, we cannot differentiate between a fair coin and a biased one. However, it is evident that the fairness of the coins is essential to ensure the anonymity property in the system, as illustrated by the following example.

Example : Assume that, whenever a cryptographer pays, an external observer obtains *almost always* one of the three outcomes represented in Figure **??**, where a stands for *agree* and d for *disagree*. More precisely, assume that these three outcomes appear with a frequency of 33% each, while the missing configuration, d, a, a, appears with a frequency of only 1%. What can the observer deduce? By examining all possible cases, it is easy to see that the coins must be biased, and more precisely, $Coin_0$ and $Coin_1$ must produce almost always *head*, and $Coin_2$ must produce almost always *tail* (or vice-versa). From this estimation, it is immediate to conclude that, in the first case, the payer is *almost for sure* $Crypt_1$, in the second case $Crypt_2$, and in the third case $Crypt_0$.

In the situation illustrated in the above example, clearly, the system does not provide anonymity. However the nondeterministic definition of anonymity is still satisfied (and it is satisfied in general, as long as "almost always" is not "always", i.e. the fourth configuration d, a, a also appears, from time to time). The problem is that the nondeterministic definition can only express whether or not it is possible to have a particular outcome, but cannot express whether one outcome is more likely than the other.

7.2.3 Probabilistic dining cryptographers

The probabilistic version of the protocol can be obtained from the nondeterministic one by attaching probabilities to the coins. We wish to remark that this is the essential change with respect to [SS96, RS01]: we faithfully model the *random mechanisms* of the protocol as probabilistic, rather than approximate them as nondeterministic.

Concerning the choices of the users (represented in this example as the choice of the master), those are in a sense independent from the protocol, and can be either nondeterministic, or probabilistic, or both.

We use the probabilistic π-calculus (π_p) introduced in [HP00, PH05a]. The essential difference with respect to the π-calculus is the presence of a *probabilistic choice operator* of the form

$$\sum_i p_i \alpha_i . P_i$$

where the p_i's represents probabilities, i.e. they satisfy $p_i \in [0, 1]$ and $\sum_i p_i = 1$, and the α_i's are non-output prefixes, i.e. either input or silent prefixes. (Actually, for the purpose of this work, only silent prefixes are used.) The detailed presentation of this calculus is in [BCPP08].

With respect to the program presented in Section 7.2.1, the definition of the $Coin_i$'s must be modified as follows (p_h and p_t represent the probabilities of the outcomes of the coin tossing):

$$Coin_i \quad = \quad p_h \tau \,.\, Head_i + p_t \tau \,.\, Tail_i$$

It is clear that the system obtained in this way combines probabilistic and nondeterministic behavior, not only because the master may be nondeterministic, but also because the various components of the system and their internal interactions can follow different scheduling policies, selected nondeterministically (although it can easily be seen that this latter form of nondeterminism is not relevant for this particular protocol).

Chapter 8

Toward an automated probabilistic anonymity checker

After having introduced the anonymity problem we consider, we explain in this chapter the details about its probabilistic analysis.

In Section 8.1 we briefly recall some basic notions about probabilistic automata and the probability of events (a more formal and detailed presentation can be found in [BCPP08]).

In Section 8.2 we illustrate the notions and assumptions which are at the basis of our notion of anonymity.

In Section 8.3 we present the various models of probabilistic anonymous executions.

In Section 8.4, we expose several considerations toward the implementation of an automatic probabilistic protocol and process checker.

Contents

8.1 Probabilistic automata 126
8.2 Our framework for probabilistic anonymity 126
8.3 The various notions of anonymity 130
8.3.1 Probabilistic anonymity for users with probabilistic and nondeterministic behavior 130
8.3.2 Probabilistic Anonymity for nondeterministic users . 135
8.3.3 Probabilistic Anonymity for fully probabilistic systems 135
8.4 Toward an automatic probabilistic checker 139

8.1 Probabilistic automata

The models of computation combining probabilistic and nondeterministic behavior are by now well established in literature, see for instance the probabilistic automata of [SL95], and have been provided with solid mathematical foundations and sophisticated tools for verification.

By unfolding a probabilistic automaton we obtain a computation tree, whose nodes, in general, offer both probabilistic and nondeterministic choices. In the probabilistic choices, the arcs are weighted with probabilities. The canonical way of defining the probabilistic notions relevant for our work is the following: First we *solve* the nondeterminism, i.e. we choose a function ς (called *scheduler*) which, for each nondeterministic choice in the computation tree, selects one of the possible alternatives. After pruning the tree from all the non-selected alternatives, we obtain a *fully probabilistic tree*. In such a tree, determined by ς, an *execution* (or *run*) is a maximal path, and an *event* is a (measurable) set of executions. In the finite case, we define the probability of an execution as the product of all the weights in its arcs, and the probability of an event e, $p_\varsigma(e)$, as the sum of the probabilities of the executions in e. For the infinite case, and for more details about the above notions, we refer to [BCPP08].

It should be clear, from the description above, that in general the probability of an event *depends on the chosen scheduler*. For example, in Figure 8.1 the tree P represents a computation tree. From its root there is a nondeterministic choice between two transition groups (aka *steps*), which represent probabilistic choices. We adopt the convention of identifying a step by drawing a curve across its transitions. Analogously, there is a nondeterministic choice between two steps in the fourth node at the level immediately below the root. The trees Q, R and S represent the result of pruning P under different schedulers. Let us denote these schedulers by ς, ϑ and φ respectively. The probability of the event b, under each of these schedulers, is: $p_\varsigma(b) = 1/3 + 1/9 = 4/9$, $p_\vartheta(b) = 1/2$, and $p_\varphi(b) = 0$.

8.2 Our framework for probabilistic anonymity

In this section we illustrate the notions and assumptions which constitute the basis for our definition of probabilistic anonymity.

The system in which the anonymous users live and operate is modeled as a probabilistic automaton M [SL95], see [BCPP08]. Like in Section 7.1, we classify the actions of M into the three sets A, B and C, which are determined by the anonymous users, the specific kind of action on which we want anonymity, and the capabilities of the observer:

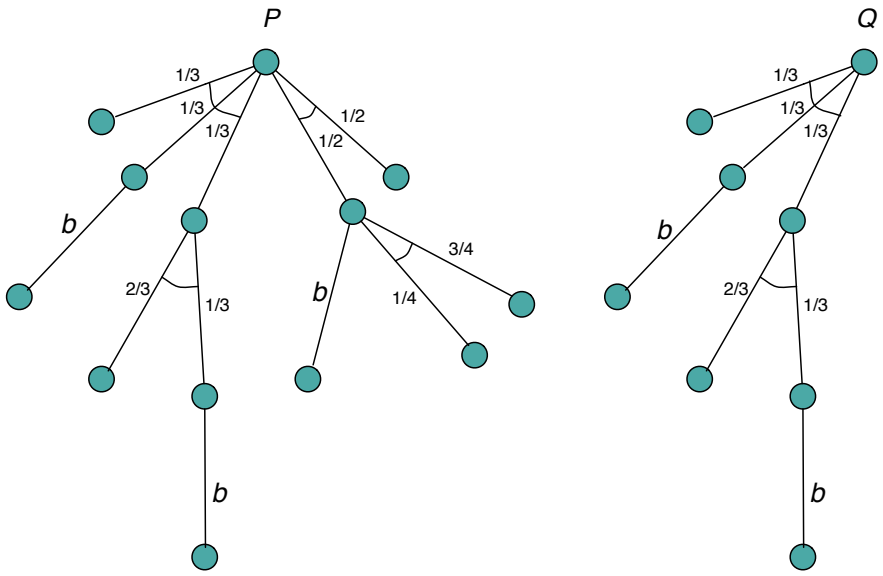


Figure 8.1: A computation tree P and the fully probabilistic trees which derive from it under different schedulers. The irrelevant labels are omitted, and the probability is omitted when it is 1.

- The set of the anonymous actions:

$$A = \{a(i) \mid i \in I\}$$

  where I is the set of the identities of the anonymous users and a is an injective functions from I to the set of actions which we call *abstract action*.

- The set of the visible actions, B. We will use b, b', ... to denote the elements of this set.

- The set of the hidden actions C.

In the following we assume that the actions in C are already restricted in the system, so we do not need to mention them explicitly.

It should be remarked that the term "visible" here is relative: we assume that the observer can see only B and a, but, to the purpose of defining anonymity and checking whether a system is anonymous, we need to leave the actions $a(i)$'s visible (i.e. not restricted) as well.

Differently from [SS96, RS01], we do not assume that the observables are necessarily the traces of visible actions. A trace, indeed, contains not only the information on *which actions* have been executed, but also *in what order* they have been executed. Now, the observer may or may not be able to detect the order, and this is important because the order may give information on the culprit. Another reason for considering such abstraction is to make the analysis simpler. If we know, for instance, that the order of the visible actions does not give any information about the culprit (for instance because we know that the interleaving choices do not depend on the choice of the culprit) then we can forget about it.

In general, we abstract from the (visible) traces by assuming a partition O on them. The observables of the system are then the elements of this partition, denoted by $o, o', \ldots$. Note that each of them is a set of traces.

For instance, in the dining cryptographers, the visible traces are the sequences of the form

$$\overline{out}_i x_i.\overline{out}_j x_j.\overline{out}_k x_k$$

for $\{i, j, k\} = \{0, 1, 2\}$, and $x_i, x_j, x_k \in \{agree, disagree\}$. If we wish to abstract from the order, we can stipulate that all the sequences with the same x_i, x_j, x_k are in the same equivalence class, and that these classes constitute the observables. In the following we will use the notation $\langle x_0, x_1, x_2 \rangle$ to represent such a class. So for instance $\langle d, a, a \rangle$, (where we have abbreviated *agree* by a and *disagree* by d for simplicity), represents all the traces in which the cryptographer 0 announces *disagree*, and the other two announce *agree*.

Another difference from [SS96, RS01] is that we consider the possibility that the observer can somehow affect the scheduler. Thus, we take the set of possible schedulers to be a parameter of the notion of anonymity. Another reason for

considering a restricted class of schedulers is to make the check of the anonymity condition more efficient. If we know, for instance, that the interleavings do not give any information about the culprit then we can fix one particular interleaving, thus reducing the number of schedulers to be taken into account.

Definition 8.2.1

An anonymity system is a tuple (M, I, a, O, Z, p), where M is a probabilistic automaton, I is the set of anonymous users, a is the abstract anonymous action, O is a set of observables, Z is a set of schedulers for M, and for every $\varsigma \in Z$, p_ς is a probability measure on the event space generated by the execution tree of M under ς (denoted by $etree(M, \varsigma)$), i.e. the σ–field generated by the cones in $etree(M, \varsigma)$ (cfr. [BCPP08]).

Note that, as expressed by the above definition, given a scheduler ς, an *event* is a set of executions in $etree(M, \varsigma)$. We introduce the following notation to represent the events of interest:

- $a(i)$: all the executions in $etree(M, \varsigma)$ containing the action $a(i)$
- a : all the executions in $etree(M, \varsigma)$ containing an action $a(i)$ for an arbitrary i
- o : all the executions in $etree(M, \varsigma)$ containing an element of o.

We use the symbols $\cup$, $\cap$ and $\neg$ to represent the union, the intersection, and the complement of events, respectively. By definition of a, we have

$$a = \bigcup_{i \in I} a(i)$$

Furthermore, by definition of O, all the observables are pairwise disjoint:

$$\forall \varsigma \in Z.\ \forall o_1, o_2 \in O.\ \ o_1 \neq o_2 \ \Rightarrow\ p_\varsigma(o_1 \cup o_2) = p_\varsigma(o_1) + p_\varsigma(o_2) \tag{8.1}$$

and they cover all possible traces:

$$\forall \varsigma \in Z.\ \ p_\varsigma(\bigcup_{o \in O} o) = 1 \tag{8.2}$$

In this work we assume there is at most one culprit per run. In other words, we assume that all the $a(i)$'s are pairwise disjoint. This assumption is fundamental for the notion of anonymity we propose, and for the results we obtain.

Assumption 8.2.1 (*At most one culprit*)

$$\forall \varsigma \in Z.\ \forall i, j \in I.\ \ i \neq j \ \Rightarrow\ p_\varsigma(a(i) \cup a(j)) = p_\varsigma(a(i)) + p_\varsigma(a(j))$$

8.3 The various notions of anonymity

8.3.1 Probabilistic anonymity for users with probabilistic and nondeterministic behavior

In this section we develop a notion of anonymity for the general case in which also the users, besides the protocol, combine probabilistic and nondeterministic behavior.

Example: An example of such kind of behavior in the dining cryptographers is the following: assume the master may have a different attitude depending on the group of cryptographers that meet for dinner. Say that there are two groups, and which of them will meet for dinner is decided nondeterministically. The master will select the payer with probabilities $p_0, \ldots, p_3$ in the case of the first group, and $q_0, \ldots, q_3$ in the case of the second. Note that this situation may be quite common in practice: a certain protocol may be used by different groups of users, which may act probabilistically, but whose probability distribution may vary from one group to the other.

Such a master can be represented in π_p as follows:

$$\begin{aligned}
Master &= \tau.Master_1 + \tau.Master_2 \\
Master_1 &= \textstyle\sum_{i=0}^{2} p_i \ \tau \,.\, \overline{m}_i\mathsf{p} \,.\, \overline{m}_{i\oplus 1}\mathsf{n} \,.\, \overline{m}_{i\oplus 2}\mathsf{n} \,.\, 0 \\
&\quad + \quad p_3\tau.\overline{m}_0\mathsf{n} \,.\, \overline{m}_1\mathsf{n} \,.\, \overline{m}_2\mathsf{n} \,.\, 0 \\
Master_2 &= \textstyle\sum_{i=0}^{2} q_i \ \tau \,.\, \overline{m}_i\mathsf{p} \,.\, \overline{m}_{i\oplus 1}\mathsf{n} \,.\, \overline{m}_{i\oplus 2}\mathsf{n} \,.\, 0 \\
&\quad + \quad q_3\tau.\overline{m}_0\mathsf{n} \,.\, \overline{m}_1\mathsf{n} \,.\, \overline{m}_2\mathsf{n} \,.\, 0
\end{aligned}$$

Note that the choice in *Master* is nondeterministic while the choices in $Master_1$ and $Master_2$ are probabilistic.

The notion of anonymity must take into account the probabilities of the $a(i)$'s. When we observe a certain event o, the probability of o having been induced by $a(i)$ must be the same as the probability of o having been induced by $a(j)$ for any other $j \in I$. To formalize this notion, we need the concept of *conditional probability*. Recall that, given two events x and y with $p(y) > 0$, the *conditional probability* of x given y, denoted by $p(x \,|\, y)$, is equal to the probability of x *and* y, divided by the probability of y:

$$p(x \,|\, y) = \frac{p(x \cap y)}{p(y)}$$

We are now ready to propose our notion of anonymity:

Definition 8.3.1

A system (M, I, a, O, Z, p) *is anonymous if*

$$\forall \varsigma, \vartheta \in Z.\ \forall i, j \in I.\ \forall o \in O.$$
$$(p_\varsigma(a(i)) > 0 \wedge p_\vartheta(a(j)) > 0) \Rightarrow p_\varsigma(o \,|\, a(i)) = p_\vartheta(o \,|\, a(j))$$

Example: Consider the system in Example 8.3.1. Assume that the coins are totally fair. For simplicity, let us fix the order of execution of the various components (interleaving)[1], so that the only nondeterministic choice is the choice of the master. Hence we have $Z = \{\varsigma, \vartheta\}$ where ς and ϑ selects $Master_1$ and $Master_2$ respectively. Assume now that $Master_1$ and $Master_2$ select as the payers $i \in I$ with probability p_i and $j \in I$ with probability q_j, respectively. The possible observable events, in both cases, are $o_0 = \langle a, a, d\rangle$, $o_1 = \langle a, d, a\rangle$, $o_2 = \langle d, a, a\rangle$, and $o_3 = \langle d, d, d\rangle$. These are the results in case one of the cryptographers is the payer. The case in which none of them is the payer gives the 4 configurations with an even number of d, which we will indicate by $o_4, \ldots, o_7$. Consider now the possible outcomes of the coins. These are 8: $\langle h, h, h\rangle$, $\langle h, h, t\rangle$, $\ldots \langle t, t, t\rangle$. It is easy to see that, independently from which cryptographer is the payer, each of the above observables is produced by exactly two configurations. If the coins are fair, then, independently from the probability of the selected cryptographer, each observable o corresponds to a cone in the tree (rooted in the node immediately after the selection of the cryptographer) which has probabilistic measure 1/4 (cfr Figure 8.2). Therefore $p_\varsigma(o \,|\, a(i)) = 1/4 = p_\vartheta(o \,|\, a(j))$. Hence Definition 8.3.1 is satisfied.

The behavior of a master which combines nondeterministic and probabilistic behavior can be much more complicated than the one illustrated above. However it is easy to see, by following the reasoning in the example above, that as long as the master does not influence the behavior of the coins, and these are fair, the conditional probability of each observable for a given payer is 1/4.

Proposition 8.3.1

Consider the dining cryptographers with arbitrary master (possibly combining nondeterminism and probability). If the coins are fair under every scheduler, then the system is probabilistically anonymous.

The proof of the above proposition is a straightforward generalization of the Example 8.3.1.

Example: Consider again the system in Example 8.3.1, but assume now that the coins are biased. Say, $Coin_0$ and $Coin_1$ give *head* with probability 9/10 and *tail* with probability 1/10, and vice-versa $Coin_2$ gives *head* with probability 1/10 and *tail* with probability 9/10. (This situation is analogous to that illustrated in Example 7.2.2.) Let us consider the observable $o_0 = \langle a, a, d\rangle$. In case $Crypt_1$ is the payer, then the

[1] It is easy to see that, for this example, it does not matter how many and which interleavings we consider.

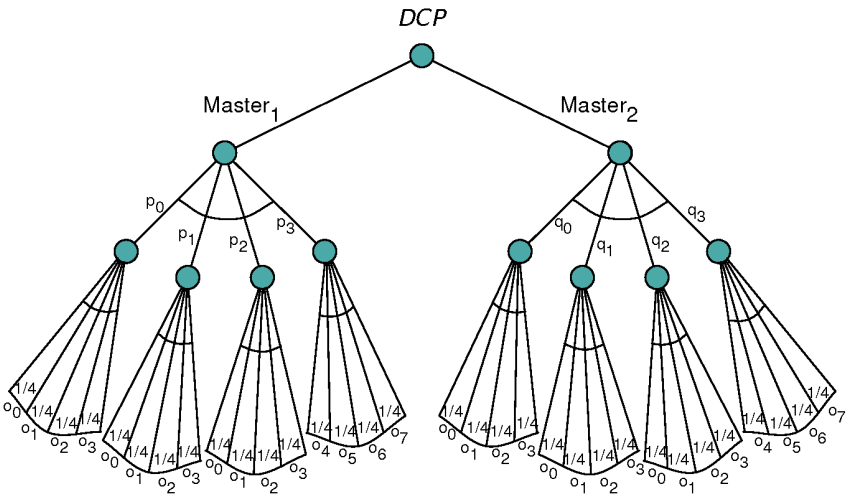


Figure 8.2: Illustration of Example 8.3.1.

probability to get o_0 is equal to the probability that the result of the coins is $\langle h, h, t\rangle$, plus the probability that the result of the coins is $\langle t, t, h\rangle$, which is $r = 9/10 * 9/10 * 9/10 + 1/10 * 1/10 * 1/10 = 730/1000$. In case $Crypt_2$ is the payer, then the probability to get $\langle a, a, d\rangle$ is equal to the probability that the result of the coins is $\langle h, h, h\rangle$, plus the probability that the result of the coins is $\langle t, t, t\rangle$, which is $s = 9/10 * 9/10 * 1/10 + 1/10 * 1/10 * 9/10 = 90/1000$. It is easy to see that the same probability holds for the others cryptographers. Figure 8.3 illustrates the situation. The observables $o_1, \ldots, o_3$ are as before, o_6 is $\langle a, d, d, \rangle$.

Hence, in the biased case, Definition 8.3.2 is not satisfied. And this is what we expect, because the system, intuitively, is not anonymous: when we observe $\langle a, a, d\rangle$, $Crypt_1$ is much more likely to be the payer than any of the others.

Independence from the probability distribution of the users

One important property of Definition 8.3.1 is that it is independent from the probability distribution of the users. Intuitively, this is due to the fact that the condition of anonymity simply means that the cones for o rooted in the nodes that result from the (probabilistic and/or nondeterministic) selection of the culprit, have the same probabilistic measure, and this measure is independent from the probability of the culprit.

We consider this property fundamental for a good notion of anonymity: An anonymity protocol, in fact, should guarantee anonymity independently from (the attitude of) the users who use it, as long as they follow the protocol.

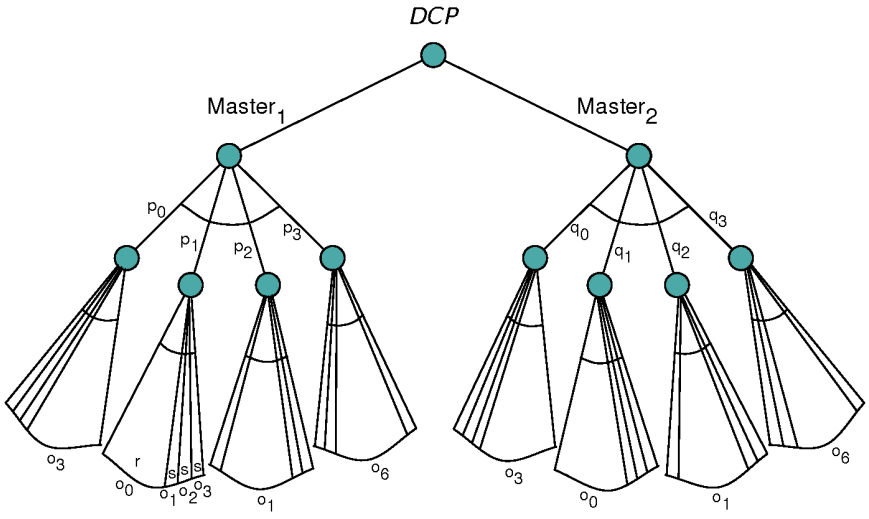


Figure 8.3: Illustration of Example 8.3.1.

Theorem 8.3.1

If (M, I, a, O, Z, p) is anonymous then for any p' which differs from p only on the $a(i)$'s, (M, I, a, O, Z, p') is anonymous.

Proof: Assume that (M, I, a, O, Z, p) is anonymous. Consider a scheduler $\varsigma \in Z$. It is sufficient to show that for every $o \in O$ and every $i \in I$, $p_\varsigma(o|a(i)) = p'_\varsigma(o|a(i))$, or equivalently

$$\frac{p_\varsigma(o \cap a(i))}{p(a(i))} = \frac{p'_\varsigma(o \cap a(i))}{p'(a(i))}$$

Assume that the choice of the $a(j)'s$ is the first choice in $etree(M, \varsigma)$. If this is not the case, then we can the tree transform the tree into one which satisfy this assumption, and in which the probability of events is the same. Figure 8.4 shows the basic step of this transformation. Let Ξ be the set of runs whose visible traces are of the form $a(i).b_1.b_2.\ldots.b_k$ for $b_1.b_2.\ldots.b_k \in o$. We have that $p_\varsigma(o \cap a(i)) = p(\Xi)$. Observe now that $p(\Xi) = p(a(i))p(\Xi')$, where Ξ' is the set of suffixes of the runs in χ that start at the node resulting from the choice of $a(i)$. Hence we have $p_\varsigma(o \cap a(i)) = p(a(i))p(\Xi')$. Analogously, $p'_\varsigma(o \cap a(i)) = p'(a(i))p'(\Xi')$. But, because of the hypothesis that p and p' only differ on the $a(i)'s$, we have $p(\Xi') = p'(\Xi')$, which concludes the proof. ■

Alternative characterization

We give here an alternative characterization of the notion of anonymity. The idea is that a system is anonymous if the probability, under a certain scheduler,

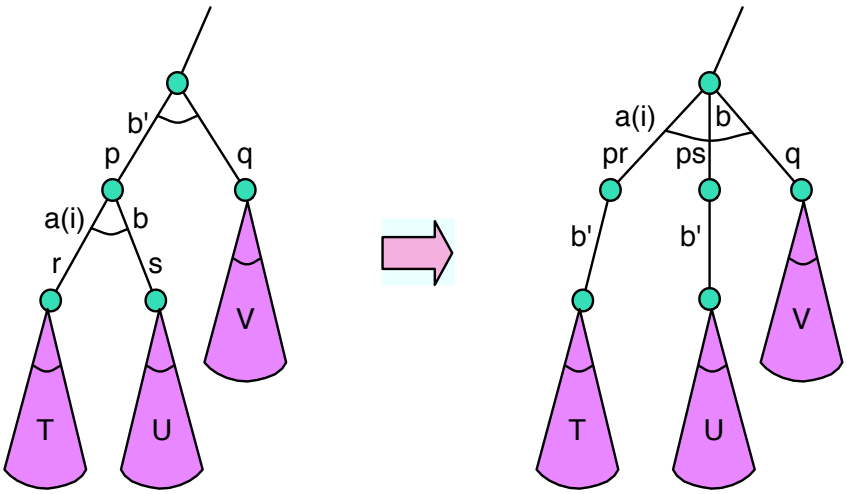


Figure 8.4: Basic transformation step for the proof of Theorem 8.3.1.

of an observable o, given $a(i)$, is the same as the probability of o given a under every other scheduler.

Theorem 8.3.2

A system (M, I, a, O, Z, p) *is anonymous iff*

$$\forall \varsigma, \vartheta \in Z.\ \forall i \in I.\ \forall o \in O.\ \ (p_\varsigma(a(i)) > 0 \wedge p_\vartheta(a) > 0) \Rightarrow p_\varsigma(o \,|\, a(i)) = p_\vartheta(o \,|\, a)$$

Proof: If part) Let $\varsigma, \vartheta \in Z$ and $i, j \in I$ such that $p_\varsigma(a(i)) > 0$ and $p_\vartheta(a(j)) > 0$. Since $p_\vartheta(a(j)) > 0$ implies $p_\vartheta(a) > 0$, by the hypothesis we have $p_\varsigma(o \,|\, a(i)) = p_\vartheta(o \,|\, a)$. Furthermore, by replacing in the hypothesis ς with ϑ and i with j we have $p_\vartheta(o \,|\, a(j)) = p_\vartheta(o \,|\, a)$.

Only if part) Let $\varsigma, \vartheta \in Z$ and $i \in I$ such that $p_\varsigma(a(i)) > 0$ and $p_\vartheta(a) > 0$.

$$\begin{aligned}
p_\vartheta(o \cap a) &= p_\vartheta(o \cap \textstyle\bigcup_{j \in I} a(j)) \\
&= p_\vartheta(\textstyle\bigcup_{j \in I} (o \cap a(j))) \\
&= \textstyle\sum_{j \in I}\ p_\vartheta(o \cap a(j)) && \text{(by Assumption 8.2.1)} \\
&= \textstyle\sum_{p_\vartheta(a(j))>0}\ p_\vartheta(o \cap a(j)) \\
&= \textstyle\sum_{p_\vartheta(a(j))>0}\ p_\vartheta(o \,|\, a(j))\ p_\vartheta(a(j)) \\
&= p_\varsigma(o \,|\, a(i))\ \textstyle\sum_{p_\vartheta(a(j))>0}\ p_\vartheta(a(j)) && \text{(by Definition 8.3.1)} \\
&= p_\varsigma(o \,|\, a(i))\ p_\vartheta(a)
\end{aligned}$$

Hence $p_\vartheta(o \mid a) = p_\vartheta(o \cap a)/p_\vartheta(a) = p_\varsigma(o \mid a(i))$. ■

8.3.2 Probabilistic Anonymity for nondeterministic users

The case in which the users are purely nondeterministic is characterized by the fact that each scheduler determines completely whether an action of the form $a(i)$ takes place or not. Formally:

$$\forall \varsigma \in Z.\ \forall i \in I.\ p_\varsigma(a) = 0 \vee p_\varsigma(a(i)) = 1 \tag{8.3}$$

It is immediate to see that, in the case of nondeterministic users, the definition of anonymity simplifies as follows:

Proposition 8.3.2

A system (M, I, a, O, Z, p) in which the choice of $a(i)$ (for $i \in I$) is nondeterministic in each run, is anonymous if

$$\forall \varsigma, \vartheta \in Z.\ \forall o \in O.\ p_\varsigma(a) = p_\vartheta(a) = 1 \ \Rightarrow\ p_\varsigma(o) = p_\vartheta(o)$$

8.3.3 Probabilistic Anonymity for fully probabilistic systems

In this section we investigate how the removal of the nondeterminism influences our definition of anonymity. We consider therefore purely probabilistic systems. This will allow us also to compare our notion with other probabilistic proposals in literature.

Since the system is totally probabilistic, the probability measures do not depend on the choice of the scheduler. To be more precise, there is only one scheduler. So we can eliminate the component Z from the tuple and we can write $p(x)$ instead of $p_\varsigma(x)$. The definition of probabilistic anonymity given in previous section (cfr. Definition 8.3.1) simplifies into the following:

Remark 8.3.1

A fully probabilistic system (M, I, a, O, p) is anonymous if

$$\forall i, j \in I.\ \forall o \in O.\ \ (p(a(i)) > 0 \wedge p(a(j)) > 0) \Rightarrow p(o \mid a(i)) = p(o \mid a(j))$$

Furthermore, the alternative characterization in Theorem 8.3.2 reduces to the following:

Remark 8.3.2

A fully probabilistic system (M, I, a, O, p) *is anonymous iff*

$$\forall i \in I.\ \forall o \in O.\ \ (p(a(i)) > 0 \wedge p(a) > 0) \quad \Rightarrow \quad p(o \,|\, a(i)) \quad = \quad p(o \,|\, a)$$

In the fully probabilistic case there are two other notions of anonymity that seem rather natural. The first is based on the intuition that a system is anonymous if the observations do not change the probability of $a(i)$: we may know the probability of $a(i)$ by some means external to the system, but the protocol should not increase our knowledge about it. This is already known in literature as *conditional anonymity* (cfr. [HO03]). The second is based on the (similar) idea that observing o rather than o' should not change our knowledge of the probability of $a(i)$.

It is possible to prove that these two notions are equivalent. Furthermore, if we assume that the action a (i.e. the existence of a culprit) is totally visible to the observer, then we can prove that these notions are equivalent to ours. The condition "a is totally visible" means that every observable o indicates unambiguously whether a has taken place or not, i.e. it either implies a, or it implies $\neg a$. In set-theoretic terms this means that either o is contained in a or in the complement of a. Formally:

Assumption 8.3.1 (*The existence of a culprit is observable*)

$$\forall \varsigma \in Z.\ \forall o \in O.\ \ p_\varsigma(o \cap a) = p_\varsigma(o) \vee p_\varsigma(o \cap \neg a) = p_\varsigma(o)$$

We prove now our claims of equivalence.

Proposition 8.3.3

Under Assumption 8.3.1, the following conditions are equivalent to each other and to our condition of anonymity (for the fully probabilistic case, cfr. Remark 8.3.1).

(i) $\forall i \in I.\ \forall o \in O.\ \ p(o \cap a) > 0 \ \Rightarrow\ p(a(i) \,|\, o) = p(a(i))/p(a)$

(ii) $\forall i \in I.\ \forall o, o' \in O.\ \ (p(o \cap a) > 0 \wedge p(o' \cap a) > 0) \ \Rightarrow\ p(a(i) \,|\, o) = p(a(i) \,|\, o')$.

Proof: The equivalence of (i) and the condition in Remark 8.3.2 is easy to prove, and we leave it as an exercise for the reader. As for the equivalence of (i) and (ii), we have:

(i) $\Rightarrow$ (ii)) Let $i \in I$, and $o, o' \in O$ such that $p(o \cap a) > 0$ and $p(o' \cap a) > 0$. By (i) we have $p(a(i) \,|\, o) = p(a(i))/p(a) = p(a(i) \,|\, o')$.

(ii) ⇒ (i)) Let $i \in I$ and $o \in O$ such that $p(o \cap a) > 0$. We have

$$\begin{array}{rcll} p(a(i)) & = & p(a(i) \cap \bigcup_{o' \in O} o') & \text{(by (8.2)} \\ & = & p(\bigcup_{o' \in O}(a(i) \cap o')) & \\ & = & \sum_{o' \in O}\ p(a(i) \cap o') & \text{(by (8.1)} \\ & = & \sum_{p(o' \cap a)>0}\ p(a(i) \cap o') & \\ & = & \sum_{p(o' \cap a)>0}\ p(a(i) \,|\, o')\ p(o') & \\ & = & p(a(i) \,|\, o)\ \sum_{p(o' \cap a)>0}\ p(o') & \text{(by (ii))} \\ & = & p(a(i) \,|\, o)\ p(a) & \text{(by (8.2) and Assumption 8.3.1)} \end{array}$$ ■

Proposition 8.3.3 can be reformulated as a general property of probabilistic spaces, independent from the notion of anonymity. Similar results have been presented in [GH03] and in [GvdLR97] (for the case in which a always occurs, i.e. $p(a) = 1$).

Since the notion of conditional anonymity (as well as the other notion in Proposition 8.3.3) is equivalent to ours, we have that also these notions are independent from the probability of the users. On the other hand, one has to keep in mind that he correspondence only holds under the Assumptions 8.2.1 (only one culprit) and 8.3.1 (the existence of a culprit is observable). Without Assumption 8.3.1 conditional anonymity *is not* independent from the probabilities of the users. Without Assumption 8.2.1 neither conditional anonymity nor our notion are.

Example: Figure 8.5 shows an example in which Assumption 8.3.1 does not hold, and conditional anonymity depends on the probability of the users. In fact the first tree satisfies conditional anonymity, while the second does not. (They both satisfy our notion of anonymity.)

Example: Figure 8.6 shows an example in which Assumption 8.2.1 does not hold, and both conditional anonymity and our notion of anonymity depend on the probability of the users. In fact the first tree satisfies both kinds of anonymity, while the second does not.

Conditional anonymity and nondeterminism

It is not clear whether the characterizations expressed in Proposition 8.3.3 can be generalized to the case of the users with combined nondeterministic and probabilistic behavior. The "naive" extensions obtained by introducing the scheduler in the formulae would not work. Let us consider the first characterization, i.e. conditional anonymity (for the other we would follow an analogous reasoning):

$$\forall i \in I.\ \forall o \in O.\ \ p(o \cap a) > 0\ \Rightarrow\ p(a(i) \,|\, o) = p(a(i))/p(a)$$

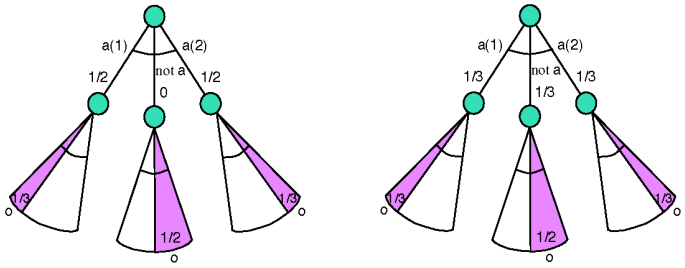


Figure 8.5: Illustration of Example 8.3.3.

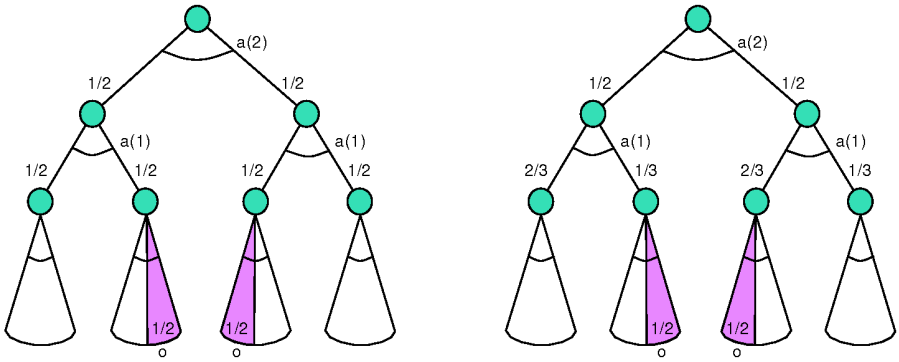


Figure 8.6: Illustration of Example 8.3.3.

One possible way of reintroducing the notion of scheduler is

$$\forall \varsigma, \vartheta \in Z.\ \forall i \in I.\ \forall o \in O.$$
$$(p_\varsigma(o \cap a) > 0 \wedge p_\vartheta(a) > 0) \Rightarrow\ p_\varsigma(a(i) \mid o) = p_\vartheta(a(i))/p_\vartheta(a)$$

However this condition is too strong because it implies that $p_\vartheta(a(i))/p_\vartheta(a)$ is the same for every ϑ, and this is clearly not the case for instance for the nondeterministic and probabilistic master specified in Section 8.3.1.

On the other hand, if we weaken the condition by identifying ς and ϑ:

$$\forall \varsigma \in Z.\ \forall i \in I.\ \forall o \in O.\ \ p_\varsigma(o \cap a) > 0 \Rightarrow\ p_\varsigma(a(i) \mid o) = p_\varsigma(a(i))/p_\varsigma(a)$$

then the condition would be too weak to ensure anonymity, as shown by the following example:

Example: Consider a system in which the master influences the behavior of the coins somehow, in such a way that when $Crypt_i$ is chosen as the payer (say, purely nondeterministically, by ς_i) the result is always $o_0 = \langle d, a, a \rangle$ for $i = 0$, $o_1 = \langle a, d, a \rangle$

for $i = 1$, and $o_2 = \langle a, a, d\rangle$ for $i = 2$. Then we would have $p_{\varsigma_i}(o_j \cap a) > 0$ only if $j = i$, and $p_{\varsigma_i}(a(i) \,|\, o_i) = 1 = p_{\varsigma_i}(a(i))/p_{\varsigma_i}(a)$. Hence the above condition would be satisfied, but the system is not be anonymous at all: whenever we observe $\langle d, a, a\rangle$, for instance, we are sure that $Crypt_0$ is the payer.

8.4 Toward an automatic probabilistic checker

We have formulated the notion of anonymity in terms of observables for processes in the probabilistic π-calculus, whose semantics is based on the probabilistic automata of [SL95]. This opens the way to the automatic verification of the property. We are currently developing a model checker for the probabilistic π-calculus. In order to achieve this goal, several practical issues have to be fixed.

An automatic checker for our probabilistic notion of anonymity needs a full evaluation of the execution. This differs from the model checkers, where the process is checked against a single property. In our case, we want to have a full review of the various outcomes of the protocol and relate them to the inputs.

The natural language for an implementation is the asynchronous version of the π-calculus. In the asynchronous variant of the π-calculus, only internal (blind) choice are allowed and message sending is limited.

For a majority of applications, the asynchronous π-calculus is the natural language for expressing a protocol or an algorithm, since most real-life communications and modern electronic networks use asynchronous communication mediums. Furthermore, the asynchronous constructs of the π_a-calculus can be argued to represent faithfully an asynchronous communication medium, as studied in the first part of this document.

For instance, the implementation of the the dining cryptographers proposed in this work is purely asynchronous. However, asynchronous communications imply algorithmic limitations, such as for implementing a distributed consensus, as studied in [Pal97]. Adding probabilities to the language helps resolving these limitations, and restoring some determinism, as studied in the third part of the document.

A naive evaluation of the probabilistic executions can be very inefficient. When communicating asynchronously, most communications happen without a particular order. This generates interleavings, where several sequence of actions can be observed in any order, but the overall result is the same. When evaluating this kind of executions, it is much more efficient to constrain the evaluation to only one of these possible interleavings. This leads to the definition of bunched executions described in the third part of the document.

Such evaluations have been proved to maintain all relevant informations on

the evaluated process. The may testing semantics is equivalent on bunched and original executions. Furthermore, two bisimilar processes for the bunched executions are bisimilar for the original executions.

We use this optimized evaluator to compute the outcome of a probabilistic protocol or process. In the case of the dining cryptographers, the optimization induced by the bunched executions is not really relevant, but for more complicated protocols, it greatly reduces the search space.

We plan on preparing a generic bunched evaluator that could be used in various common and uncommon situations, in order to contribute to the automatization of probabilistic protocol and process checkers.

Conclusion

In this document, I have presented the various work achieved during my thesis. Although the various results presented are different, they all shared a common interest, which is the study of asynchronous and probabilistic formal languages in the context of security analysis.

The work presented in the first part makes a connection between the algebraic definitions of asynchronous communications and the operational asynchronous mediums. These results give ground for understanding the relations between the abstract algebraic properties of the asynchronous communications and the concrete implementation of the communication mediums.

The second part studies the algebraic notion of unordered communications. In particular, it shows how an evaluation of an asynchronous process can be greatly reduced by reducing the executed transitions, taking into account the redundancy of the various unordered communications, by forcing deterministic computations to happen sequentially.

Among the intuitions that arise from the second part, an important one is that the non-deterministic computations in the asynchronous formal language are in fact internal choices of the agents. Hence, when adding probabilities to these choices, it is possible to have a deterministic language. This is purpose of the third part, which defines a probabilistic concurrent language with internal probabilistic choices.

This language is proved to be fully deterministic, and a denotational semantics is proposed for giving a meaning to the programs of the language. This semantics is correct and fully abstract with regard to the operational semantics of the language.

Eventually, the fourth part presents an application of a probabilistic and non-deterministic language to the security analysis of the dining cryptographers problem.

Further work on these topics include the implementation of a probabilistic security checker in order to automatize the methods used in the fourth part. This security analysis, based on the algebraic properties of the communications, will be relevant for buffered asynchronous communications, according to the results

in the first part, and greatly optimized, according to the results in the second part.

List of Figures

1.1 Confluence with τ 25
1.2 Equivalent graphs that are not homeomorphic. 27
1.3 Partial trace equivalent processes 27
1.4 Two asynchronously bisimilar graphs. 30
1.5 Weakly asynchronously bisimilar graphs. 31
1.6 Relations between the various bisimilations. 32

2.1 Transitions of a $\pi_{\mathfrak{B}}$ sum. 41
2.2 Transitions of the π_a encoding of the $\pi_{\mathfrak{B}}$ sum in Figure 2.1. . . . 42
2.3 Impossibility to encode a stack. 44
2.4 Asynchronously confluent processes. 47

5.1 The Crowds anonymous routing protocol 86

6.1 An example of a WOL 96

7.1 Classification of the actions in an anonymous system (cfr. [RS01]). 118
7.2 Chaum's protocol for the dining cryptographers [Cha88, RS01]. . 120

8.1 A computation tree P and the fully probabilistic trees which derive from it under different schedulers. The irrelevant labels are omitted, and the probability is omitted when it is 1. 127
8.2 Illustration of Example 8.3.1. 132
8.3 Illustration of Example 8.3.1. 133
8.4 Basic transformation step for the proof of Theorem 8.3.1. 134
8.5 Illustration of Example 8.3.3. 138
8.6 Illustration of Example 8.3.3. 138

Bibliography

[ACS98] 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.

[AG99] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 10January 1999.

[AL00] Roberto M. Amadio and Denis Lugiez. On the reachability problem in cryptographic protocols. In *Proceedings of CONCUR 00*, volume 1877 of *Lecture Notes in Computer Science*. Springer, 2000. INRIA Research Report 3915, march 2000.

[And92] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *J. Log. Comput.*, 2(3):297–347, 1992.

[BCPP08] Romain Beauxis, Konstantinos Chatzikokolakis, Catuscia Palamidessi, and Prakash Panangaden. Formal approaches to information-hiding (tutorial). In Gilles Barthe and Cédric Fournet, editors, *Proceedings of the Third Symposium on Trustworthy Global Computing (TGC 2007)*, volume 4912 of *Lecture Notes in Computer Science*, pages 347–362. Springer, 2008. `http://www.lix.polytechnique.fr/~catuscia/papers/Anonymity/Channels-TGC07/tgc.pdf`.

[Bea09] Romain Beauxis. A smooth probabilistic extension of concurrent constraint programming. In Mogens Nielsen, Antonín Kucera, Peter Bro Miltersen, Catuscia Palamidessi, Petr Tuma, and Frank D. Valencia, editors, *SOFSEM*, volume 5404 of *Lecture Notes in Computer Science*. Springer, 2009.

[BHR84] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, 1984.

[BK84] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1,3):109–137, 1984.

[BKT84] J. A. Bergstra, J. W. Klop, and J. V. Tucker. Process algebra with asynchronous communication mechanisms. In A. W. Roscoe S. D. Brookes and G. Winskel, editors, *Proceedings of the Seminar on Concurrency*, volume 197 of *LNCS*, pages 76–95, Pittsburgh, PA, July 1984. Springer.

[Bou92] Gérard Boudol. Asynchrony and the π-calculus (note). Rapport de Recherche 1702, INRIA, Sophia-Antipolis, 1992.

[BP05] Mohit Bhargava and Catuscia Palamidessi. Probabilistic anonymity. In Martín Abadi and Luca de Alfaro, editors, *Proceedings of CONCUR*, volume 3653 of *Lecture Notes in Computer Science*, pages 171–185. Springer, 2005. `http://www.lix.polytechnique.fr/~catuscia/papers/Anonymity/concur.pdf`.

[BP09] Romain Beauxis and Catuscia Palamidessi. Probabilistic and nondeterministic aspects of anonymity. In *To appear in TCS*, 2009.

[BPV08] Romain Beauxis, Catuscia Palamidessi, and Frank D. Valencia. On the asynchronous nature of the asynchronous pi-calculus. In *Concurrency, Graphs and Models*, pages 473–492, 2008.

[CG00] Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theoretical Computer Science (TCS)*, 240(1):177–213, 2000.

[Cha88] David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1:65–75, 1988.

[CPP06] Konstantinos Chatzikokolakis, Catuscia Palamidessi, and Prakash Panangaden. Anonymity protocols as noisy channels. In *TGC*, pages 281–300, 2006.

[CSWH00] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies, International Workshop on Design Issues in Anonymity and Unobservability*, volume 2009 of *Lecture Notes in Computer Science*, pages 44–66. Springer, 2000.

[dBKP92] F. S. de Boer, J. W. Klop, and C. Palamidessi. Asynchronous communication in process algebra. In Andre Scedrov, editor, *Proceedings of the 7th Annual IEEE Symposium on Logic in Computer Science*, pages 137–147, Santa Cruz, CA, June 1992. IEEE Computer Society Press.

[EG99] Joost Engelfriet and Tjalling Gelsema. Multisets and structural congruence of the pi-calculus with replication. *Theor. Comput. Sci.*, 211(1-2):311–337, 1999.

[GH03] P. D. Grunwald and J. Y. Halpern. Updating probabilities. *Journal of Artificial Intelligence Research*, 19:243–278, 2003.

[GJP99] Vineet Gupta, Radha Jagadeesan, and Prakash Panangaden. Stochastic processes as concurrent constraint programs. In *Symposium on Principles of Programming Languages*, pages 189–202. 1999.

[GJS97] Vineet Gupta, Radha Jagadeesan, and Vijay Saraswat. Probabilistic concurrent constraint programming. In Antoni Mazurkiewicz and Józef Winkowski, editors, *CONCUR '97: Concurrency Theory, 8th International Conference*, volume 1243 of *Lecture Notes in Computer Science*, pages 243–257, Warsaw, Poland, 1–4 July 1997. Springer-Verlag.

[GL05] Jean Goubault-Larrecq. Extensions of valuations. *Mathematical Structures in Computer Science*, 15(2):271–297, 2005.

[GvdLR97] R.D. Gill, M. van der Laan, and J. Robins. Coarsening at random: Characterizations, conjectures and counterexamples. In D.Y. Lin and T.R. Fleming, editors, *Proceedings of the First Seattle Symposium in Biostatistics*, Lecture Notes in Statistics, pages 255–294. Springer, 1997.

[HO03] Joseph Y. Halpern and Kevin R. O'Neill. Anonymity and information hiding in multiagent systems. In *Proc. of the 16th IEEE Computer Security Foundations Workshop*, pages 75–88, 2003.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[HP00] Oltea Mihaela Herescu and Catuscia Palamidessi. Probabilistic asynchronous π-calculus. In Jerzy Tiuryn, editor, *Proceedings of FOSSACS 2000 (Part of ETAPS 2000)*, volume 1784 of *Lecture Notes in Computer Science*, pages 146–160. Springer, 2000. `http://www.lix.polytechnique.fr/~catuscia/papers/Prob_asy_pi/fossacs.ps`.

[HS04] Dominic Hughes and Vitaly Shmatikov. Information hiding, anonymity and privacy: a modular approach. *Journal of Computer Security*, 12(1):3–36, 2004.

[HT91] 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, 1991.

[Jon90] C. Jones. *Probabilistic non-determinism*. PhD thesis, University of Edinburgh, 1990.

[Koz81] Dexter Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22(3):328–350, 1981.

[Low97] Gavin Lowe. Casper: A compiler for the analysis of security protocols. In *Proceedings of 10th IEEE Computer Security Foundations Workshop*, 1997. Also in Journal of Computer Security, Volume 6, pages 53-84, 1998.

[Lyn96] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, CS, 1996.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, New York, NY, 1980.

[Mil89] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[MPW92] 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. A preliminary version appeared as Technical Reports ECF-LFCS-89-85 and -86, University of Edinburgh, 1989.

[MS07] Dale Miller and Alexis Saurin. From proofs to focused proofs: A modular proof of focalization in linear logic. In Jacques Duparc and Thomas A. Henzinger, editors, *CSL*, volume 4646 of *Lecture Notes in Computer Science*, pages 405–419. Springer, 2007.

[Nes00] 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*.

[NP00] Uwe Nestmann and Benjamin C. Pierce. Decoding choice encodings. *Inf. Comput.*, 163(1):1–59, 2000.

[Pal97] 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.

[Pal03] Catuscia Palamidessi. Comparing the expressive power of the synchronous and the asynchronous pi-calculus. *Mathematical Structures in Computer Science*, 13(5):685–719, 2003. A short version of this paper appeared in POPL'97. `http://www.lix.polytechnique.fr/~catuscia/papers/pi_calc/mscs.pdf`.

[PH05a] Catuscia Palamidessi and Oltea M. Herescu. A randomized encoding of the π-calculus with mixed choice. *Theoretical Computer Sci-*

ence, 335(2-3):373–404, 2005. `http://www.lix.polytechnique.fr/~catuscia/papers/prob_enc/report.pdf`.

[PH05b] Catuscia Palamidessi and Oltea Mihaela Herescu. A randomized encoding of the pi-calculus with mixed choice. *Theor. Comput. Sci.*, 335(2-3):373–404, 2005.

[Pie00] Alessandra Di Pierro. Randomised algorithms and probabilistic constraint programming. In *Proc. of the ERCIM/Compulog Workshop on Constraints*, June 19-21 2000.

[PW98] A. Di Pierro and Herbert Wiklicky. A Banach space based semantics for probabilistic concurrent constraint programming. In *Proceedings of CATS'98, Computing: the 4th Australian Theory Symposium*, February 2-3 1998.

[Ros95] A. W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *Proceedings of the 8th IEEE Computer Security Foundations Workshop*, pages 98–107. IEEE Computer Soc Press, 1995.

[RR98] Michael K. Reiter and Aviel D. Rubin. Crowds: anonymity for Web transactions. *ACM Transactions on Information and System Security*, 1(1):66–92, 1998.

[RR99] Michael K. Reiter and Aviel D. Rubin. Anonymous web transactions with crowds. *Commun. ACM*, 42(2):32–38, 1999.

[RS01] Peter Y. Ryan and Steve Schneider. *Modelling and Analysis of Security Protocols.* Addison-Wesley, 2001.

[San95] Davide Sangiorgi. On the proof method for bisimulation (extended abstract). In Jirí Wiedermann and Petr Hájek, editors, *MFCS*, volume 969 of *Lecture Notes in Computer Science*, pages 479–488. Springer, 1995.

[Sar89] Vijay A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Melon University, 1989.

[Sch96] S. Schneider. Security properties and CSP. In *Proceedings of the IEEE Symposium Security and Privacy*, 1996.

[Sel97] Peter Selinger. First-order axioms for asynchrony. In Antoni Mazurkiewicz and Józef Winkowski, editors, *CONCUR97: Concurrency Theory*, volume 1243 of *Lecture Notes in Computer Science*, pages 376–390. Springer-Verlag, 1997.

[SGR97] P.F. Syverson, D.M. Goldschlag, and M.G. Reed. Anonymous connections and onion routing. In *IEEE Symposium on Security and Privacy*, pages 44–54, Oakland, California, 1997.

[SL95] 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: 481-496.

[SRP91a] Vijay A. Saraswat, Martin Rinard, and Prakash Panangaden. Semantic foundations of concurrent constraint programming. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 333–352. ACM Press, 1991.

[SRP91b] Vijay A. Saraswat, Martin Rinard, and Prakash Panangaden. Semantic foundations of concurrent constraint programming. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 333–352, Orlando, Florida, January 21–23, 1991. ACM SIGACT-SIGPLAN, ACM Press. Preliminary report.

[SS96] Steve Schneider and Abraham Sidiropoulos. CSP and anonymity. In *Proc. of the European Symposium on Research in Computer Security (ESORICS)*, volume 1146 of *Lecture Notes in Computer Science*, pages 198–218. Springer, 1996.

[SS99] Paul F. Syverson and Stuart G. Stubblebine. Group principals and the formalization of anonymity. In *World Congress on Formal Methods (1)*, pages 814–833, 1999.

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

[Tix95] R. Tix. Stetige bewertungen auf topologischen räumen. Master's thesis, TH Darmstadt, 1995.

Index

π-calculus, 18
$\pi_{\mathfrak{B}}$-calculus, 36
π_a-calculus, 26
π_{ic}-calculus, 25
π_{io}-calculus, 60
π_{sc}-calculus, 23
(primitive) constraints, 82

algebraic element, 97
anonymity, 117
anonymity system, 129
anonymous actions, 128
asynchronous bisimulation, 30
Axiom of Dependent Choice, 93

bisimilarity, 28
bisimulation, 28
bound names, 20, 37
buffer, 34
buffer's content, 35
bunched labeled transition system, 64
bunched trace, 67
bunched transition, 67

CCP+P, 81
conditional anonymity, 137
confluence, 23
congruence, 52
context, 52
Crowds anonymous routing, 86
Cylindric constraint system, 82

denotational semantics, 92
dining cryptographers, 119
dominant traces, 106

early bisimilarity, 52, 54
early semantics, 21, 50
entailment relation, 82

FIFO, 33
filter, 94
finite subsets, 82
finitely algebraic, 83
focusing, 64
free names, 20, 37
fully abstract, 110
fully probabilistic systems, 135

indeterminates, 82
infinite runs, 87
input prefix, 19

labeled transition system, 21, 64
late bisimilarity, 51, 54
late semantics, 21, 50
late transition system, 50
LIFO, 33
linear closure operator, 104

naive late bisimilarity, 59
nondeterministic users, 135

observational equivalence, 26

prime filter, 95
probabilistic automata, 126
probabilistic tree, 126

Queues, 33

renaming, 53

sendable items, 35
sending continuation, 19
sending prefix, 19

Simple Constraint System, 82
simple constraint system, 82
Simple Valuation, 94
sober space, 94
Stacks, 33
strong asynchronous bisimilarity, 30
strong bisimilarity, 28
strong input/output late bisimilarity, 60
strong late bisimilarity, 51
structural congruence, 21, 36
synchronous communication, 13

testing semantics, 27, 67
tokens, 82
total sequence, 93

valuation, 93
variables, 82
vector cone, 103

weak asynchronous bisimulation, 31
well-founded, 95
well-ordered, 96