

Introduction to Choreographies

First edition (DRAFT)

Fabrizio Montesi
Department of Mathematics and Computer Science
University of Southern Denmark

© Fabrizio Montesi 2019, all rights reserved.

Last update: December 10, 2019

Contents

<i>Preface: Alice, Bob, Concurrency, and Distribution</i>	5
<i>This book</i>	9
1 Inference systems	11
1.1 Example: Flight connections	11
1.2 Derivations	13
1.3 Underivable propositions	17
1.4 Rule derivability and admissibility	19
1.4.1 Derivable rules	20
1.4.2 Rule admissibility	22
2 Simple choreographies	29
2.1 Syntax	30
2.2 Semantics	31
3 From choreographic programs to process programs	35
3.1 Simple processes	36
3.1.1 Syntax	36
3.1.2 Semantics	38
3.2 Endpoint Projection (EPP)	41
3.2.1 From choreographies to processes	41
3.2.2 Towards a correct EPP	45
3.2.3 Simple Concurrent Choreographies	46
3.3 Correctness of EPP	48
4 Local computation	55
4.1 Stores, expressions, and evaluation	55
4.2 Stateful Choreographies	58
4.2.1 Syntax	58
4.2.2 Semantics	59

CONTENTS

4.3	Stateful Processes	60
4.3.1	Syntax	61
4.3.2	Semantics	61
4.3.3	EndPoint Projection	61
4.4	Correctness of EPP	62
5	Conditionals and knowledge of choice	67
5.1	Modelling Conditionals	67
5.1.1	Choreographies	67
5.1.2	Processes	70
5.1.3	EndPoint Projection	70
5.2	Knowledge of choice	71
5.2.1	EPP with detection	72
5.3	Selections	75
5.3.1	Choreographies	76
5.3.2	Processes	77
5.3.3	EPP for choreographies with selections	79
5.3.4	Operational correspondence for EPP with selections	80
6	Recursion	89
6.1	Choreographies	89
6.1.1	Syntax	89
6.1.2	Semantics	91
6.2	Processes	95
A	Solution to selected exercises	97
	List of Notations	103
	Index	104

Preface:

Alice, Bob, Concurrency, and Distribution

We live in the era of *concurrency*, the performance of multiple tasks at a time, and *distribution*, the act of computing using communicating connected devices. These aspects have become pervasive in modern computing. Today, even mobile phones and tiny computers like Raspberry Pis feature multiple processing units of different kinds, with purposes that go from generic computing to more specialised ones like graphics and artificial intelligence. Our computer networks are getting bigger than ever, with the rise of the World Wide Web, telecommunications, cloud computing, and the Internet of Things. This transformation is making the number of computer programs that communicate with each other over a network explode. By 2025, the Internet alone is expected to connect from 25 to 100 billion devices [OECD 2016].

On the one hand, modern computer networks and their applications have become the drivers of our technological advancement. They give us better citizen services, a more efficient industry (Industry 4.0), new ways to connect socially, and even better health with smart medical devices. On the other hand, these systems and their software are increasingly complex. Services depend on other services to function. For example, a web store might depend on an external service provided by a bank to carry out customer payments. The web store, the customer's web browser, and the bank service are thus integrated: they communicate with each other to reach the goal of transferring the right amount of money to the right recipient, such that the customer can get the product she wants from the store. In concurrent and distributed systems, the heart of integration is the notion of *protocol*: a document that prescribes the communications that the involved parties should perform in order to reach a goal.

It is important that protocols are clear and precise. If they are ambiguous, we risk that the designers of different parts of the same system interpret them differently, which leads to errors. The consequences of such errors can be dire,

including data getting in the system (deadlocks), data corruption, or data leaks. The more we equip programmers with solid methods for specifying and implementing protocols correctly, the more likely they are to succeed at integrating the different parts of concurrent and distributed systems correctly. The ultimate quest is to increase the reliability of these systems.

It is also important that protocols are as concise as possible: the bigger a protocol or the harder it is to write it down, the higher the chance that we make some mistake in writing it. Computer scientists and mathematicians might get a familiar feeling when presented with the necessity of achieving clarity, precision, and conciseness in writing. A computer scientist could point out that we need a good *language* to write protocols. A mathematician could say that we need a good *notation*. There is no contradiction here, since a language is, in fact, a notation. Both computer scientists and mathematician might say that we could (or should!) be even more ambitious and seek an *elegant* language.

Needham and Schroeder introduced an interesting notation for writing protocols in **1978**. A communication of a message M from the participant A to the participant B is written

$$A \rightarrow B : M .$$

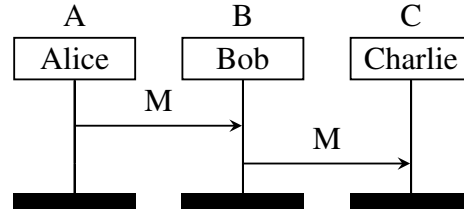
To define a protocol where A sends a message M to B , and then B passes the same message to C , we can just compose communications in a sequence:

$$\begin{array}{l} A \rightarrow B : M \\ B \rightarrow C : M . \end{array}$$

This notation is called “Alice and Bob notation”, due to a presentational style found in security research whereby the participants A and B represent the fictional characters Alice and Bob, respectively, who use the protocol to perform some task. There might be more participants, like C in our example—typically a shorthand for Carol, or Charlie. The first mention of Alice and Bob appeared in the seminal paper by **Rivest et al. [1978]** on their public-key cryptosystem:

“For our scenarios we suppose that A and B (also known as Alice and Bob) are two users of a public-key cryptosystem”.

Over the years, researchers and developers created many more protocol notations. Some of these notations are graphical rather than textual, like Message Sequence Charts **[International Telecommunication Union 1996]**. The message sequence chart of our protocol with Alice, Bob, and Charlie looks as follows.



For our particular example, the graphical representation of our protocol (as a message sequence chart) and our previous textual representation (in Alice and Bob notation) are equivalent, in the sense that they contain the same information. This is not always the case: not all notations are equally expressive.

In the beginning of the 2000s, researchers and practitioners took the idea of protocol notations even further, and developed the idea of *choreography*. A choreography offers more details. For example, some details that might be included are:

- the kind of data being transmitted, or even the actual functions used to compute the data to be transmitted;
- nested protocols, i.e., the ability to call another protocol like a procedure;
- the state of participants, e.g., their memory states.

Choreographies are typically written in languages designed to be readable *mechanically*. This makes them amenable to be used in computer programs and, also, rigorous mathematical reasoning. In this book, we will start with a very simple choreography language and then progressively extend it with more sophisticated features, like parallelism and recursion. We will see that it is possible to define mathematically a *semantics* for choreographies, which gives us an interpretation of what running a protocol means. We will also see that it is possible to translate choreographies to a theoretical model of executable programs, which gives us an interpretation of how choreographies can be correctly implemented in the real world.

Although choreographies still represent a young and active area of research, they have already emerged in many places. In 2005, the World Wide Web Consortium (W3C)—the main international standards organisation for the web—drafted the Web Services Choreography Description Language (WS-CDL), for defining interactions among web services [W3C WS-CDL Working Group 2004]. In 2011, the global technology standards consortium Object Management Group (OMG) introduced choreographies in their notation for business processes [Object Management Group 2011]. The recent paradigm of microservices [Dragoni et al.

2017] advocates the use of choreographies to achieve better scalability. All this momentum is motivating a lot of research on both the theory of choreographies and its application to programming [Ancona et al. 2016, Hüttel et al. 2016]. These days, Alice and Bob surely are in the spotlight.

This book

This book is an introduction to the basic theory of choreographies. It explains what choreographies are and how we can model them mathematically. Its primary intended audience consists of computer science students and researchers, but it is also designed to be approachable by mathematicians (willing to become) familiar with context-free grammars.

The aim of this book is to be pedagogical, and to equip the reader with a new perspective on how we can abstract, design, and reason about concurrent and distributed systems. It is not an aim to be comprehensive, and we will not present features to capture all possible protocols. References to alternative notations, further developments, and techniques to model choreographies are given where appropriate. It is assumed that the reader is familiar with the notion of concurrency and the basic intuition of how distributed systems are programmed.

Prerequisites To read this book, you should be familiar with:

- discrete mathematics and the induction proof method;
- context-free grammars (only basic knowledge is required);
- basic data structures, like trees and graphs;
- concurrent and distributed systems.

These prerequisites are attainable in most computer science B.Sc. degrees, or with three years of relevant experience.

To study choreographies, we are going to define choreography languages and then write choreographies as terms of these languages. The syntax of languages is going to be defined using context-free grammars. To give meaning to choreographies, we are going to use extensively Plotkin’s structural approach to operational semantics [Plotkin 2004].

The rules defining the semantics of choreographies are going to be rules of inference, borrowing from deductive systems. Knowing formal systems based on rules of inference is an advantage, but not a requirement for reading this book:

chapter 1 provides a brief introduction to the essential knowledge on these systems that we need for the rest of the book. The reader familiar with inference systems or structural operational semantics can safely skip the first chapter and jump straight to chapter 2.

An important aspect of choreographies is determining how they can be executed correctly in concurrent and distributed systems, in terms of independent programs for *processes*. To model process programs, we will borrow techniques from the area of process calculi. We will introduce the necessary notions on process calculi as we go along, so knowing this area is not a requirement for reading this book. The reader familiar with process calculi will recognise that we borrow many ideas from Milner’s seminal calculus of communicating systems [Milner 1980].

Some exercises in this book are marked with !, indicating a higher degree of difficulty. For the exercises marked with \hookrightarrow , a solution is given in appendix A.

Chapter 1

Inference systems

Before we venture into the study of choreographies, we need to become familiar with the formalism that we are going to use throughout this book: inference systems. Inference systems are widely used in formal logic and the specification of programming languages (our case).¹

Definition 1 (Inference systems and inference rules). *An inference system is a set of inference rules (also called rules of inference). An inference rule has the form*

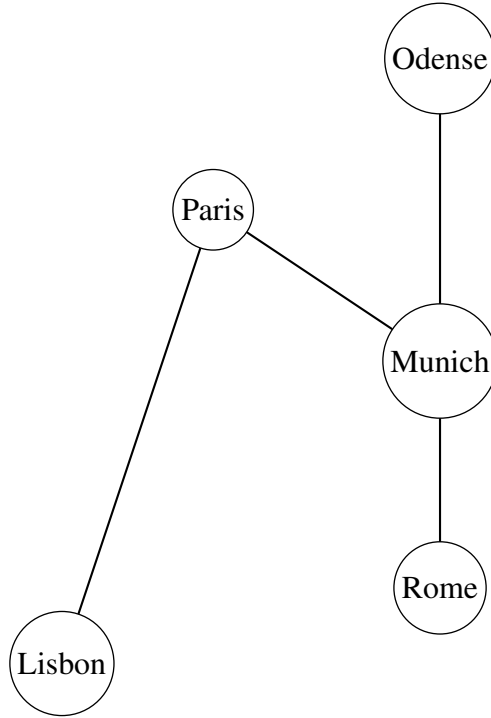
$$\frac{\text{Premise 1} \quad \text{Premise 2} \quad \dots \quad \text{Premise } n}{\text{Conclusion}}$$

and reads “If the premises Premise 1, Premise 2, ..., and Premise n hold, then Conclusion holds”. It is perfectly valid for an inference rule to have no premises: we call such rules axioms, because their conclusions always hold. An inference rule has always exactly one conclusion.

1.1 Example: Flight connections

Consider the following undirected graph of direct flights between cities.

¹For further reading on these systems, see the lecture notes by [Martin-Löf \[1996\]](#).



Let A , B , and C range over cities. We denote that two cities A and B are connected by a direct flight with the *proposition* $\text{conn}(A, B)$. Then, we can represent our graph as the set of axioms below.²

$$\frac{}{\text{conn}(\text{Odense}, \text{Munich})} \quad \frac{}{\text{conn}(\text{Munich}, \text{Rome})} \quad \frac{}{\text{conn}(\text{Paris}, \text{Munich})}$$

$$\frac{}{\text{conn}(\text{Paris}, \text{Lisbon})}$$

Notice that our graph is undirected, meaning that all direct flights are available also in the opposite direction. This is not faithfully represented by our axioms: if $\text{conn}(A, B)$, it should also be the case that $\text{conn}(B, A)$. One option to solve this discrepancy is to double the number of our axioms, to include their symmetric versions—e.g., we would add an axiom concluding with $\text{conn}(\text{Munich}, \text{Odense})$. A more concise option is to formalise the general concept of symmetry of connections with a rule of inference, as follows.

$$\frac{\text{conn}(A, B)}{\text{conn}(B, A)} \text{SYM}$$

The label SYM is the name of our new inference rule; it is just a decoration to remember what the rule does (SYM is a shorthand for symmetry). Rule SYM tells

²This example is inspired by [Pfenning's](#) lecture notes on inference rules, where he also uses inference rules to model graphs [\[Pfenning 2012\]](#).

1.2. Derivations

$$\begin{array}{c}
 \overline{\text{conn}(\text{Odense}, \text{Munich})} \quad \overline{\text{conn}(\text{Munich}, \text{Rome})} \quad \overline{\text{conn}(\text{Paris}, \text{Munich})} \\
 \\
 \overline{\text{conn}(\text{Paris}, \text{Lisbon})} \\
 \\
 \frac{\text{conn}(A, B)}{\text{conn}(B, A)} \text{SYM} \quad \frac{\text{conn}(A, B)}{\text{path}(A, B)} \text{DIR} \quad \frac{\text{path}(A, B) \quad \text{path}(B, C)}{\text{path}(A, C)} \text{TRANS}
 \end{array}$$

Figure 1.1: An inference system for flights.

us that if we have a connection from any A to any B (premise), then we have a connection from B to A (conclusion). In this rule, A and B are *schematic variables*: they can stand for any of our cities. So if we were to add more connections in the future, their symmetric version would also become immediately available thanks to rule SYM.

Now that we are satisfied with how our inference system captures the graph, we can use it to find flight paths from a city to another. Say that for any two cities A and B , the proposition $\text{path}(A, B)$ denotes that there is a path from A to B . Finding paths is relatively easy. First, we observe that direct connections give us a path. (In the following rule, DIR stands for direct.)

$$\frac{\text{conn}(A, B)}{\text{path}(A, B)} \text{DIR}$$

Second, we formulate a rule for multi-step paths. If there is a path from A to B , and a path from B to C , then we have a path from A to C . In other words, paths are transitive. (TRANS stands for transitivity.)

$$\frac{\text{path}(A, B) \quad \text{path}(B, C)}{\text{path}(A, C)} \text{TRANS}$$

The whole system is displayed in fig. 1.1.

1.2 Derivations

The key feature of an inference system is the capability of performing *derivations*. Suppose that we wanted to answer the following question.

Is there a flight path from Odense to Rome?

Answering this question in our inference system for flight connections corresponds to showing that $\text{path}(\text{Odense}, \text{Rome})$ holds. To do this, we have to build a *derivation tree*. We will call derivation trees also simply *derivations*, or *proof trees* (since they prove that something can be derived in a given inference system).

From Odense to Rome We start looking for our derivation from what we want to conclude with—our conclusion is what we want to prove.

$$\text{path}(\text{Odense}, \text{Rome})$$

Observe that we have only two inference rules that can conclude something of this form: DIR and TRANS. Let us try to apply DIR first, by substituting its schematic variables with the cities that we need:

$$\frac{\text{conn}(\text{Rome}, \text{Odense})}{\text{path}(\text{Odense}, \text{Rome})} \text{ DIR .}$$

Our derivation is not complete, because we are left with a premise that we are now responsible for proving: $\text{conn}(\text{Odense}, \text{Rome})$. We can read what we have now as: if Rome is connected to Odense, then there is a path from Odense to Rome. However, proving $\text{conn}(\text{Odense}, \text{Rome})$ seems unfeasible: there is no rule that concludes that, and we cannot hope to conclude it by applying rule SYM, since that would require in turn to prove $\text{conn}(\text{Rome}, \text{Odense})$.

Now that we know that rule DIR is not helpful for our derivation, our only remaining option is to apply rule TRANS. Clearly, we should instantiate A as Odense and C as Rome in the application of TRANS. How should we instantiate B , though? (We informally write $B?$ for “we do not know what B should be yet” here.)

$$\frac{\text{path}(\text{Odense}, B?) \quad \text{path}(B?, \text{Rome})}{\text{path}(\text{Odense}, \text{Rome})} \text{ TRANS}$$

One way would be to try out all possible cities as our B , in search of a city that will allow us to continue our derivation. However, looking at the definition of our graph, it is easy to see that the right choice is to pick Munich, since that city is connected to both Odense and Rome.

$$\frac{\text{path}(\text{Odense}, \text{Munich}) \quad \text{path}(\text{Munich}, \text{Rome})}{\text{path}(\text{Odense}, \text{Rome})} \text{ TRANS}$$

What we have above is an instantiation of rule TRANS. (We will call rule instantiations also rule applications in the remainder, since it corresponds to “applying” the rule to the premises in order to reach the conclusion.) The derivation depends on the validity of some premises. We can read it as a proof of the

1.2. Derivations

statement “If $\text{path}(\text{Odense}, \text{Munich})$ holds and $\text{path}(\text{Munich}, \text{Rome})$ holds, then $\text{path}(\text{Odense}, \text{Rome})$ holds”. In other words, our derivation of $\text{path}(\text{Odense}, \text{Rome})$ has two hypotheses: $\text{path}(\text{Odense}, \text{Munich})$ and $\text{path}(\text{Munich}, \text{Rome})$. Thus, we have gone from the task of deriving $\text{path}(\text{Odense}, \text{Rome})$ to the tasks of deriving $\text{path}(\text{Odense}, \text{Munich})$ and $\text{path}(\text{Munich}, \text{Rome})$, respectively. Thanks to the notation of inference rules, we can complete these tasks just by “digging deeper” with further rule applications. Since we know that Odense and Munich are connected, we can try using rule DIR.

$$\frac{\frac{\text{conn}(\text{Odense}, \text{Munich})}{\text{path}(\text{Odense}, \text{Munich})} \text{DIR} \quad \text{path}(\text{Munich}, \text{Rome})}{\text{path}(\text{Odense}, \text{Rome})} \text{TRANS}$$

We now have to prove $\text{conn}(\text{Odense}, \text{Munich})$. We have that as the conclusion of one of our axioms, so we can conclude that branch of our derivation by applying the related axiom.

$$\frac{\frac{\text{conn}(\text{Odense}, \text{Munich})}{\text{path}(\text{Odense}, \text{Munich})} \text{DIR} \quad \text{path}(\text{Munich}, \text{Rome})}{\text{path}(\text{Odense}, \text{Rome})} \text{TRANS}$$

We can finish our derivation for the right premise $\text{path}(\text{Munich}, \text{Rome})$ likewise.

$$\frac{\frac{\text{conn}(\text{Odense}, \text{Munich})}{\text{path}(\text{Odense}, \text{Munich})} \text{DIR} \quad \frac{\frac{\text{conn}(\text{Munich}, \text{Rome})}{\text{path}(\text{Munich}, \text{Rome})} \text{DIR}}{\text{path}(\text{Odense}, \text{Rome})} \text{TRANS}$$

What we have now is a *complete* derivation, i.e., a derivation without hypotheses. We can tell by the fact that there are no more premises left to prove. Since the derivation is complete, it shows that $\text{path}(\text{Odense}, \text{Rome})$ can always be derived in our inference system, without making any assumptions. In general, we say that a proposition is derivable if there exists at least one complete derivation with such proposition as conclusion.

The structure of derivations We now move to the formal definition of derivations.

First, some notation. Let \mathcal{D} (for derivation) range over derivations and p range over propositions. (Sometimes we will also write \mathcal{P} , for “proof”, to denote a

derivation.) We shall write $\frac{\mathcal{D}}{p}$ for “the derivation \mathcal{D} has conclusion p ”. For example, assume that \mathcal{D} is our latest derivation, as follows.

$$\mathcal{D} \triangleq \frac{\frac{\text{conn}(\text{Odense}, \text{Munich})}{\text{path}(\text{Odense}, \text{Munich})} \text{DIR} \quad \frac{\text{conn}(\text{Munich}, \text{Rome})}{\text{path}(\text{Munich}, \text{Rome})} \text{DIR}}{\text{path}(\text{Odense}, \text{Rome})} \text{TRANS}$$

The symbol \triangleq in the equation above means “is defined as”. Therefore, we have that $\frac{\mathcal{D}}{\text{path}(\text{Odense}, \text{Rome})}$.

Observe that \mathcal{D} can be seen as a tree, by viewing each rule application as a node of the tree, where the root node is the rule application that reaches the conclusion of the derivation. (In the last derivation, the root is the application of rule TRANS.) This is a very useful property that we will use throughout the book, in particular to apply the principle of induction in our formal reasonings about derivations.

Definition 2 (Derivation). *Let S be an inference system. Then:*

- Any application of a rule in S is a derivation in S .
- Let $\mathcal{D}_1, \dots, \mathcal{D}_n$ be derivations in S , for some natural number $n > 0$, with respective conclusions p_1, \dots, p_n . Then, any application of a rule R in S with p_1, \dots, p_n as premises is a derivation in S .

Intuitively, a complete derivation is a derivation where the “leaves” of the derivation tree (the base cases) are all axioms. We can formalise this by tuning slightly the base case of the previous definition. (The difference is emphasised.)

Definition 3 (Complete Derivation). *Let S be an inference system.*

- Any application of an axiom in S is a complete derivation in S .
- Let $\mathcal{D}_1, \dots, \mathcal{D}_n$ be complete derivations in S , for some natural number $n > 0$, with respective conclusions p_1, \dots, p_n . Then, any application of a rule R in S with p_1, \dots, p_n as premises is a complete derivation in S .

We impose that derivations are finite: we are interested in proofs that can be checked mechanically in finite time.

Exercise 1. *Prove that the following statements hold for the system in fig. 1.1.*

1. *The proposition $\text{path}(\text{Paris}, \text{Rome})$ is derivable.*

1.3. Underivable propositions

$$\begin{array}{c}
\overline{\text{conn}(\text{Odense}, \text{Munich})} \quad \overline{\text{conn}(\text{Munich}, \text{Rome})} \quad \overline{\text{conn}(\text{Paris}, \text{Munich})} \\
\\
\overline{\text{conn}(\text{Paris}, \text{Lisbon})} \\
\\
\frac{\text{conn}(A, B)}{\text{conn}(B, A)} \text{SYM} \\
\\
\frac{\text{conn}(A, B)}{\text{path}(A, B, 1)} \text{DIRW} \quad \frac{\text{path}(A, B, n) \quad \text{path}(B, C, m)}{\text{path}(A, C, n + m)} \text{TRANSW}
\end{array}$$

Figure 1.2: Weighted rules for flight paths.

2. The proposition $\text{path}(\text{Odense}, \text{Odense})$ is derivable. (Hint: in rule TRANS, nothing forbids A from being the same as C .)
3. For every A , the proposition $\text{path}(A, A)$ is derivable.

Exercise 2. Prove the following statements about the system in fig. 1.1.

1. For any A , there are infinitely many derivations that conclude $\text{path}(A, A)$.
2. For any A and B , if there exists at least one derivation that concludes $\text{path}(A, B)$, then there are infinitely many derivations that conclude $\text{path}(A, B)$.

Exercise 3 (\leftrightarrow). Consider the system in fig. 1.2, which replaces rules DIR and TRANS respectively with the alternative rules DIRW and TRANSW, which measure the length of a path (paths are “weighted”, with each connection having weight 1).

Prove that, for any A and B , if $\text{path}(A, B)$ is derivable in the system in fig. 1.1, then there exists n such that $\text{path}(A, B, n)$ is derivable in the system in fig. 1.2.

Suggestion: proceed by structural induction on the proof of $\text{path}(A, B)$.

1.3 Underivable propositions

Showing that a proposition holds requires showing a proof, i.e., a derivation, in the inference systems of interest. Once the proof is shown, then we just need to check that all rules have been applied correctly. If we are convinced that this is the case, then we are convinced that the proof is correct and that the proposition indeed holds.

Knowing what can be derived is paramount to establish the adequacy of an inference system, but it is just as important to check what *cannot* be derived. But how can we prove that something cannot be derived? This can be tricky, because it requires us to reason about all the possible derivations that could, potentially, conclude with our proposition and showing that none of those derivations can actually be built.

Consider a simple example: showing that $\text{conn}(\text{Lisbon}, \text{Rome})$ is not derivable. Intuitively, there is no direct connection between Lisbon and Rome in our graph. But how can we show this formally?

First, we observe that the only rules that can have a conclusion of the form $\text{conn}(A, B)$ for any A and B are our axioms and rule SYM. This observation conveniently restricts the set of rules that we have to consider. None of the axioms can be applied for $A = \text{Lisbon}$ and $B = \text{Rome}$, as in our case. We are left only with rule SYM, so any search of a derivation of $\text{conn}(\text{Lisbon}, \text{Rome})$ must begin as follows.

$$\frac{\text{conn}(\text{Rome}, \text{Lisbon})}{\text{conn}(\text{Lisbon}, \text{Rome})} \text{SYM}$$

Again, by the same argument, there is no rule to apply for $\text{conn}(\text{Rome}, \text{Lisbon})$ but SYM. So we obtain that the proof *must* continue with another application of SYM.

$$\frac{\frac{\text{conn}(\text{Lisbon}, \text{Rome})}{\text{conn}(\text{Rome}, \text{Lisbon})} \text{SYM}}{\text{conn}(\text{Lisbon}, \text{Rome})} \text{SYM}$$

We got back to where we started: we have to prove $\text{conn}(\text{Lisbon}, \text{Rome})$. Since we have only one way to prove it, and we have just shown that it leads to the exact same premise, this points out that our proof search will go on indefinitely and that we will never reach a finite derivation. So, $\text{conn}(\text{Lisbon}, \text{Rome})$ cannot be proven.

Let us be more formal, to convince ourselves that $\text{conn}(\text{Lisbon}, \text{Rome})$ cannot be derived more decisively. Since every derivation is a finite tree, we can measure the height of a derivation as a natural number (it is the height of the tree). Thus, among all the proofs of $\text{conn}(\text{Lisbon}, \text{Rome})$, there is at least one of minimal height, in the sense that there is no other proof of $\text{conn}(\text{Lisbon}, \text{Rome})$ with lower height. We attempt at finding this minimal proof. The reasoning goes as before, and we soon end up seeing that a minimal proof necessarily starts as follows.

$$\frac{\frac{\text{conn}(\text{Lisbon}, \text{Rome})}{\text{conn}(\text{Rome}, \text{Lisbon})} \text{SYM}}{\text{conn}(\text{Lisbon}, \text{Rome})} \text{SYM}$$

1.4. Rule derivability and admissibility

$$\begin{array}{c}
\overline{\text{conn}(\text{Odense}, \text{Munich})} \quad \overline{\text{conn}(\text{Munich}, \text{Rome})} \quad \overline{\text{conn}(\text{Paris}, \text{Lisbon})} \\
\\
\frac{\text{conn}(A, B)}{\text{conn}(B, A)} \text{SYM} \\
\\
\frac{\text{conn}(A, B)}{\text{path}(A, B, 1)} \text{DIRW} \quad \frac{\text{path}(A, B, n) \quad \text{path}(B, C, m)}{\text{path}(A, C, n + m)} \text{TRANSW}
\end{array}$$

Figure 1.3: A limited and weighted flight system.

So we have to find some proof of our premise $\text{conn}(\text{Lisbon}, \text{Rome})$ on top. But if such a proof exists, it would be a proof of $\text{conn}(\text{Lisbon}, \text{Rome})$ that is *smaller* than the proof that we are building (because it would not have the first two applications of SYM of our proof). Thus our minimal proof must be bigger than another proof, and we reach a contradiction.

Exercise 4. Consider the system in fig. 1.3, which removes the direct flight from Paris to Munich.

Prove that it is not possible to derive $\text{path}(\text{Lisbon}, \text{Munich}, 1)$.

Exercise 5. Prove that it is not possible to derive $\text{path}(\text{Lisbon}, \text{Munich}, 2)$ using the system in fig. 1.3.

Exercise 6 (!). Prove that there exists no n such that $\text{path}(\text{Lisbon}, \text{Munich}, n)$ is derivable in the system in fig. 1.3.

1.4 Rule derivability and admissibility

Consider the system in fig. 1.4, which replaces rule TRANS from fig. 1.1 with rule STEP. The difference is that rule STEP requires a direct connection from the source A to some city B , and then a path from B to the destination C .

From an algorithmic perspective, adopting rule TRANS or rule STEP might lead to slightly different search strategies. Searching for a path from A to C using rule STEP roughly corresponds to: look up in our database of direct connections (given by the axioms and their reflexive closure, thanks to rule SYM) from A to some B , and then recursively try to find a path from B to C ; if we fail, we have to try with another B , if possible. By contrast, searching for a path from A to C using rule TRANS corresponds to recursively trying to find a path from A to some B , and then again recursively trying to find a path from B to C ; again, if we fail,

$$\begin{array}{c}
 \overline{\text{conn}(\text{Odense}, \text{Munich})} \quad \overline{\text{conn}(\text{Munich}, \text{Rome})} \quad \overline{\text{conn}(\text{Paris}, \text{Munich})} \\
 \overline{\text{conn}(\text{Paris}, \text{Lisbon})} \\
 \frac{\text{conn}(A, B)}{\text{conn}(B, A)} \text{SYM} \quad \frac{\text{conn}(A, B)}{\text{path}(A, B)} \text{DIR} \quad \frac{\text{conn}(A, B) \quad \text{path}(B, C)}{\text{path}(A, C)} \text{STEP}
 \end{array}$$

Figure 1.4: An alternative way of constructing paths.

we have to try with another B , if possible. Of course, these strategies are only for paths not covered already by rule DIR, which covers the case in which A and C are connected directly.

Since the two systems are different, a key question is whether they are *equally powerful*, in the sense that every derivable proposition in one of the two is derivable also in the other.

There are only two kinds of propositions that we can derive in our two systems: $\text{conn}(A, B)$ and $\text{path}(A, B)$. It is easy to see that, for any A and B , $\text{conn}(A, B)$ is derivable in the system with rule TRANS if and only if it is derivable in the system with rule STEP, simply because the two systems share exactly the same rules for deriving conn propositions. For propositions of the form $\text{path}(A, B)$, the situation is more complicated. We tackle the two directions separately (from the system with rule STEP to the system with rule TRANS, and vice versa). The exploration of each direction leads to its own useful new concept—derivable rules and admissible rules.

1.4.1 Derivable rules

We start by showing that the system with rule TRANS (fig. 1.1) can derive all paths that can be derived in the system with rule STEP (fig. 1.4).

To do this, we prove that adding rule STEP to the system with rule TRANS would not add any new derivable propositions. Recall that rule STEP looks as follows.

$$\frac{\text{conn}(A, B) \quad \text{path}(B, C)}{\text{path}(A, C)} \text{STEP}$$

The key observation here is that we can build a derivation that, from the premises $\text{conn}(A, B)$ and $\text{path}(B, C)$, concludes $\text{path}(A, C)$ by using the rules

1.4. Rule derivability and admissibility

in fig. 1.1. Here it is.

$$\frac{\frac{\text{conn}(A, B)}{\text{path}(A, B)} \text{DIR} \quad \text{path}(B, C)}{\text{path}(A, C)} \text{TRANS} \quad (1.1)$$

The derivation above holds for any A , B , and C . Since it is parametric on these schematic variables, a more correct name might be derivation scheme, but we will allow ourselves the abuse of terminology and simply call it a derivation. This derivation is proof that rule STEP is *derivable* in the system in fig. 1.1. In general, we say that a rule is derivable whenever its conclusion can be derived from its premises by using rules that are already in the system. In other words, if we can build a derivation from the premises of the rule to its conclusion, then the rule is derivable.

A very nice and convenient property of derivable rules is that their applications can be rewritten locally inside of derivations, without the need for modifying the derivations of the premises. Let us see an example to understand what this means. From the derivation in eq. (1.1), we now know that we can rewrite every application of rule STEP into a valid derivation in the system in fig. 1.1 as follows. We also show where the derivations of the premises go (\rightarrow here means “is rewritten to”):

$$\frac{\frac{\text{conn}(A, B)}{\text{path}(A, C)} \text{STEP} \quad \text{path}(B, C)}{\text{path}(A, C)} \rightarrow \frac{\frac{\text{conn}(A, B)}{\text{path}(A, B)} \text{DIR} \quad \text{path}(B, C)}{\text{path}(A, C)} \text{TRANS} \quad (1.2)$$

We can apply this transformation to any derivation in the system with rule STEP (fig. 1.4). For instance, consider the following derivation of a multi-hop path from Odense to Lisbon:

$$\frac{\text{conn}(\text{Odense}, \text{Munich}) \quad \frac{\frac{\text{conn}(\text{Paris}, \text{Munich})}{\text{conn}(\text{Munich}, \text{Paris})} \text{SYM} \quad \frac{\text{conn}(\text{Paris}, \text{Lisbon})}{\text{path}(\text{Paris}, \text{Lisbon})} \text{DIR}}{\text{path}(\text{Munich}, \text{Lisbon})} \text{STEP} \quad \text{STEP}$$

To translate this derivation into a derivation in the system in fig. 1.1, we can just rewrite each application of rule STEP as indicated by eq. (1.2). The order in which we pick and rewrite these applications does not matter.³ For example, here is the

³In fact, thanks to the locality of our transformation, one could even devise a concurrent algorithm that replaces all occurrences of STEP in parallel.

result of replacing the top-right occurrence of STEP first:

$$\frac{\frac{\text{conn}(\text{Odense, Munich})}{\text{path}(\text{Odense, Munich})} \quad \frac{\frac{\frac{\text{conn}(\text{Paris, Munich})}{\text{conn}(\text{Munich, Paris})} \text{SYM} \quad \frac{\text{conn}(\text{Paris, Lisbon})}{\text{path}(\text{Paris, Lisbon})} \text{DIR}}{\text{path}(\text{Munich, Paris})} \text{DIR}}{\text{path}(\text{Munich, Lisbon})} \text{TRANS}}{\text{path}(\text{Odense, Lisbon})} \text{STEP}$$

And here the result of replacing also the remaining occurrence:

$$\frac{\frac{\text{conn}(\text{Odense, Munich})}{\text{path}(\text{Odense, Munich})} \text{DIR} \quad \frac{\frac{\frac{\frac{\text{conn}(\text{Paris, Munich})}{\text{conn}(\text{Munich, Paris})} \text{SYM} \quad \frac{\text{conn}(\text{Paris, Lisbon})}{\text{path}(\text{Paris, Lisbon})} \text{DIR}}{\text{path}(\text{Munich, Paris})} \text{DIR}}{\text{path}(\text{Munich, Lisbon})} \text{TRANS}}{\text{path}(\text{Odense, Lisbon})} \text{TRANS} \quad (1.3)$$

What we have in eq. (1.3) is a complete derivation that is valid in the system in fig. 1.1.

1.4.2 Rule admissibility

We now move to the other direction: proving that adding rule TRANS to the system with rule STEP would not add any new derivable propositions.

Recall that rule TRANS is defined as follows.

$$\frac{\text{path}(A, B) \quad \text{path}(B, C)}{\text{path}(A, C)} \text{TRANS}$$

As a first attempt, we could try the same strategy that we followed in section 1.4.1: deriving the conclusion $\text{path}(A, C)$ from the premises $\text{path}(A, B)$ and $\text{path}(B, C)$ using the rules in fig. 1.4.

Unfortunately, we reach a dead end pretty quickly when trying to show that rule TRANS is derivable in the system with rule STEP. The only way to build a path with multiple connections is by using rule STEP, which requires a conn as premise. But our only available premises are path propositions, and we have no rule that allows us to conclude conn from a path.

We resort to a different proof technique and show that rule TRANS is *admissible* in the system with rule STEP (fig. 1.4). An admissible rule is one that does not add any new derivable propositions. All derivable rules are also admissible, but admissible rules are not necessarily derivable (just like our case here with rule

1.4. Rule derivability and admissibility

TRANS). It is sometimes convenient to mark admissible rules and their applications explicitly. Here, we will distinguish them by using a dashed horizontal line.

Theorem 1. *The rule*

$$\frac{\text{path}(A, B) \quad \text{path}(B, C)}{\text{path}(A, C)} \text{ TRANS}$$

is admissible in the system in fig. 1.4.

Proof. We need to prove that, for every derivation \mathcal{D} using the rules in fig. 1.4 and rule TRANS, there exists a derivation with the same conclusion that uses only the rules in fig. 1.4. To shorten our text, we shall say that a derivation is TRANS-free when it does not contain any applications of rule TRANS.

We proceed by induction on the size of \mathcal{D} . We have a case for each one of the rules that can be applied last in \mathcal{D} . For the induction to work, we prove a stronger result: for every derivation \mathcal{D} using the rules in fig. 1.4 and rule TRANS, there exists a derivation with the same conclusion that uses only the rules in fig. 1.4 *and is not bigger than* \mathcal{D} . This last remark about size is going to be important in the last case of this proof.

Case \mathcal{D} ends with one of the axioms. In these cases \mathcal{D} is trivially TRANS-free (as it consists only of the axiom), so the thesis holds by picking \mathcal{D} .

Case \mathcal{D} ends with an application of rule SYM:

$$\mathcal{D} = \frac{\frac{\mathcal{D}'}{\text{conn}(A, B)}}{\text{conn}(B, A)} \text{ SYM}$$

for some derivation \mathcal{D}' .

All rules that can conclude $\text{conn}(A, B)$ are either axioms or rules that have only conn propositions as premises. Since TRANS does not conclude with a conn proposition, \mathcal{D}' is necessarily TRANS-free and so is \mathcal{D} .

Case \mathcal{D} ends with an application of rule DIR:

$$\mathcal{D} = \frac{\frac{\mathcal{D}'}{\text{conn}(A, B)}}{\text{path}(A, B)} \text{ DIR}$$

for some derivation \mathcal{D}' .

By following the same reasoning for the previous case, we know that \mathcal{D}' and \mathcal{D} are TRANS-free.

Case \mathcal{D} ends with an application of rule TRANS:

$$\mathcal{D} = \frac{\frac{\mathcal{E}}{\text{path}(A, B)} \quad \frac{\mathcal{F}}{\text{path}(B, C)}}{\text{path}(A, C)} \text{ TRANS}$$

for some derivations \mathcal{E} and \mathcal{F} .

This is the most interesting case. By induction hypothesis, we know that there exists a TRANS-free derivation \mathcal{E}' such that $\text{path}(A, B)$.

There are only two possibilities for how \mathcal{E}' ends: either with an application of rule DIR or with an application of rule STEP. This gives us two subcases.

Case For some \mathcal{E}'' ,

$$\mathcal{E}' = \frac{\frac{\mathcal{E}''}{\text{conn}(A, B)}}{\text{path}(A, B)} \text{ DIR} .$$

The thesis follows from the derivation

$$\frac{\frac{\mathcal{E}''}{\text{conn}(A, B)} \quad \frac{\mathcal{F}}{\text{path}(B, C)}}{\text{path}(A, C)} \text{ STEP} .$$

Case For some B' , \mathcal{E}_1 and \mathcal{E}_2 ,

$$\mathcal{E}' = \frac{\frac{\mathcal{E}_1}{\text{conn}(A, B')} \quad \frac{\mathcal{E}_2}{\text{path}(B', B)}}{\text{path}(A, B)} \text{ STEP} .$$

Consider the following derivation \mathcal{G} .

$$\mathcal{G} \triangleq \frac{\frac{\mathcal{E}_2}{\text{path}(B', B)} \quad \frac{\mathcal{F}}{\text{path}(B, C)}}{\text{path}(B', C)} \text{ TRANS}$$

The derivation \mathcal{G} is smaller than \mathcal{D} : \mathcal{E}_2 is part of \mathcal{E}' , which by induction hypothesis is not bigger than \mathcal{E} ; and \mathcal{F} is part of \mathcal{D} . Thus, we can invoke the induction hypothesis on \mathcal{G} and get that there exists a

TRANS-free derivation \mathcal{G}' with the same conclusion: $\text{path}(B', C)$.

The thesis now follows by:

$$\frac{\frac{\mathcal{E}_1}{\text{conn}(A, B')} \quad \frac{\mathcal{G}'}{\text{path}(B', C)}}{\text{path}(A, C)} \text{ STEP} .$$

1.4. Rule derivability and admissibility

□

Example 1. *To the reader not familiar with this kind of proofs, the last case might look like a bit of a magic trick! How are we manipulating the derivation, exactly?*

Let us look at a concrete, nontrivial example of a proof \mathcal{D} containing applications of TRANS (to help readability, we use the same derivation names as in the proof of theorem 1):

$$\mathcal{D} \triangleq \frac{\frac{\mathcal{E}}{\text{path}(\text{Odense}, \text{Paris})} \quad \frac{\mathcal{F}}{\text{path}(\text{Paris}, \text{Lisbon})}}{\text{path}(\text{Odense}, \text{Lisbon})} \text{ TRANS}$$

where

$$\mathcal{E} \triangleq \frac{\frac{\frac{\text{conn}(\text{Odense}, \text{Munich})}{\text{path}(\text{Odense}, \text{Munich})} \text{ DIR} \quad \frac{\frac{\text{conn}(\text{Paris}, \text{Munich})}{\text{conn}(\text{Munich}, \text{Paris})} \text{ SYM} \quad \frac{\text{conn}(\text{Munich}, \text{Paris})}{\text{path}(\text{Munich}, \text{Paris})} \text{ DIR}}{\text{path}(\text{Odense}, \text{Paris})} \text{ TRANS} .$$

$$\mathcal{F} \triangleq \frac{\text{conn}(\text{Paris}, \text{Lisbon})}{\text{path}(\text{Paris}, \text{Lisbon})} \text{ DIR}$$

We now “run” the proof of theorem 1 on \mathcal{D} . We are clearly in the last case, so we start by obtaining a TRANS-free derivation \mathcal{E}' that has the same conclusion as \mathcal{E} :

$$\mathcal{E}' = \frac{\frac{\text{conn}(\text{Odense}, \text{Munich})}{\text{path}(\text{Odense}, \text{Paris})} \quad \frac{\frac{\text{conn}(\text{Paris}, \text{Munich})}{\text{conn}(\text{Munich}, \text{Paris})} \text{ SYM} \quad \frac{\text{conn}(\text{Munich}, \text{Paris})}{\text{path}(\text{Munich}, \text{Paris})} \text{ DIR}}{\text{path}(\text{Odense}, \text{Paris})} \text{ STEP}$$

\mathcal{E}' is not bigger than \mathcal{E} (it is actually smaller), as expected. We have that

$$B' = \text{Munich}$$

$$\mathcal{E}'_1 = \text{conn}(\text{Odense}, \text{Munich})$$

$$\mathcal{E}'_2 = \frac{\frac{\text{conn}(\text{Paris}, \text{Munich})}{\text{conn}(\text{Munich}, \text{Paris})} \text{ SYM} \quad \frac{\text{conn}(\text{Munich}, \text{Paris})}{\text{path}(\text{Munich}, \text{Paris})} \text{ DIR}}$$

$$\mathcal{G} = \frac{\frac{\mathcal{E}'_2}{\text{path}(\text{Munich}, \text{Paris})} \quad \frac{\mathcal{F}}{\text{path}(\text{Paris}, \text{Lisbon})}}{\text{path}(\text{Munich}, \text{Lisbon})} \text{ TRANS}$$

$$\begin{array}{c}
 \overline{\text{conn}(\text{Odense}, \text{Munich})} \quad \overline{\text{conn}(\text{Munich}, \text{Rome})} \quad \overline{\text{conn}(\text{Paris}, \text{Munich})} \\
 \\
 \overline{\text{conn}(\text{Paris}, \text{Lisbon})} \\
 \\
 \frac{\text{conn}(A, B)}{\text{conn}(B, A)} \text{SYM} \\
 \\
 \frac{\text{conn}(A, B)}{\text{path}(A, B, 1)} \text{DIRW} \quad \frac{\text{conn}(A, B) \quad \text{path}(B, C, n)}{\text{path}(A, C, n+1)} \text{STEPW}
 \end{array}$$

Figure 1.5: A weighted version of the system in fig. 1.4.

To obtain \mathcal{G}' , we apply recursively the same reasoning to \mathcal{G} , obtaining:

$$\mathcal{G}' = \frac{\overline{\text{conn}(\text{Munich}, \text{Paris})} \quad \frac{\overline{\text{conn}(\text{Paris}, \text{Munich})} \quad \overline{\text{conn}(\text{Munich}, \text{Paris})}}{\text{path}(\text{Munich}, \text{Paris})} \text{DIR}}{\text{path}(\text{Munich}, \text{Lisbon})} \text{STEP} \cdot \text{SYM}$$

The final TRANS-free result is:

$$\mathcal{D}' = \frac{\overline{\text{conn}(\text{Odense}, \text{Munich})} \quad \overline{\text{path}(\text{Munich}, \text{Lisbon})}}{\text{path}(\text{Odense}, \text{Lisbon})} \text{STEP} \cdot \mathcal{G}'$$

Exercise 7 (!). Prove that if $\text{path}(A, B)$ is derivable in the system in fig. 1.4, then there exists a nonempty sequence of derivable propositions $\text{conn}(C_1, C'_1), \dots, \text{conn}(C_n, C'_n)$ for some C_1, \dots, C_n such that $n > 0$, $C_1 = A$ (the sequence starts from A), and $C'_n = B$ (the sequence ends at B).

Prove that if $\text{path}(A, B)$ is derivable in the system in fig. 1.1, then there exists a nonempty sequence of derivable propositions $\text{conn}(C_1, C'_1), \dots, \text{conn}(C_n, C'_n)$ for some C_1, \dots, C_n such that $n > 0$, $C_1 = A$ (the sequence starts from A), and $C'_n = B$ (the sequence ends at B).

Exercise 8. Prove that, for any A and B , if $\text{path}(A, B)$ is derivable in the system in fig. 1.4 then there exists n such that $\text{path}(A, B, n)$ is derivable in the system in fig. 1.5.

1.4. Rule derivability and admissibility

Exercise 9. *Prove that, for any A and B , if $\text{path}(A, B)$ is derivable in the system in fig. 1.1 then there exists n such that $\text{path}(A, B, n)$ is derivable in the system in fig. 1.5.*

Hint: Use theorem 1 and the result of exercise 8.

Chapter 2

Simple choreographies

Now that we have familiarised ourselves with formal systems based on inference rules, we can proceed to using them for the study of concurrency. We start in this chapter by building our first, and very simple, choreography model. Our aim is to design the simplest possible choreography language that captures the essence of what a choreography model is and how we can use it. We will add further features later on, by building on the basic concepts that we shall establish in this chapter. In a sense, you could consider the material in this chapter a sort of technical preliminaries.

The cornerstone of our study will be the notion of *process*, an independent agent that can perform local computation and communicate with other agents by means of message passing (Input/Output, or I/O for short). Processes are abstract representations of computer programs executed concurrently, each possessing an independent control state and memory. Essentially, what we are going to do is to use inference systems to model concurrent systems that consist of processes communicating with each other.

Example 2. *As guiding example for this chapter, suppose that we want to define a system that consists of two processes, called Buyer and Seller. Suppose also that we want these two processes to interact as follows:*

1. *Buyer sends the title of a book she wishes to buy to Seller;*
2. *Seller replies to Buyer with the price of the book.*

The description above is informal. However, it gives us some important indications on how a mathematical formalism for choreographies might look like: our description talks about *multiple* processes and how they interact. We are adopting a global view on all the interactions among the processes that we are interested in. More specifically: each step of our protocol talks about *both* the sender and the

$$C ::= p \rightarrow q; C \mid 0$$

Figure 2.1: Simple choreographies, syntax.

receiver of the communication; and we are explicitly ordering communications (as in the “Alice and Bob notation” from the Preface).

2.1 Syntax

Our first choreography model is called simple choreographies. We start by formalising its *syntax*, which defines the set of (choreographic) programs that can be written in this model.

We refer to processes by using process names. Let Pid be an infinite set of *process names*, ranged over by p, q, \dots, t . We call a process names also *process identifier*, pid for short, recalling the nomenclature from operating systems. The syntax of simple choreographies is given by the grammar in fig. 2.1, where C ranges over a choreography. Let $SimpleChor$ be the language of that grammar, i.e., the set of its derivable terms. In other words, $SimpleChor$ is the set of all simple choreographies. From the grammar, we can see that there are two forms that a choreography can take.

- The empty choreography 0 (also called the terminated choreography), which describes a terminated protocol with no further actions to perform. You can think of it as the end of a choreographic program.
- An interaction term $p \rightarrow q; C$, which means that a message is communicated from process p to process q . After this interaction is performed, the choreography proceeds as defined by the continuation C .

The syntax of simple choreographies is minimalistic, and an interaction between two processes is meant as an atomic action—communications are synchronous.¹ We assume that communications are always between different processes, i.e., whenever we write $p \rightarrow q$, we require $p \neq q$.

Example 3. *The following choreography defines the behaviour that we informally described in example 2.*

Buyer \rightarrow Seller; Seller \rightarrow Buyer; 0

¹We shall see how to deal with asynchronous communications later, in ??.

2.2. Semantics

Note that what we have is actually a rather coarse abstraction of what we described in example 2, because we are not formalising what is being sent from a process to another. For example, the informal description stated that Buyer sends “the title of a book she wishes to buy” to Seller in the first interaction, but our choreography above does not define this part. It simply states that Buyer sends some unspecified message to Seller, and that Seller replies to Buyer afterwards. We are going to add the possibility to specify the content of messages later on.

2.2 Semantics

Now that we can write simple choreographies, we give them a semantics in terms of an operational interpretation. The objective is to formalise abstractly the execution of a choreography. We use one of the most established approaches to define this kind of interpretation (perhaps *the* most established approach), called labelled transition systems.

Definition 4 (Labelled transition system). *A labelled transition system (lts) is a triple (S, L, \longrightarrow) where S is the set of states, L is the set of labels, and $\longrightarrow \subseteq S \times L \times S$ is the transition relation.*

We will range over labels with μ, μ' , etc. Intuitively, labels represent what we can observe about the step from one state into another. For this reason, they are also sometimes called *actions* or *observables*. We adopt the standard shorthand notation for transitions: we write $s_1 \xrightarrow{\mu} s_2$ whenever $(s_1, \mu, s_2) \in \longrightarrow$, for some states s_1 and s_2 in S and label μ in L .

To define an lts for simple choreographies, we follow the structural operational semantics approach by Plotkin [2004]: we define the behaviour of a (choreographic) program in terms of the behaviours of its parts.

Definition 5 (Lts for simple choreographies). *The lts of simple choreographies is the lts (S, L, \longrightarrow) , where:*

- *the set of states S is the set of all choreographies, i.e., $S = \text{SimpleChor}$;*
- *the set of labels L is defined as the set of all possible communications between any two processes, i.e., $L = \{p \rightarrow q \mid p, q \in \text{Pid}\}$;*
- *the transition relation \longrightarrow is the smallest relation satisfying the rule in fig. 2.2.*

Formally, the transition relation \longrightarrow of simple choreographies is defined as the smallest relation satisfying the rule displayed in fig. 2.2. There is only one rule,

$$\frac{}{p \rightarrow q; C \xrightarrow{p \rightarrow q} C} \text{COM}$$

Figure 2.2: Simple choreographies, semantics.

called COM, which is an axiom: it always allows us to execute interactions—if a programmer wishes for an interaction to take place, it always will. In the rule, p , q , and C are all schematic variables, on which we impose no conditions. So the rule works for all process names and choreographies. (Recall, however, that we assumed $p \neq q$ in communication terms, so this is assumed also here.)

Example 4. Let C be the program from example 3:

$$C \triangleq \text{Buyer} \rightarrow \text{Seller}; \text{Seller} \rightarrow \text{Buyer}; 0.$$

We have the following transition

$$C \xrightarrow{\text{Buyer} \rightarrow \text{Seller}} \text{Seller} \rightarrow \text{Buyer}; 0$$

which formally tells us that C can execute a communication from Buyer to Seller (from the label $\text{Buyer} \rightarrow \text{Seller}$). The transition transforms the program into its continuation, from which we can observe a further transition:

$$\text{Seller} \rightarrow \text{Buyer}; 0 \xrightarrow{\text{Seller} \rightarrow \text{Buyer}} 0.$$

There are no further transitions, since we reached term 0, for which there are no transition rules.

So the execution of C is that we first have a communication from Buyer to Seller and then a communication from Seller to Buyer, which is exactly the communication flow that we wanted in example 2.

Conventions on labelled transition systems for programs In this chapter, we have defined all the components of the LTS of simple choreographies explicitly: the set of states, the set of labels, and the transition relation. However, once a grammar like that in fig. 2.1 and a set of inference rules for the transition relation like that in fig. 2.2 are given, this level of detail is unnecessary with a few conventions. These conventions will save us quite a bit of ink in the next chapters, since we will update the syntax and semantics of choreographies a few times to explore different directions and make them more powerful.

First, we can just assume that the set of states of a choreography model is the language of its grammar. Second, let the set of labels for a choreography model be the union of all labels that can be instantiated by replacing the schematic variables

2.2. Semantics

in the inference system for its semantics. Third, let the transition relation always be the smallest relation satisfying the rules of the inference system given for the choreography model.

With these conventions in place, we just need to give the syntax and transition inference system for a choreography model and we will immediately know its intended lts. For example, for simple choreographies, we would just need to present the syntax in fig. 2.1 and the rules in fig. 2.2. We follow these conventions in the remainder.

Other notations and terminologies for labelled transition systems We define some useful notations and terminologies for labelled transition systems, adapting some from Sangiorgi [2011].

Whenever $C \xrightarrow{\mu} C'$, we say that C' is a μ -derivative of C , or sometimes simply a derivative of C .

It can be useful to combine transitions. Let $\vec{\mu}$ be a shorthand for a sequence of labels μ_1, \dots, μ_n , for some n ; then $C \xrightarrow{\vec{\mu}} C'$ holds whenever there exist C_1, \dots, C_{n-1} such that $C \xrightarrow{\mu_1} C_1 \dots C_{n-1} \xrightarrow{\mu_n} C'$. When $C \xrightarrow{\vec{\mu}} C'$ for some choreographies C and C' and sequence of labels $\vec{\mu}$, we say that C' is a derivative of C under $\vec{\mu}$, or simply a *multi-step derivative* of C . We also write $C \xrightarrow{\vec{\mu}}^\mu C'$ whenever there exists C'' such that $C \xrightarrow{\vec{\mu}} C''$ and $C'' \xrightarrow{\mu} C'$.

We write $C \xrightarrow{\mu}$ (read “ C can make a transition with label μ ”) whenever there exists some C' such that $C \xrightarrow{\mu} C'$. Likewise, we write $C \not\xrightarrow{\mu}$ (read “ C cannot make a transition with label μ ”) whenever there exists no C' such that $C \xrightarrow{\mu} C'$.

For example, $0 \not\xrightarrow{\mu}$ for any label μ , which formalises that 0 represents the terminated choreography.

Example 5. With our new notation in place, we can show a complete execution trace of the choreography C from example 3 in a single shot. Recall the choreography:

$$C \triangleq \text{Buyer} \rightarrow \text{Seller}; \text{Seller} \rightarrow \text{Buyer}; 0.$$

We have the following transitions:

$$C \xrightarrow{\text{Buyer} \rightarrow \text{Seller} \quad \text{Seller} \rightarrow \text{Buyer}} 0.$$

So we first have an interaction where Buyer sends a message to Seller and then we have an interaction where Seller sends a message to Buyer, which is exactly the communication flow that we wanted in example 2. Thus, the formal semantics of C matches our informal description.

Proposition 1 (Strong termination for simple choreographies). *For any simple choreography C such that $C \neq \mathbf{0}$ (C is not the terminated choreography), there exists a sequence of labels $\vec{\mu}$ such that $C \xrightarrow{\vec{\mu}} \mathbf{0}$.*

Exercise 10 (\hookrightarrow). *Prove proposition 1. Hint: proceed by induction on the structure of the choreography C .*

Chapter 3

From choreographic programs to process programs

Simple choreographies is like a high-level programming language, providing us with a useful abstraction (the communication term) to talk about what we are interested in. In our case, what we are interested in is defining the interactions that we want to take place among our processes—we want to write protocols. However, high-level programs are not particularly useful if we do not know how they can be implemented in practice. Other high-level languages face the same situation: they offer useful abstractions, like functions and objects, but their programs need to be converted to something that the computer can actually execute, like machine code. Here, we are not interested in reaching the detail of machine code, but in bridging the conceptual difference between choreographies and the typical programs that can be executed in concurrent systems.

In a concurrent and distributed system, each process has its own program that is run by the computer that hosts it. To communicate, processes can then send and receive messages to/from each other. A process does not (necessarily) know what programs the other processes are running, only its own. This is different from choreographies, where the behaviour of multiple processes is defined from a global viewpoint.

Example 6. *To implement our scenario from example 2 following the standard methodology for programming concurrent and distributed systems, we would have to produce two programs: one for process Buyer and one for process Seller.*

Informally, a program for Buyer could look as follows.

- *Send the title of the book to buy to Seller;*
- *receive the price of the book from Seller.*

For Seller, we could use the following (abstract) program.

$$P, Q, R ::= p!; P \mid p?; P \mid 0$$

Figure 3.1: Simple processes, syntax.

- *Receive the title of the book from Buyer;*
- *send the price of the book to Buyer.*

In a nutshell, choreographies give us a global view on the communications to be enacted by the system; whereas realistic concurrent and distributed systems expect to have a program for each process (we call these *process programs*), based on the local view of that process and using send and receive actions (also called Input/Output, or I/O) to interact with the other processes.

Thus, if we want to make choreographies useful, we need the following two elements.

- A formal model for process programs, or *process model*.
- A way to relate our choreography model (simple choreographies) to the process model or, more concretely, choreography programs to process programs.

3.1 Simple processes

In this section we define a simple process model to describe system implementations. We will use it later to build a notion of *correspondence* between what should happen (given as a choreography) and what the system actually does (given as a term in this process model).

We assume that each process is uniquely identified by its name on the network, and that process names can be used to direct messages from one process to each other (similarly to URLs on the web). based on actors.

3.1.1 Syntax

The syntax of simple processes is given in fig. 3.1. A process term P can be:

- a send action $p!; P$, read “send a message to process p and then do P ”;
- a receive action $p?; P$, read “receive a message from process p and then do P ”;

3.1. Simple processes

- or the inactive process 0 .

We write *SimpleProc* for the set of all process terms in simple processes. We call process terms also process programs, or simply processes.

Notice that we are using 0 to represent both inactive choreographies and processes, so our notation is overloaded. This is intuitive because they are both “no-op” terms, i.e., they perform no actions.

We model systems where processes can interact as functions that map process names to process terms, called *networks*. A preliminary ingredient is required to reach the formal definition. Let N be a function from process names to terms:

$$N : Pid \longrightarrow SimpleProc .$$

The *support* of N , written $\text{supp}(N)$, is the subset of the domain Pid containing those process names that are not mapped to 0 ; in set-builder notation:

$$\text{supp}(N) = \{p \mid N(p) \neq 0\} .$$

We say that a function N has *finite support* if the set $\text{supp}(N)$ is finite.

Definition 6 (Network). *A network N is a function with finite support from process names to process terms:*

$$N : Pid \longrightarrow SimpleProc .$$

Notation for networks Since networks have finite support, we can define any network precisely using a finite representation. To do this, we introduce a notation inspired by other works on process models [Milner 1980, Hennessy 2007]. Intuitively, we write $p[P]$ for “process p runs the program P ”, and use the *parallel operator* “ $|$ ” to compose such processes in systems where all processes run in parallel.

Formally, let $p \in Pid$. Then, we write $p[P]$ for the network N such that

$$N(q) = \begin{cases} P & \text{if } q = p \\ 0 & \text{otherwise} \end{cases}$$

Given any two sets S_1 and S_2 , we write $S_1 \# S_2$ whenever $S_1 \cap S_2 = \emptyset$, that is, the two sets are disjoint (they do not share any elements). Let N and M be networks with disjoint supports, i.e., $\text{supp}(N) \# \text{supp}(M)$. Then, we write $N \mid M$ for the network obtained by composing N and M :

$$(N \mid M)(p) = \begin{cases} N(p) & \text{if } p \in \text{supp}(N) \\ M(p) & \text{otherwise} \end{cases} .$$

Chapter 3. From choreographic programs to process programs

The *terminated network* $\mathbf{0}$ is the network such that:

$$\mathbf{0}(p) = \mathbf{0} \quad \text{for all } p \in \text{Pid}.$$

Again, there is no ambiguity in using the same symbol $\mathbf{0}$, as we did for the terminated choreography and process term, since the context will always clarify what we are referring to. On the other hand, $\mathbf{0}$ gives us the useful mnemonic of an “inactive” object.

Example 7. *The two process programs informally described in example 6 can be formalised as the following network.*

$$\text{Buyer}[\text{Seller!}; \text{Seller?}; \mathbf{0}] \mid \text{Seller}[\text{Buyer?}; \text{Buyer!}; \mathbf{0}]$$

Exercise 11 (Unit of parallel composition). *Prove that $\mathbf{0}$ is the unit of parallel composition, that is, that for all N ,*

$$\mathbf{0} \mid N = N.$$

Exercise 12 (Commutativity of parallel composition). *Prove that the parallel operator is commutative, that is, for all N and M ,*

$$N \mid M = M \mid N.$$

Hint: Recall that $N \mid M$ is defined only when $\text{supp}(N) \# \text{supp}(M)$.

Exercise 13 (Associativity of parallel composition). *Prove that the parallel operator is associative, that is, for all N_1 , N_2 , and N_3 ,*

$$N_1 \mid (N_2 \mid N_3) = (N_1 \mid N_2) \mid N_3.$$

3.1.2 Semantics

Similarly to what we have done for simple choreographies, we equip simple processes with a labelled transition system semantics. The idea is that each transition models a step in the execution of a network.

Formally, the lts of simple processes is given by the inference system displayed in fig. 3.2—recall our convention for defining an lts from an inference system from section 2.2. There are two rules. The first rule, called COM, is an axiom:

$$\frac{}{p[q!; P] \mid q[p?; Q] \xrightarrow{p \rightarrow q} p[P] \mid q[Q]} \text{COM}.$$

Rule COM matches a send action from a process p to a process q with the compatible receive action by process q from p . The two processes then proceed with

3.1. Simple processes

$$\frac{}{p[q!; P] \mid q[p?; Q] \xrightarrow{p \rightarrow q} p[P] \mid q[Q]} \text{COM} \quad \frac{N \xrightarrow{\mu} N'}{N \mid M \xrightarrow{\mu} N' \mid M} \text{PAR}$$

Figure 3.2: Simple processes, semantics.

their respective continuations. The second rule, called PAR, states that if a part of a network can perform a transition, then this can be observed also in a larger context:

$$\frac{N \xrightarrow{\mu} N'}{N \mid M \xrightarrow{\mu} N' \mid M} \text{PAR}.$$

Observe that the part of the network not affected by the transition remains (M) remains unaffected. Also, since the parallel operator for networks is commutative and associative (exercise 12 and exercise 13), the rule supports performing transitions in any part of the network.

Remark 1. We use the same name (COM) for the communication rules of both choreographies and processes, as a mnemonic that the two rules intuitively model the same concept. This causes no ambiguity, since the two rules act on different syntactic terms, so their usages will be clear from the context.

Example 8. Let N be the network from example 7:

$$\text{Buyer}[\text{Seller}!; \text{Seller}?; 0] \mid \text{Seller}[\text{Buyer}?; \text{Buyer}!; 0].$$

We can observe the intended communication between Buyer and Seller by rule COM:

$$\frac{\text{Buyer}[\text{Seller}!; \text{Seller}?; 0] \mid \text{Seller}[\text{Buyer}?; \text{Buyer}!; 0] \xrightarrow{\text{Buyer} \rightarrow \text{Seller}} \text{Buyer}[\text{Seller}?; 0] \mid \text{Seller}[\text{Buyer}!; 0]}{\text{COM}}.$$

Let N' be the derivative above. We can proceed further and observe that N' can terminate by performing the expected second communication.

$$\frac{N' \xrightarrow{\text{Seller} \rightarrow \text{Buyer}} \text{Seller}[0] \mid \text{Buyer}[0]}{\text{COM}}.$$

Observe that the derivative above is equal to the terminated network, 0 :

$$\text{Seller}[0] \mid \text{Buyer}[0] = 0.$$

In other words, we have that 0 is a derivative of N under $\text{Buyer} \rightarrow \text{Seller}$, $\text{Seller} \rightarrow \text{Buyer}$, or more concisely:

$$N \xrightarrow{\text{Buyer} \rightarrow \text{Seller}} \xrightarrow{\text{Seller} \rightarrow \text{Buyer}} 0.$$

Chapter 3. From choreographic programs to process programs

Example 9. Let N be the following network of three processes, for some p , q , and r . The idea is that q forwards a message from p to r .

$$N \triangleq p[q!; 0] \mid q[p?; r!; 0] \mid r[q?; 0]$$

We can derive the first communication from p to q by using rule PAR:

$$\frac{\frac{p[q!; 0] \mid q[p?; r!; 0] \xrightarrow{p \rightarrow q} q[r!; 0]}{p[q!; 0] \mid q[p?; r!; 0] \mid r[q?; 0] \xrightarrow{p \rightarrow q} q[r!; 0] \mid r[q?; 0]} \text{PAR}}{\text{COM}}$$

By rule COM, the derivative above can terminate.

$$\frac{q[r!; 0] \mid r[q?; 0] \xrightarrow{q \rightarrow r} 0}{\text{COM}}$$

Hence, $N \xrightarrow{p \rightarrow q} \xrightarrow{q \rightarrow r} 0$.

Exercise 14. Let N be defined as:

$$N \triangleq p[q?; r?; r!; 0] \mid q[p!; 0] \mid r[p!; p?; s!; 0] \mid s[r?; 0].$$

Prove that there exists a sequence of labels $\vec{\mu}$ such that $N \xrightarrow{\vec{\mu}} 0$.

Hint: use the commutativity and associativity properties of parallel composition to apply rule PAR correctly in your derivations.

Differently from choreographies, it is possible to write networks that can get “stuck”, in the sense that they may not terminate. In fact, there is no equivalent of proposition 1 for simple processes, since there exist networks that might not terminate.

Exercise 15. Prove the following propositions.

1. $p[q!; 0] \not\xrightarrow{\mu}$ for any μ .
2. $p[q!; 0] \mid q[p!; 0] \not\xrightarrow{\mu}$ for any μ .
3. There is no $\vec{\mu}$ such that

$$p[q?; r!; q?; 0] \mid q[p!; p!; 0] \mid r[q?; p?; 0] \xrightarrow{\vec{\mu}} 0.$$

Lemma 1. For any N , μ , and N' , if $N \xrightarrow{\mu} N'$ then $N(r) = N'(r)$ for all r such that $r \notin \text{pn}(\mu)$.

Exercise 16 (\Leftarrow). Prove lemma 1.

Hint: Proceed by induction on the derivation of the transition $N \xrightarrow{\mu} N'$.

3.2. Endpoint Projection (EPP)

3.2 Endpoint Projection (EPP)

The network in example 7 works as intended, but we had to come up with it manually. If we could figure out a mechanical method of going from a choreography (which formalises what we want) to a network (which formalises an implementation), we would save time. If we could also prove that such method *always gives us a correct result*, we would also save ourselves the potential mistakes that come from the manual activity of writing a process network that should implement what we want.

3.2.1 From choreographies to processes

An operational intuition To gain some intuition on how we could develop a method to translate choreographies into processes, we can look at our examples. Let C be the choreography from example 3, i.e.,

$$C \triangleq \text{Buyer} \rightarrow \text{Seller}; \text{Seller} \rightarrow \text{Buyer}; 0 ,$$

and let N be the network from example 7, i.e.,

$$N \triangleq \text{Buyer}[\text{Seller}!; \text{Seller}?; 0] \mid \text{Seller}[\text{Buyer}?; \text{Buyer}!; 0] .$$

Intuitively, N is a correct process implementation of C , in the sense that N performs exactly the communications prescribed by C . We can check for this by looking at the respective transitions of the choreography and the network. The only transition that C can perform is

$$C \xrightarrow{\text{Buyer} \rightarrow \text{Seller}} \text{Seller} \rightarrow \text{Buyer}; 0 ,$$

and the only transition that N can perform is

$$\frac{\text{Buyer}[\text{Seller}!; \text{Seller}?; 0] \mid \text{Seller}[\text{Buyer}?; \text{Buyer}!; 0]}{\text{Buyer}[\text{Seller}?; 0] \mid \text{Seller}[\text{Buyer}!; 0]} \xrightarrow{\text{Buyer} \rightarrow \text{Seller}} .$$

These two transitions have the same label, so they correspond to each other. If we look at the derivatives of C and N , they have just one possible transition each, which also correspond to each other:

$$\frac{\text{Seller} \rightarrow \text{Buyer}; 0}{\text{Buyer}[\text{Seller}?; 0] \mid \text{Seller}[\text{Buyer}!; 0]} \xrightarrow{\text{Seller} \rightarrow \text{Buyer}} 0 .$$

Now the choreography has terminated and the network cannot make any further transitions with observable labels.

$$\llbracket p \rightarrow q; C \rrbracket_r = \begin{cases} q!; \llbracket C \rrbracket_r & \text{if } r = p \\ p?; \llbracket C \rrbracket_r & \text{if } r = q \\ \llbracket C \rrbracket_r & \text{otherwise} \end{cases}$$

$$\llbracket 0 \rrbracket_p = 0$$

Figure 3.3: Process projection for simple choreographies.

What we learn from this example is that a network “implements” a choreography correctly if the two can “mimic” each other transitions, using the same labels. In other words, executing a network that implements a choreography correctly gives rise to the interactions described in the choreography.

Formal definition We can now move to formally defining our desired method for translating choreographies into correct process implementations, as a function from choreographies to networks. This function is commonly called *EndPoint Projection* (EPP for short), since it projects each interaction in the choreography to the local action that each process (an endpoint) should perform in the network [Qiu et al. 2007, Lanese et al. 2008, Carbone et al. 2012]. Indeed, we can think of an interaction like Buyer \rightarrow Seller as consisting of two parts, i.e., the send action by Buyer and the receive action by Seller. So the send action that the process implementing Buyer should perform is the first component of the interaction, and the receive action by Seller is the second component.

The key part of the definition of EPP is a function called *process projection*, written $\llbracket \bullet \rrbracket_\bullet$. Intuitively, given a choreography C and a process name p , $\llbracket C \rrbracket_p$ returns the process term that defines the actions that p should execute in order to play its part as specified by C . Process projection is defined by structural recursion on choreographies, as follows.

Definition 7 (Process projection). *Let C be a choreography and p a process name ($p \in \text{Pid}$). The projection of C for p , written $\llbracket C \rrbracket_p$, is defined by the rules given in fig. 3.3.*

Example 10. *Let $C_{\text{BuyerSeller}}$ be the choreography in example 3:*

$$C_{\text{BuyerSeller}} \triangleq \text{Buyer} \rightarrow \text{Seller}; \text{Seller} \rightarrow \text{Buyer}; 0.$$

If we construct a network consisting of the process projections for Buyer and Seller, we obtain the correct implementation of $C_{\text{BuyerSeller}}$ that we have manually written before:

$$\text{Buyer}[\llbracket C \rrbracket_{\text{Buyer}}] \mid \text{Seller}[\llbracket C \rrbracket_{\text{Seller}}] = \text{Buyer}[\text{Seller!}; \text{Seller?}; 0] \mid \text{Seller}[\text{Buyer?}; \text{Buyer!}; 0].$$

3.2. Endpoint Projection (EPP)

The definition of EPP is obtained by generalising the idea to map each process name to the related process projection for the originating choreography. We write $\llbracket C \rrbracket$ for the EPP of a choreography C . Formally, $\llbracket \bullet \rrbracket$ is a function from choreographies to networks defined as follows.

Definition 8 (EndPoint Projection (EPP)). *Let C be a choreography. The endpoint projection (EPP) of C , written $\llbracket C \rrbracket$, is the network such that, for all $p \in \text{Pid}$,*

$$\llbracket C \rrbracket(p) = \llbracket C \rrbracket_p .$$

Example 11. *Let $C_{\text{BuyerSeller}}$ be the choreography in example 3:*

$$C_{\text{BuyerSeller}} \triangleq \text{Buyer} \rightarrow \text{Seller}; \text{Seller} \rightarrow \text{Buyer}; 0 .$$

The EPP of $C_{\text{BuyerSeller}}$ is exactly the network that we defined in example 7:

$$\llbracket C_{\text{BuyerSeller}} \rrbracket = \text{Buyer}[\text{Seller}!; \text{Seller}?; 0] \mid \text{Seller}[\text{Buyer}?; \text{Buyer}!; 0] .$$

Exercise 17. *Write the EPP of the choreography*

$$p \rightarrow q; r \rightarrow q; r \rightarrow s; q \rightarrow p; 0 .$$

A sanity check on EPP Definition 8 claims that EPP produces a network, and definition 6 requires every network to have finite support. Thus, as a sanity check, we should check that $\llbracket C \rrbracket$ has finite support for all choreographies C .

By definition of support, the support of $\llbracket C \rrbracket$ is defined as

$$\text{supp}(\llbracket C \rrbracket) = \{p \mid \llbracket C \rrbracket(p) \neq 0\} .$$

Since $\llbracket C \rrbracket(p) = \llbracket C \rrbracket_p$, we can rewrite the equation above as

$$\text{supp}(\llbracket C \rrbracket) = \{p \mid \llbracket C \rrbracket_p \neq 0\} .$$

The question is thus whether the set

$$\{p \mid \llbracket C \rrbracket_p \neq 0\}$$

is finite. Intuitively, it is, since a choreography can only mention a finite number of processes. Let us prove this formally.

First, we define a function, pn , that maps each choreography to the set of process names that it mentions. We define pn by recursion on the structure of the choreography, as follows.

$$\begin{aligned} \text{pn}(0) &= \emptyset \\ \text{pn}(p \rightarrow q; C) &= \{p, q\} \cup \text{pn}(C) \end{aligned}$$

Given any choreography C , it is easy to prove that $\text{pn}(C)$ is a finite set.

Chapter 3. From choreographic programs to process programs

Proposition 2. *For any C , $\text{pn}(C)$ is a finite set.*

Proof. By induction on C .

Case $C = 0$. In this case, $\text{pn}(0) = \emptyset$, which is a finite set.

Case $C = p \rightarrow q; C'$ for some p, q , and C' . By induction hypothesis, we know that $\text{pn}(C')$ is a finite set. By definition of pn , $\text{pn}(C) = \{p, q\} \cup \text{pn}(C')$. So $\text{pn}(C)$ is the union of two finite sets, which is also a finite set.

□

Now we prove that, for any C , if a process name does not appear in $\text{pn}(C)$, then $\llbracket C \rrbracket_p = 0$.

Lemma 2. *For any C and p , $p \notin \text{pn}(C)$ implies $\llbracket C \rrbracket_p = 0$.*

Proof. By induction on C .

Case $C = 0$. In this case, the thesis follows trivially because $\llbracket C \rrbracket_p = 0$ for all p .

Case $C = q \rightarrow r; C'$ for some q, r , and C' . By definition of pn ,

$$\text{pn}(C) = \{q, r\} \cup \text{pn}(C').$$

Hence, by the assumption $p \notin \text{pn}(C)$, we have that (1) $p \notin \{q, r\}$ and (2) $p \notin \text{pn}(C')$. From (1) and the definition of EPP, we get $\llbracket C \rrbracket_p = \llbracket C' \rrbracket_p$. From (2), we can apply the induction hypothesis to $\llbracket C' \rrbracket$ and obtain that $\llbracket C' \rrbracket_p = 0$.

□

Corollary 1. *For any C , $\llbracket C \rrbracket$ has finite support.*

Proof. Direct consequence of proposition 2 and lemma 2. □

Exercise 18. *Prove that, for any C , $\text{supp}(\llbracket C \rrbracket) = \text{pn}(C)$.*

Exercise 19 (!). *Another way of proving the result of corollary 1 is to proceed by structural induction on the choreography C and build directly the support of $\llbracket C \rrbracket$. Develop such proof.*

Hint: follow the structure of the proof of proposition 2.

3.2. Endpoint Projection (EPP)

3.2.2 Towards a correct EPP

That EPP works for our simple buyer-seller example is a good sign. But does EPP work *in general*? That is, are all choreographies projected to a correct network implementation that does exactly what the originating choreography prescribes?

Unfortunately, we can easily find a counterexample where the EPP of a choreography performs transitions that are not allowed for by the choreography. The counterexample is simple. Take the following choreography:

$$C_{\text{problem}} \triangleq p \rightarrow q; r \rightarrow s; 0. \quad (3.1)$$

The EPP of this choreography is:

$$\llbracket C_{\text{problem}} \rrbracket = p[q!; 0] \mid q[p?; 0] \mid r[s!; 0] \mid s[r?; 0].$$

We have the following transition for this network, by synchronising r with s :

$$p[q!; 0] \mid q[p?; 0] \mid r[s!; 0] \mid s[r?; 0] \xrightarrow{r \rightarrow s} p[q!; 0] \mid q[p?; 0] \mid r[0] \mid s[0].$$

However, C_{problem} cannot mimic this transition: $C_{\text{problem}} \not\xrightarrow{r \rightarrow s}$. Instead, it can only perform the first interaction between p and q according to the semantics of simple choreographies.

Choreography C_{problem} exemplifies a general problem: our framework is not sound yet, because the projection of a choreography can perform “extra” transitions with respect to the choreography. If we look closely though, the communication $r \rightarrow s$ executed by the EPP of C_{problem} is indeed defined in the choreography; it is just happening *too early*. There are two ways to fix this: forbidding independent sequences of interactions, or allowing for out of order execution in choreographies.

Forbidding independent sequences of interactions One way is to say that choreographies like C_{problem} are “forbidden”, because the sequence of interactions $p \rightarrow q; r \rightarrow s$ is not enforced by any causality relation between the two interactions. More specifically, the processes p , q , r , and s are all different, hence they operate independently—as the transition for the projection of C_{problem} shows. However, if the process names of the two interactions overlapped in any way, this problem would not appear. For example, consider the choreography $p \rightarrow q; r \rightarrow q; 0$. Its projection executes as expected by the choreography because process q necessarily needs to complete the first interaction before participating in the second.

Exercise 20 (\Leftarrow). *Check that the choreography $p \rightarrow q; r \rightarrow q; 0$ can mimic the transitions of its EPP.*

$$C ::= p \rightarrow q; C \mid 0$$

Figure 3.4: Simple concurrent choreographies, syntax.

Detecting sequences of interactions that do not have causal dependencies can be done mechanically. Thus, it is possible to automatically detect whether a choreography respects the condition of not having sequences of independent interactions, as in C_{problem} . Forbidding programmers to write choreographies like C_{problem} was a popular approach (and still is in some works, when useful) in the first studies on choreographies, like [Fu et al. 2005b, Qiu et al. 2007, Carbone et al. 2007].

Out of order execution The other way to solve our issue with sequences of independent interactions is to extend the semantics of choreographies to correctly capture the parallel nature of processes. Going back to C_{problem} again, if we could somehow design a semantics that allowed for the the transition

$$p \rightarrow q; r \rightarrow s; 0 \xrightarrow{r \rightarrow s} p \rightarrow q; 0$$

then we would be fine, because that transition would match the problematic one that the EPP of the choreography can do. Observe that this transition does not respect the order in which instructions are given in the choreography. This kind of semantics is typically called “out-of-order execution”. The idea is widespread in many domains. For example, modern CPUs and/or language runtimes may change the order in which instructions are executed to increase performance, when it is safe to do so—a typical example is the single-threaded imperative code $x++;$ $y++;$, where the order in which the two increments are done is influential and the runtime may thus decide to parallelise it.

Since the inception of out-of-order execution for choreographies [Carbone and Montesi 2013], similar ideas have been adopted in different works [Deniélou and Yoshida 2013, Honda et al. 2016, Cruz-Filipe and Montesi 2019].

In this book, we follow the out of order approach.

3.2.3 Simple Concurrent Choreographies

We update our model of simple choreographies to capture concurrent execution of different processes.

The syntax of choreographies remains unchanged. It is displayed in fig. 3.4.

3.2. Endpoint Projection (EPP)

$$\frac{}{p \rightarrow q; C \xrightarrow{p \rightarrow q} C} \text{COM} \quad \frac{C \xrightarrow{\mu} C' \quad \{p, q\} \# \text{pn}(\mu)}{p \rightarrow q; C \xrightarrow{\mu} p \rightarrow q; C'} \text{DELAY}$$

Figure 3.5: Simple concurrent choreographies, semantics.

The semantics of choreographies (fig. 3.5), instead, has an extra rule:

$$\frac{C \xrightarrow{\mu} C' \quad \{p, q\} \# \text{pn}(\mu)}{p \rightarrow q; C \xrightarrow{\mu} p \rightarrow q; C'} \text{DELAY} .$$

Rule DELAY allows us to *delay* the execution of a communication prescribed by the choreography, and let us observe a transition that comes from its continuation instead, *assuming that such transition does not involve any of the processes in the communication term*. This condition is formalised by the premise $\{p, q\} \# \text{pn}(\mu)$, where function pn is a function from labels to sets of process names defined as follows:

$$\text{pn}(p \rightarrow q) = \{p, q\} .$$

Example 12. *This is the transition we needed for the choreography in section 3.2.2.*

$$p \rightarrow q; r \rightarrow s; 0 \xrightarrow{r \rightarrow s} p \rightarrow q; 0 .$$

We can now perform it with our new semantics. Here is the derivation:

$$\frac{\frac{}{r \rightarrow s; 0 \xrightarrow{r \rightarrow s} 0} \text{COM} \quad \{p, q\} \# \{r, s\}}{p \rightarrow q; r \rightarrow s; 0 \xrightarrow{r \rightarrow s} p \rightarrow q; 0} \text{DELAY} .$$

Exercise 21. *Show all the possible multi-step derivatives of the following choreography.*

$$p \rightarrow q; r \rightarrow s; q \rightarrow r; 0$$

A note on premises: side conditions The reader might be puzzled by the fact that the second premise of rule DELAY, $\{p, q\} \# \text{pn}(\mu)$, has a form for which we have not specified any inference rules. Indeed, in example 12, we have not derived the required premise $\{p, q\} \# \{r, s\}$, but simply stated it.

This kind of premises are sometimes called *side conditions*, since they merely express constraints on the schematic variables (in this case p , q , and μ) of a rule. The rule can be applied only if these constraints are respected. If one goes down this road, then side conditions are not exactly premises “morally”, since they do not require a derivation.

Another way to look at it is that we certainly *could* define an inference system for deriving basic facts such as $\{p, q\} \# \{r, s\}$. It is just that these facts are so elementary that analysing them to such a level of detail is uninteresting for our purposes here (but definitely not in general, and in fact inference systems for this kind of statements do exist).

We take the second viewpoint here. That is, whenever we specify a rule with some premise for which we have not specified an inference system, we are going to assume that such an inference system exists. In our examples, we are simply going to state these premises and assume that there exists a derivation that proves it. We shall adopt this “shortcut” only for basic propositions about sets.

3.3 Correctness of EPP

We are now in possession of all the necessary ingredients to prove that EPP is correct in general, meaning that it yields the right result for all simple concurrent choreographies. Formally, we are going to prove two results.

- (Completeness) For any C , μ , and C' , $C \xrightarrow{\mu} C'$ implies $\llbracket C \rrbracket \xrightarrow{\mu} \llbracket C' \rrbracket$.
- (Soundness) For any C , μ , and N , $\llbracket C \rrbracket \xrightarrow{\mu} N$ implies $C \xrightarrow{\mu} C'$ for some C' such that $N = \llbracket C' \rrbracket$.

Intuitively, the completeness part means that the network generated by the EPP of a choreography does *all* that the choreography says. Conversely, the soundness part means that the network generated by the EPP of a choreography does *only* what the choreography says. The two parts combined give us correctness: the network generated by EPP does exactly what is defined in the originating choreography. So what happens is what we want!

A useful property for our proof will be that process projection remains invariant under orthogonal transitions.

Lemma 3. *If $C \xrightarrow{\mu} C'$, then $\llbracket C \rrbracket_r = \llbracket C' \rrbracket_r$ for all $r \notin \text{pn}(\mu)$.*

Exercise 22 (\leftrightarrow). *Prove lemma 3.*

Hint: Proceed by induction on the derivation of the transition $C \xrightarrow{\mu} C'$.

Notation for the proof Before proceeding to the proof of the theorem, it is convenient to introduce some additional notation on networks.

First, let C be a choreography and $S = \{p_1, \dots, p_n\}$ be a finite set of process names. The *restriction* of a network N to S , written $N|_S$, is defined as:

$$N|_S = p_1[N(p_1)] \mid \dots \mid p_n[N(p_n)] .$$

3.3. Correctness of EPP

We shall write $N|_{\vec{p}}$ as abbreviation for $N|_{\{\vec{p}\}}$ (since the ordering of the sequence \vec{p} does not matter).

We define also a notation to “erase” processes in a network. For any network N and process name p , we write $N \setminus p$ for the network such that:

$$(N \setminus p)(q) = \begin{cases} 0 & \text{if } q = p \\ N(q) & \text{otherwise} \end{cases}.$$

The notation is generalised to sequences of process names in the expected way, that is, for any non-empty sequence \vec{q} :

$$N \setminus p, \vec{q} = (N \setminus p) \setminus \vec{q}.$$

We give \setminus a higher priority than $|$, in the sense that

$$N \setminus \vec{p} | M = (N \setminus \vec{p}) | M$$

for all N , M , and \vec{p} .

The notation supports the following property.

Proposition 3. *For any N and \vec{p} ,*

$$N = (N \setminus \vec{p}) | N|_{\vec{p}}$$

Proposition 3 is going to be useful for focusing on some processes of interest. We are going to use it often in the remainder, sometimes implicitly.

For the upcoming proof specifically, by the definition of EPP, we can reformulate proposition 3 to obtain a useful property about choreography projections.

Proposition 4. *For any C and $\vec{p} = p_1, \dots, p_n$,*

$$\llbracket C \rrbracket = (\llbracket C \rrbracket \setminus \vec{p}) | p_1 \left[\llbracket C \rrbracket_{p_1} \right] | \dots | p_n \left[\llbracket C \rrbracket_{p_n} \right].$$

Exercise 23. *Prove that $N \setminus \text{supp}(N) = 0$.*

Our notation also supports an important semantic property: in network transitions, only the processes mentioned in a label are required.

Lemma 4. *If $N \xrightarrow{\mu} N'$ and $p \notin \text{pn}(\mu)$, then $N \setminus p \xrightarrow{\mu} N' \setminus p$.*

Exercise 24. *Prove lemma 4.*

By iterating applications of lemma 4, one can additionally prove the following result on the minimal network required to perform a given transition.

Corollary 2. *If $N \xrightarrow{\mu} N'$, then $N|_{\text{pn}(\mu)} \xrightarrow{\mu} N'|_{\text{pn}(\mu)}$.*

We can now prove the results about EPP. We prove each one as a separate lemma, and combine these results later in a single theorem.

Chapter 3. From choreographic programs to process programs

The proof We prove completeness first.

Lemma 5 (Completeness of EPP). *For any C , μ , and C' ,*

$$C \xrightarrow{\mu} C' \text{ implies } \llbracket C \rrbracket \xrightarrow{\mu} \llbracket C' \rrbracket .$$

Proof. We proceed by induction on the derivation of the transition $C \xrightarrow{\mu} C'$.

Case The derivation of the transition ends with an application of rule COM:

$$\frac{}{p \rightarrow q; C' \xrightarrow{p \rightarrow q} C'} \text{ COM} .$$

By the definition of EPP,

$$\llbracket C \rrbracket = \llbracket C \rrbracket \setminus p, q \mid p[q!; \llbracket C' \rrbracket_p] \mid q[p?; \llbracket C' \rrbracket_q] .$$

By lemma 3, $\llbracket C \rrbracket \setminus p, q = \llbracket C' \rrbracket \setminus p, q$.

Therefore, we can rewrite $\llbracket C \rrbracket$ from above as:

$$\llbracket C \rrbracket = \llbracket C' \rrbracket \setminus p, q \mid p[q!; \llbracket C' \rrbracket_p] \mid q[p?; \llbracket C' \rrbracket_q] .$$

By rules PAR and COM for networks, $\llbracket C \rrbracket$ can mimic the choreographic transition:

$$\llbracket C \rrbracket \xrightarrow{p \rightarrow q} \llbracket C' \rrbracket \setminus p, q \mid p[\llbracket C' \rrbracket_p] \mid q[\llbracket C' \rrbracket_q] .$$

Notice that the derivative of $\llbracket C \rrbracket$ above is exactly the EPP of C' , hence $\llbracket C \rrbracket \xrightarrow{p \rightarrow q} \llbracket C' \rrbracket$.

Case The derivation of the transition ends with an application of rule DELAY:

$$\frac{C_1 \xrightarrow{\mu} C_2 \quad \{p, q\} \# \text{pn}(\mu)}{p \rightarrow q; C_1 \xrightarrow{\mu} p \rightarrow q; C_2} \text{ DELAY} .$$

By definition of EPP,

$$\llbracket C \rrbracket = \llbracket C_1 \rrbracket \setminus p, q \mid p[q!; \llbracket C_1 \rrbracket_p] \mid q[p?; \llbracket C_1 \rrbracket_q] .$$

By induction hypothesis, $\llbracket C_1 \rrbracket \xrightarrow{\mu} \llbracket C_2 \rrbracket$. Since p and q do not appear in μ (premise of DELAY above), by lemma 4 we obtain $\llbracket C_1 \rrbracket \setminus p, q \xrightarrow{\mu} \llbracket C_2 \rrbracket \setminus p, q$. We use this to make the following derivation:

$$\frac{\frac{\llbracket C_1 \rrbracket \setminus p, q \xrightarrow{\mu} \llbracket C_2 \rrbracket \setminus p, q}{\llbracket C_1 \rrbracket \setminus p, q \mid p[q!; \llbracket C_1 \rrbracket_p] \mid q[p?; \llbracket C_1 \rrbracket_q] \xrightarrow{\mu} \llbracket C_2 \rrbracket \setminus p, q \mid p[q!; \llbracket C_1 \rrbracket_p] \mid q[p?; \llbracket C_1 \rrbracket_q]} \text{ PAR}}{\llbracket C_2 \rrbracket \setminus p, q \mid p[q!; \llbracket C_1 \rrbracket_p] \mid q[p?; \llbracket C_1 \rrbracket_q]} . \quad (3.2)$$

3.3. Correctness of EPP

From the premises of rule DELAY at the beginning and lemma 3, we obtain that $\llbracket C_1 \rrbracket_p = \llbracket C_2 \rrbracket_p$ and $\llbracket C_1 \rrbracket_q = \llbracket C_2 \rrbracket_q$. Therefore the derivative we obtained with rule PAR is equal to $\llbracket p \rightarrow q; C_2 \rrbracket$, and the thesis is proven. We summarise the proof of this case:

$$\begin{aligned}
\llbracket C \rrbracket &= \text{(definition 8)} \\
\llbracket C_1 \rrbracket \setminus p, q \mid p \llbracket q!; \llbracket C_1 \rrbracket_p \rrbracket \mid q \llbracket p?; \llbracket C_1 \rrbracket_q \rrbracket &\xrightarrow{\mu} \text{(eq. (3.2))} \\
\llbracket C_2 \rrbracket \setminus p, q \mid p \llbracket q!; \llbracket C_1 \rrbracket_p \rrbracket \mid q \llbracket p?; \llbracket C_1 \rrbracket_q \rrbracket &= \text{(lemma 3)} \\
\llbracket C_2 \rrbracket \setminus p, q \mid p \llbracket q!; \llbracket C_2 \rrbracket_p \rrbracket \mid q \llbracket p?; \llbracket C_2 \rrbracket_q \rrbracket &= \text{(definition 8)} \\
\llbracket p \rightarrow q; C_2 \rrbracket &.
\end{aligned}$$

□

We now move on to proving the soundness result.

Lemma 6 (Soundness of EPP). *For any C , μ , and N ,*

$$\llbracket C \rrbracket \xrightarrow{\mu} N \text{ implies } C \xrightarrow{\mu} C' \text{ for some } C' \text{ such that } N = \llbracket C' \rrbracket .$$

Proof. We proceed by induction on C and consider all the possible transitions that N can have.

Case $C = 0$. In this case, $\llbracket C \rrbracket = 0$ so there are no transitions to consider.

Case $C = p \rightarrow q; C_1$ for some C_1 . By definition of EPP,

$$\llbracket C \rrbracket = \llbracket C_1 \rrbracket \setminus p, q \mid p \llbracket q!; \llbracket C_1 \rrbracket_p \rrbracket \mid q \llbracket p?; \llbracket C_1 \rrbracket_q \rrbracket .$$

Now, if $\llbracket C \rrbracket \xrightarrow{\mu} N$, we have two cases: either $\mu = p \rightarrow q$ (the transition is the execution of the communication between p and q), or p and q do not appear in μ ($\{p, q\} \# \text{pn}(\mu)$).

In the first case, by the semantics of networks and lemma 1 we obtain

$$N = \llbracket C_1 \rrbracket \setminus p, q \mid p \llbracket \llbracket C_1 \rrbracket_p \rrbracket \mid q \llbracket \llbracket C_1 \rrbracket_q \rrbracket = \llbracket C_1 \rrbracket .$$

Therefore, C can mimic the transition by $\llbracket C \rrbracket$ by rule COM:

$$p \rightarrow q; C_1 \xrightarrow{p \rightarrow q} C_1 .$$

In the second case, by lemma 4 we have that

$$\llbracket C_1 \rrbracket \setminus p, q \xrightarrow{\mu} N \setminus p, q .$$

Chapter 3. From choreographic programs to process programs

From this, we can derive the following transition by using rule PAR:

$$\llbracket C_1 \rrbracket \xrightarrow{\mu} N \setminus p, q \mid p \llbracket C_1 \rrbracket_p \mid q \llbracket C_1 \rrbracket_q .$$

By induction hypothesis, we obtain that there exists C_2 such that

$$C_1 \xrightarrow{\mu} C_2 \text{ such that } \llbracket C_2 \rrbracket = N \setminus p, q \mid p \llbracket C_1 \rrbracket_p \mid q \llbracket C_1 \rrbracket_q . \quad (3.3)$$

Equation (3.3) reveals that $N \setminus p, q = \llbracket C_2 \rrbracket \setminus p, q$. Also, since $\llbracket C \rrbracket \xrightarrow{\mu} N$, lemma 1 tells us that $N \upharpoonright_{p,q} = \llbracket C \rrbracket \upharpoonright_{p,q}$. Therefore, given that $C = p \rightarrow q; C_1$,

$$N = \llbracket C_2 \rrbracket \setminus p, q \mid \llbracket p \rightarrow q; C_1 \rrbracket \upharpoonright_{p,q} . \quad (3.4)$$

We can now move to mimicking the transition on the choreography level. Since we are in the case where p and q do not appear in μ , we can use the transition $C_1 \xrightarrow{\mu} C_2$ from eq. (3.3) to apply rule DELAY as follows.

$$\frac{C_1 \xrightarrow{\mu} C_2 \quad \text{pn}(\mu) \# \{p, q\}}{p \rightarrow q; C_1 \xrightarrow{\mu} p \rightarrow q; C_2} \text{DELAY}$$

We are left with having to prove that $\llbracket p \rightarrow q; C_2 \rrbracket = N$. According to eq. (3.4), this is true if $\llbracket p \rightarrow q; C_1 \rrbracket \upharpoonright_{p,q} = \llbracket p \rightarrow q; C_2 \rrbracket \upharpoonright_{p,q}$, which follows from the above derivation and lemma 3.

□

By combining lemmas 5 and 6, we obtain the following theorem.

Theorem 2 (Correctness of EPP for simple concurrent choreographies).

- (Completeness) For any C , μ , and C' , $C \xrightarrow{\mu} C'$ implies $\llbracket C \rrbracket \xrightarrow{\mu} \llbracket C' \rrbracket$.
- (Soundness) For any C , μ , and N , $\llbracket C \rrbracket \xrightarrow{\mu} N$ implies $C \xrightarrow{\mu} C'$ for some C' such that $N = \llbracket C' \rrbracket$.

The correctness of EPP gives us a powerful result for free: the EPP of a choreography never gets stuck and can always terminate. Intuitively, this works because interactions in choreographies are written atomically, in the sense that they specify both the send and receive actions needed for the communication in a single term. EPP then projects the send and receive actions correctly into the process terms of sender and receiver, so that they are always well-matched. This prevents mismatched communications, which are possible to write in simple processes as we have seen in exercise 15.

3.3. Correctness of EPP

We now combine proposition 1 with the completeness part of theorem 2, which respectively say that “a choreography can always reduce until it terminates” and “the EPP of a choreography can always do what the choreography does”. This means that “the EPP of a choreography can always reduce until it terminates”, as formalised below.

Theorem 3 (Strong termination for EPP). *For any C , if N is a multi-step derivative of $\llbracket C \rrbracket$, then there exists $\vec{\mu}$ (possibly empty) such that $N \xrightarrow{\vec{\mu}} \mathbf{0}$.*

Proof (sketch). If $C = \mathbf{0}$, then $\llbracket C \rrbracket = \mathbf{0}$. Since $\mathbf{0}$ has no derivatives we necessarily have that $N = \mathbf{0}$ and the thesis follows.

Otherwise ($C \neq \mathbf{0}$), by theorem 2 we know that there exists C' that is a multi-step derivative of C such that $N = \llbracket C' \rrbracket$. By proposition 1, C' can make a sequence of transitions $\vec{\mu}$ such that $C' \xrightarrow{\vec{\mu}} \mathbf{0}$. By the completeness part of theorem 2, N can match these transitions and reach the EPP of $\mathbf{0}$, which is $\mathbf{0}$. \square

Chapter 4

Local computation

Now that we know the basic principles of choreographies and EPP, we can begin to extend our model to capture more interesting—and realistic—examples.

In this chapter, we shall equip our choreographic and process models with the capability of storing values and performing local computation at processes. This will enable us to capture our initial scenario from example 2 more precisely, i.e., we want to define the *content* of the messages exchanged by Buyer and Seller.

4.1 Stores, expressions, and evaluation

Stores Let Var be an infinite set of *variable names*, or variables, ranged over by x, y, z . Also, let Val be an infinite set of *values*, ranged over by v, u . While variables are just names (identifiers), we assume that Val contains all data that we might be interesting in modelling, including integers, strings, and lists.

A *process store* σ maps variables to their respective values. Formally, a process store σ is a function from variables to values:

$$\sigma : Var \longrightarrow Val .$$

We write $PStore$ for the set of all process stores. Also, we shall write $\sigma [x \mapsto v]$ for the update of store σ with the new mapping $x \mapsto v$:

$$\sigma [y \mapsto v] (x) = \begin{cases} v & \text{if } x = y \\ \sigma(x) & \text{otherwise} \end{cases} .$$

A *choreographic store* Σ maps process names to their respective process stores. That is, a choreographic store Σ is a function from process names to process stores:

$$\Sigma : Pid \longrightarrow PStore .$$

$$e ::= v \mid x \mid f(\vec{e})$$

Figure 4.1: Expressions.

One can think of a process store as the memory of a process, and of a choreographic store as the global view over the memory of an entire system of processes.

We shall write $\Sigma[p.x \mapsto v]$ for the update of store Σ such that p 's local variable x is mapped to v :

$$\Sigma[q.x \mapsto v](p) = \begin{cases} \Sigma(p)[x \mapsto v] & \text{if } p = q \\ \Sigma(p) & \text{otherwise} \end{cases}.$$

Expressions Processes compute new values by means of *expressions*, ranged over by e . The syntax of expressions is displayed in fig. 4.1.

There are three forms that an expression can take:

- some constant value v (for any possible v);
- a variable x ;
- a function call $f(\vec{e})$ (also called function invocation), where f is a *function identifier*.

In a function invocation $f(\vec{e})$, \vec{e} is an abbreviation for a sequence of expressions e_1, \dots, e_n , which we call the *arguments* of f . The identifier f is a reference to a known *local function* over values, i.e., a function that a process can execute locally without necessitating to communicate with other processes.

We will abuse expression notation by adopting some syntactic sugar in the remainder, whenever the intuition is obvious. For example, $1 + 2$ (the sum of the natural numbers 1 and 2) is an abbreviation the function invocation $plus(1, 2)$, where $plus$ is the function such that $plus(x, y) = x + y$.

Evaluation Evaluation models the computation of an expression, which returns a value. We write the proposition $\sigma \vdash e \Downarrow v$ for “ e is evaluated as v under the process store σ ”. The idea is that we might need to read the process store in order to compute an expression, since expressions can contain variables whose values is defined in the store.

The inference system for evaluation is displayed in fig. 4.2.

Rule VAL says that a constant value v is always evaluated to itself:

$$\frac{}{\sigma \vdash v \Downarrow v} \text{ VAL}.$$

4.1. Stores, expressions, and evaluation

$$\begin{array}{c}
\frac{}{\sigma \vdash v \downarrow v} \text{ VAL} \qquad \frac{}{\sigma \vdash x \downarrow \sigma(x)} \text{ VAR} \\
\\
\frac{\sigma \vdash e_1 \downarrow v_1 \cdots \sigma \vdash e_n \downarrow v_n \quad \vdash f(\vec{v}) \downarrow v}{\sigma \vdash f(\vec{e}) \downarrow v} \text{ CALL}
\end{array}$$

Figure 4.2: Expression evaluation.

Rule VAR evaluates a variable to the value that it is mapped to in the process store:

$$\frac{}{\sigma \vdash x \downarrow \sigma(x)} \text{ VAR} .$$

Finally, rule CALL evaluates a function call:

$$\frac{\sigma \vdash e_1 \downarrow v_1 \cdots \sigma \vdash e_n \downarrow v_n \quad \vdash f(\vec{v}) \downarrow v}{\sigma \vdash f(\vec{e}) \downarrow v} \text{ CALL} .$$

In the rule, $\sigma \vdash e_1 \downarrow v_1 \cdots \sigma \vdash e_n \downarrow v_n$ means that there is a premise for each argument e_i in \vec{e} . This means that the number of premises of this rule depends on the number of arguments in the function call. These premises evaluate all the expressions that we are passing as arguments to the function. Then, the proposition in the right premise, $\vdash f(\vec{v}) \downarrow v$, reads “invoking function f with arguments \vec{v} evaluates to the value v ”. Value v is thus also the value that we assign to the entire evaluation in the conclusion.

We do not specify how propositions of the kind $\vdash f(\vec{v}) \downarrow v$ can be derived, since this would depend on the specifics of local functions, which we chose to abstract from. We just assume that such a system exists, and that for any f and \vec{v} , it is always possible to find some v such that $\vdash f(\vec{v}) \downarrow v$ in *finite time*.

Remark 2. Our rule CALL evaluates the expressions passed as arguments before evaluating the function invocation. This is sometimes called an *early semantics*, since all arguments are evaluated regardless of whether the function that we are invoking will actually use them. Early evaluation is what many mainstream programming languages use, like Java and C.

Some programming languages, like Haskell, use *lazy evaluation* instead, whereby expressions are passed as they are in function invocations, and are eventually evaluated only if and when the function body actually needs them. This means that the set of values Val includes also the set of all expressions.

Remark 3. In many programming languages, the evaluation of an expression might not terminate, e.g., due to an infinite loop. This is not a problem for our theory: in practice, we could enforce a timeout to all evaluations, and assume that

$$\begin{aligned}
 C &::= I; C \mid 0 \\
 I &::= p.e \rightarrow q.x \mid p.x := e
 \end{aligned}$$

Figure 4.3: Stateful choreographies, syntax.

a special “failure” value is returned if the timeout is reached before the evaluation terminated.

Another practical problem is that evaluating a function invocation might require typing constraints. For example, *plus* might be invocable only by passing numbers as arguments. What if we pass two oranges instead? Again, we can assume that a special failure value is returned in these cases too. *Cruz-Filipe and Montesi [2017]* showed that choreographies can be easily combined with static typing systems that avoid these problems.

4.2 Stateful Choreographies

We enhance choreographies with our new notions for local computation. The key idea is that now processes are equipped with their own local memories, which they can manipulate during execution.

4.2.1 Syntax

The syntax of stateful choreographies is given in fig. 4.3.

In the syntax, I stands for instruction. A choreography is either a term $I; C$, read “run the instruction I and then do C ”, or the terminated choreography 0 . An instruction I can be:

- a communication $p.e \rightarrow q.x$, where process p evaluates expression e locally (according to its local store) and sends the resulting value to process q , which stores the value in its local variable x ;
- a local assignment $p.x := e$, where p evaluates expression e locally and sets its variable x to the resulting value.

We write $\text{pn}(I)$ for the set of process names involved in I , defined as follows.

$$\begin{aligned}
 \text{pn}(p.e \rightarrow q.x) &= \{p, q\} \\
 \text{pn}(p.x := e) &= \{p\}
 \end{aligned}$$

4.2. Stateful Choreographies

$$\begin{array}{c}
\frac{\Sigma(p) \vdash e \downarrow v}{\langle p.x := e; C, \Sigma \rangle \xrightarrow{\tau @ p} \langle C, \Sigma [p.x \mapsto v] \rangle} \text{LOCAL} \\
\\
\frac{\Sigma(p) \vdash e \downarrow v}{\langle p.e \rightarrow q.x; C, \Sigma \rangle \xrightarrow{p.v \rightarrow q} \langle C, \Sigma [q.x \mapsto v] \rangle} \text{COM} \\
\\
\frac{\langle C, \Sigma \rangle \xrightarrow{\mu} \langle C', \Sigma' \rangle \quad \text{pn}(I) \# \text{pn}(\mu)}{\langle I; C, \Sigma \rangle \xrightarrow{\mu} \langle I; C', \Sigma' \rangle} \text{DELAY}
\end{array}$$

Figure 4.4: Stateful choreographies, semantics.

Example 13. We can finally give a precise choreography for our buyer-seller example (example 2), including computation and message contents as well. Recall the informal description of the example:

1. Buyer sends the title of a book she wishes to buy to Seller;
2. Seller replies to Buyer with the price of the book.

A corresponding choreography that defines this behaviour is

Buyer.title \rightarrow Seller.x; Seller.cat(x) \rightarrow Buyer.price; 0

where *title* is a variable, *cat* is a function that given a book title returns the price for it (*cat* stands for catalogue, if you like), and *price* is a variable.

4.2.2 Semantics

Now that processes may read from and write to their local memories (stores), the semantics of choreographies needs to keep track of the stores of all processes. We achieve this by generalising the labelled transition system for choreographies to pairs that consist of a choreography and a choreographic store.

Formally, the semantics of stateful choreographies is defined by the rules given in fig. 4.4. Observe that transitions now have the form $\langle C, \Sigma \rangle \xrightarrow{\mu} \langle C', \Sigma' \rangle$, read “the choreography C equipped with store Σ has a transition with label μ to the choreography C' with store Σ' ”. We call a pair $\langle C, \Sigma \rangle$ a *configuration*.

Rule LOCAL models a local assignment:

$$\frac{\Sigma(p) \vdash e \downarrow v}{\langle p.x := e; C, \Sigma \rangle \xrightarrow{\tau @ p} \langle C, \Sigma [p.x \mapsto v] \rangle} \text{LOCAL} .$$

The premise $\Sigma(p) \vdash e \downarrow v$ evaluates the expression e under the store of the evaluating process p , $\Sigma(p)$. In the derivative, we update the store of process p with the resulting value v . The new label $\tau@p$ indicates that process p has performed an *internal action*, i.e., an action that does not require communications. We symbolise internal actions with τ .

We also update rule COM to consider that messages now contain values:

$$\frac{\Sigma(p) \vdash e \downarrow v}{\langle p.e \rightarrow q.x; C, \Sigma \rangle \xrightarrow{p.v \rightarrow q} \langle C, \Sigma [q.x \mapsto v] \rangle} \text{COM}.$$

The premise evaluates the expression of the sender p under its store. The resulting value v is then sent to the receiver q , and in the derivative we update the value of the variable x used by the receiver in its store. Note that the transition label now contains the sent value v .

Finally, we have to update rule DELAY to consider our new instructions:

$$\frac{\langle C, \Sigma \rangle \xrightarrow{\mu} \langle C', \Sigma' \rangle \quad \text{pn}(I) \# \text{pn}(\mu)}{\langle I; C, \Sigma \rangle \xrightarrow{\mu} \langle I; C', \Sigma' \rangle} \text{DELAY}.$$

In the rule, the continuation C in a choreography $I; C$ is allowed to perform a transition with label μ if μ does not involve any of the processes in I . We update the definition of $\text{pn}(\mu)$ as follows.

$$\begin{aligned} \text{pn}(p.v \rightarrow q) &= \{p, q\} \\ \text{pn}(\tau@p) &= \{p\} \end{aligned}$$

Exercise 25. Let Σ be such that $\Sigma(p)(\text{title}) = \text{"Flowers for Algernon"}$ and cat be a function such that $\text{cat}(\text{"Flowers for Algernon"}) = 100$. Show the transitions of the choreography from example 13 under Σ :

$$\langle \text{Buyer.title} \rightarrow \text{Seller.x}; \text{Seller.cat}(x) \rightarrow \text{Buyer.price}; \mathbf{0}, \Sigma \rangle.$$

4.3 Stateful Processes

Since we updated our choreography model, we also need to update our process model to describe the implementations of choreographies. We apply a similar extension to what we have seen for choreographies, i.e., we pair networks with stores.

4.3. Stateful Processes

$$P ::= p!e; P \mid p?x; P \mid x := e; P \mid 0$$

Figure 4.5: Stateful processes, syntax.

4.3.1 Syntax

The syntax of stateful processes is given in fig. 4.5.

A process term P can be:

- a send action $p!e; P$, read “send the result of evaluating expression e to p , and then proceed as P ”;
- a receive action $p?x; P$, read “receive a value from process p , store it in variable x , and then proceed as P ”;
- an assignment $x := e$, read “store the value obtained by evaluating expression e in variable x ”.

We write $StatefulProc$ for the set of all stateful processes.

Definition 9 (Stateful network). *A stateful network N is a (total) function from process names to process terms:*

$$N : Pid \longrightarrow StatefulProc.$$

4.3.2 Semantics

The semantics of stateful processes is an extension of the semantics of simple processes, which recalls the development of the semantics of stateful choreographies. In particular, we extend networks to *network configurations* $\langle N, \Sigma \rangle$, like we did for stateful choreographies. The rules for transitions are given in fig. 4.6.

4.3.3 EndPoint Projection

We have to update our definition of EPP to our new language model, which now includes primitives for local computation. The general principle of distributing code over process terms remains the same: we just need to update the definition of process projection to handle our new primitives.

First, let us gain some intuition. The network implementation of the choreography given in example 13 should look like the following.

$$\text{Buyer}[\text{Seller!title; Seller?price; 0}] \quad | \quad \text{Seller}[\text{Buyer?x; Buyer!cat}(x); 0]$$

$$\begin{array}{c}
 \frac{\Sigma(\mathbf{p}) \vdash e \downarrow v}{\langle \mathbf{p}[q!e; P] \mid q[p?x; Q], \Sigma \rangle \xrightarrow{p.v \rightarrow q} \langle \mathbf{p}[P] \mid q[Q], \Sigma[q.x \mapsto v] \rangle} \text{COM} \\
 \\
 \frac{\Sigma(\mathbf{p}) \vdash e \downarrow v}{\langle \mathbf{p}[x := e; P], \Sigma \rangle \xrightarrow{\tau @ \mathbf{p}} \langle \mathbf{p}[P], \Sigma[\mathbf{p}.x \mapsto v] \rangle} \text{LOCAL} \\
 \\
 \frac{\langle N, \Sigma \rangle \xrightarrow{\mu} \langle N', \Sigma' \rangle}{\langle N \mid M, \Sigma \rangle \xrightarrow{\mu} \langle N' \mid M, \Sigma' \rangle} \text{PAR}
 \end{array}$$

Figure 4.6: Stateful processes, semantics.

$$\begin{aligned}
 \llbracket \mathbf{p}.e \rightarrow \mathbf{q}.x; C \rrbracket_r &= \begin{cases} q!e; \llbracket C \rrbracket_r & \text{if } r = \mathbf{p} \\ p?x; \llbracket C \rrbracket_r & \text{if } r = \mathbf{q} \\ \llbracket C \rrbracket_r & \text{otherwise} \end{cases} \\
 \llbracket \mathbf{p}.x := e; C \rrbracket_r &= \begin{cases} x := e; \llbracket C \rrbracket_r & \text{if } r = \mathbf{p} \\ \llbracket C \rrbracket_r & \text{otherwise} \end{cases} \\
 \llbracket 0 \rrbracket_p &= 0
 \end{aligned}$$

Figure 4.7: Process projection for stateful choreographies.

We generalise this intuition in our updated definition for process projection, which is displayed in fig. 4.7.

The definition of EPP remains the same (definition 8), modulo the usage of our updated process projection. We report the definition again, for convenience.

Definition 10 (EndPoint Projection (EPP)). *Let C be a choreography. The end-point projection (EPP) of C , written $\llbracket C \rrbracket$, is the network such that, for all $\mathbf{p} \in \text{Pid}$,*

$$\llbracket C \rrbracket(\mathbf{p}) = \llbracket C \rrbracket_p.$$

4.4 Correctness of EPP

To state the correctness result of EPP for stateful choreographies we now need to take stores into consideration. We shall establish that, for any store Σ , the choreographic configuration $\langle C, \Sigma \rangle$ mimicks and is mimicked by the network configuration $\langle \llbracket C \rrbracket, \Sigma \rangle$.

The proof of this result follows the same structure as that of theorem 2. We will need to invoke similar properties to those we used for the simpler setting in

4.4. Correctness of EPP

the previous chapters. Specifically, we have to update the forms of lemma 3 and lemma 4 to deal with stores.

Lemma 7. *If $\langle C, \Sigma \rangle \xrightarrow{\mu} \langle C', \Sigma' \rangle$, then $\llbracket C \rrbracket_r = \llbracket C' \rrbracket_r$ for all $r \notin \text{pn}(\mu)$.*

Lemma 8. *If $\langle N, \Sigma \rangle \xrightarrow{\mu} \langle N', \Sigma' \rangle$ and $p \notin \text{pn}(\mu)$, then $\langle N \setminus p, \Sigma \rangle \xrightarrow{\mu} \langle N' \setminus p, \Sigma' \rangle$.*

Exercise 26. *Prove lemma 7.*

Exercise 27. *Prove lemma 8.*

Theorem 4 (Correctness of EPP for stateful choreographies).

- (Completeness) For any C, Σ, μ, C' , and Σ' , $\langle C, \Sigma \rangle \xrightarrow{\mu} \langle C', \Sigma' \rangle$ implies $\langle \llbracket C \rrbracket, \Sigma \rangle \xrightarrow{\mu} \langle \llbracket C' \rrbracket, \Sigma' \rangle$.
- (Soundness) For any C, Σ, μ, N , and Σ' , $\langle \llbracket C \rrbracket, \Sigma \rangle \xrightarrow{\mu} \langle N, \Sigma' \rangle$ implies $\langle C, \Sigma \rangle \xrightarrow{\mu} C'$ for some C' such that $N = \langle \llbracket C' \rrbracket, \Sigma' \rangle$.

Proof (Completeness part). We proceed by induction on the derivation of the transition $C \xrightarrow{\mu} C'$.

Case The derivation of the transition ends with an application of rule LOCAL:

$$\frac{\Sigma(p) \vdash e \downarrow v}{\langle p.x := e; C', \Sigma \rangle \xrightarrow{\tau @ p} \langle C', \Sigma[p.x \mapsto v] \rangle} \text{LOCAL}.$$

By the definition of EPP,

$$\llbracket C \rrbracket = \llbracket C \rrbracket \setminus p \mid p[x := e; \llbracket C' \rrbracket_p].$$

By lemma 7, $\llbracket C \rrbracket \setminus p = \llbracket C' \rrbracket \setminus p$.

Therefore, we can rewrite $\llbracket C \rrbracket$ from above as:

$$\llbracket C \rrbracket = \llbracket C' \rrbracket \setminus p \mid p[x := e; \llbracket C' \rrbracket_p].$$

By rules PAR and LOCAL for networks, $\langle \llbracket C \rrbracket, \Sigma \rangle$ can mimic the choreographic transition:

$$\langle \llbracket C \rrbracket, \Sigma \rangle \xrightarrow{\tau @ p} \langle \llbracket C' \rrbracket \setminus p \mid p[\llbracket C' \rrbracket_p], \Sigma[q.x \mapsto v] \rangle.$$

Notice that the choreography in the derivative of $\langle \llbracket C \rrbracket, \Sigma \rangle$ above is exactly the EPP of C' , hence $\langle \llbracket C \rrbracket, \Sigma \rangle \xrightarrow{\tau @ p} \langle \llbracket C' \rrbracket, \Sigma[q.x \mapsto v] \rangle$.

Case The derivation of the transition ends with an application of rule COM:

$$\frac{\Sigma(p) \vdash e \downarrow v}{\langle p.e \rightarrow q.x; C', \Sigma \rangle \xrightarrow{p.v \rightarrow q} \langle C', \Sigma[q.x \mapsto v] \rangle} \text{COM}.$$

We proceed as for the previous case (that for rule LOCAL). The only differences are that we need to consider the projections of both p and q (in the previous case, we only needed to consider one process, i.e., the one making the assignment) and that we shall use rule COM to mimic the transition at the level of networks. We report this case in full anyway, for the reader's convenience.

By the definition of EPP,

$$\llbracket C \rrbracket = \llbracket C \rrbracket \setminus p, q \mid p[q!e; \llbracket C' \rrbracket_p] \mid q[p?x; \llbracket C' \rrbracket_q].$$

By lemma 7, $\llbracket C \rrbracket \setminus p, q = \llbracket C' \rrbracket \setminus p, q$.

Therefore, we can rewrite $\llbracket C \rrbracket$ from above as:

$$\llbracket C \rrbracket = \llbracket C' \rrbracket \setminus p, q \mid p[q!e; \llbracket C' \rrbracket_p] \mid q[p?x; \llbracket C' \rrbracket_q].$$

By rules PAR and COM for networks, $\langle \llbracket C \rrbracket, \Sigma \rangle$ can mimic the choreographic transition:

$$\langle \llbracket C \rrbracket, \Sigma \rangle \xrightarrow{p.v \rightarrow q} \langle \llbracket C' \rrbracket \setminus p, q \mid p[\llbracket C' \rrbracket_p] \mid q[\llbracket C' \rrbracket_q], \Sigma[q.x \mapsto v] \rangle.$$

Notice that the choreography in the derivative of $\langle \llbracket C \rrbracket, \Sigma \rangle$ above is exactly the EPP of C' , hence $\langle \llbracket C \rrbracket, \Sigma \rangle \xrightarrow{p.v \rightarrow q} \langle \llbracket C' \rrbracket, \Sigma[q.x \mapsto v] \rangle$.

Case The derivation of the transition ends with an application of rule DELAY:

$$\frac{\langle C_1, \Sigma \rangle \xrightarrow{\mu} \langle C_2, \Sigma' \rangle \quad \text{pn}(I) \# \text{pn}(\mu)}{\langle I; C_1, \Sigma \rangle \xrightarrow{\mu} \langle I; C_2, \Sigma' \rangle} \text{DELAY}.$$

We have two cases, depending on whether I is a local computation ($I = p.x := e$) or a communication ($I = p.e \rightarrow q.x$). We deal with the first case, and leave the second to the reader (the reasoning is similar).

By definition of EPP,

$$\llbracket C \rrbracket = \llbracket C_1 \rrbracket \setminus p \mid p[x := e; \llbracket C_1 \rrbracket_p].$$

4.4. Correctness of EPP

By induction hypothesis,

$$\langle \llbracket C_1 \rrbracket, \Sigma \rangle \xrightarrow{\mu} \langle \llbracket C_2 \rrbracket, \Sigma' \rangle .$$

Since p does not appear in μ (premise of DELAY above), by lemma 8 we obtain

$$\langle \llbracket C_1 \rrbracket \setminus p, \Sigma \rangle \xrightarrow{\mu} \langle \llbracket C_2 \rrbracket \setminus p, \Sigma' \rangle .$$

We use this to make the following derivation:

$$\frac{\langle \llbracket C_1 \rrbracket \setminus p, \Sigma \rangle \xrightarrow{\mu} \langle \llbracket C_2 \rrbracket \setminus p, \Sigma' \rangle}{\begin{array}{c} \langle \llbracket C_1 \rrbracket \setminus p \mid p \left[x := e; \llbracket C_1 \rrbracket_p \right], \Sigma \rangle \xrightarrow{\mu} \\ \langle \llbracket C_2 \rrbracket \setminus p \mid p \left[x := e; \llbracket C_1 \rrbracket_p \right], \Sigma' \rangle \end{array}} \text{PAR} \quad (4.1)$$

From the premises of rule DELAY at the beginning and lemma 7, we have that $\llbracket C_1 \rrbracket_p = \llbracket C_2 \rrbracket_p$. Therefore the derivative we obtained with rule PAR is equal to $\llbracket p.x := e; C_2 \rrbracket$, and the thesis is proven.

□

Exercise 28 (!). Prove the soundness part of theorem 4.

Hint: Follow the structure of the proof of lemma 6, paying attention to the difference between the definitions of syntax, semantics, and EPP between chapters 2 and 3 and this chapter.

Chapter 5

Conditionals and knowledge of choice

The choreographies that we have seen so far are simple sequences of interactions. What if we wanted to express a choice between alternative behaviours? For instance, in our example with Buyer and Seller, Buyer may proceed by deciding whether to buy the book or not depending on the price given by the Seller. In this chapter, we extend our framework with conditionals that allow to capture this kind of situations. We are going to see that this has nontrivial consequences on EPP: some choreographies will turn out to be unprojectable, i.e., our definition of EPP is not going to be able to translate them to a network implementation.

5.1 Modelling Conditionals

We start by extending our models of choreographies and processes with syntax and semantics for conditional execution.

5.1.1 Choreographies

Syntax We extend statements in choreographies to be instructions, denoted I , which may also contain conditionals (if-then-else constructs). The new syntax is given in fig. 5.1.

There is one new kind of instructions, the conditional $\text{if } p.f \text{ then } C_1 \text{ else } C_2$, read “process p evaluates expression e , and the choreography proceeds as C_1 if the result is the value `true`, or proceeds as C_2 otherwise”. Thus, we now assume that the set of possible values contains booleans. The condition e used in a conditional is called a *guard*.

$$\begin{aligned}
 C &::= I; C \mid \mathbf{0} \\
 I &::= p.e \rightarrow q.x \mid p.x := e \mid \text{if } p.e \text{ then } C_1 \text{ else } C_2
 \end{aligned}$$

Figure 5.1: Choreographies with conditionals, syntax.

$$\begin{aligned}
 &\frac{\Sigma(p) \vdash e \downarrow v}{\langle p.x := e; C, \Sigma \rangle \xrightarrow{\tau @ p} \langle C, \Sigma [p.x \mapsto v] \rangle} \text{LOCAL} \\
 &\frac{\Sigma(p) \vdash e \downarrow v}{\langle p.e \rightarrow q.x; C, \Sigma \rangle \xrightarrow{p.v \rightarrow q} \langle C, \Sigma [q.x \mapsto v] \rangle} \text{COM} \\
 &\frac{i = 1 \text{ if } \Sigma(p) \vdash e \downarrow \text{true}, i = 2 \text{ otherwise}}{\langle \text{if } p.e \text{ then } C_1 \text{ else } C_2; C, \Sigma \rangle \xrightarrow{\tau @ p} \langle C_i \circ C, \Sigma \rangle} \text{COND} \\
 &\frac{\langle C, \Sigma \rangle \xrightarrow{\mu} \langle C', \Sigma' \rangle \quad \text{pn}(I) \# \text{pn}(\mu)}{\langle I; C, \Sigma \rangle \xrightarrow{\mu} \langle I; C', \Sigma' \rangle} \text{DELAY} \\
 &\frac{\langle C_1, \Sigma \rangle \xrightarrow{\mu} \langle C'_1, \Sigma' \rangle \quad \langle C_2, \Sigma \rangle \xrightarrow{\mu} \langle C'_2, \Sigma' \rangle \quad p \notin \text{pn}(\mu)}{\langle \text{if } p.e \text{ then } C_1 \text{ else } C_2; C, \Sigma \rangle \xrightarrow{\mu} \langle \text{if } p.e \text{ then } C'_1 \text{ else } C'_2; C, \Sigma' \rangle} \text{DELAYCOND}
 \end{aligned}$$

Figure 5.2: Choreographies with conditionals, semantics.

Extending function pn to the new syntax is easy:

$$\begin{aligned}
 \text{pn}(I; C) &= \text{pn}(I) \cup \text{pn}(C) \\
 \text{pn}(\mathbf{0}) &= \emptyset \\
 \text{pn}(p.e \rightarrow q.x) &= \{p, q\} \\
 \text{pn}(p.x := e) &= \{p\} \\
 \text{pn}(\text{if } p.e \text{ then } C_1 \text{ else } C_2) &= \{p\} \cup \text{pn}(C_1) \cup \text{pn}(C_2) .
 \end{aligned}$$

Semantics The semantics of choreographies with conditionals is defined by the rules displayed in fig. 5.2.

There are two new rules.

Rule COND formalises the intended meaning of conditionals, choosing the

5.1. Modelling Conditionals

$$0 \mathbin{\text{;}} C = C \quad (I; C) \mathbin{\text{;}} C' = I; (C \mathbin{\text{;}} C')$$

Figure 5.3: Choreographies with conditionals, $\mathbin{\text{;}}$ operator (“fat semicolon”).

right branch depending on the result of the guard:

$$\frac{i = 1 \text{ if } \Sigma(p) \vdash e \downarrow \text{true}, i = 2 \text{ otherwise}}{\langle \text{if } p.e \text{ then } C_1 \text{ else } C_2; C, \Sigma \rangle \xrightarrow{\tau @ p} \langle C_i \mathbin{\text{;}} C, \Sigma \rangle} \text{COND}.$$

Observe that the conclusion of the rule uses a new operator $\mathbin{\text{;}}$ (the “fat semicolon”). Before giving the formal definition of $\mathbin{\text{;}}$, let us see an example about why it is necessary. Consider the choreography

$$\text{if } p.\text{true} \text{ then } 0 \text{ else } 0; C.$$

If we used the simple semicolon $;$ instead of the fat semicolon $\mathbin{\text{;}}$ in the conclusion of rule COND, this choreography would perform a transition to $0; C$, which is not a valid term in our syntax. One option is to complicate our syntax and semantics to deal with generic sequential composition, whereby general choreographies C and C' can be composed as $C; C'$. Another option, which is the one we follow here, is to avoid this problem entirely by using the $\mathbin{\text{;}}$ operator. In this example, $0 \mathbin{\text{;}} C = C$, which respects our syntax. In general, $C \mathbin{\text{;}} C'$ replaces the “exit points” of C (occurrences of 0 that represent termination in C) with C' , so it behaves as the sequential composition of C and C' . Formally, $\mathbin{\text{;}}$ is defined by the rules given in fig. 5.3.

The other new rule in our semantics, rule DELAYCOND, is necessary to model concurrent execution of independent processes inside of conditionals. Consider the following choreography, where we use round parentheses to elicit its structure.

$$(\text{if } p.e \text{ then } q.x := e'; 0 \text{ else } q.x := e'; 0); 0$$

Observe that the local assignment by q is going to be the same regardless of which branch p chooses. Thus, since p and q represent independent processes, it should be possible to perform the local assignment by q before p evaluates its conditional and chooses which branch should be executed. Rule DELAYCOND captures this kind of situations by allowing transitions from inside of conditional branches to be performed, whenever these do not depend on evaluating the guard (meaning that the same transition can be performed by both branches):

$$\frac{\langle C_1, \Sigma \rangle \xrightarrow{\mu} \langle C'_1, \Sigma' \rangle \quad \langle C_2, \Sigma \rangle \xrightarrow{\mu} \langle C'_2, \Sigma' \rangle \quad p \notin \text{pn}(\mu)}{\langle \text{if } p.e \text{ then } C_1 \text{ else } C_2; C, \Sigma \rangle \xrightarrow{\mu} \langle \text{if } p.e \text{ then } C'_1 \text{ else } C'_2; C, \Sigma' \rangle} \text{DELAYCOND}.$$

$$\begin{aligned}
 P &::= I; P \mid \mathbf{0} \\
 I &::= p!e \mid p?x \mid x := e \mid \text{if } e \text{ then } P \text{ else } Q
 \end{aligned}$$

Figure 5.4: Processes with conditionals, syntax.

$$\begin{aligned}
 &\frac{\Sigma(p) \vdash e \downarrow v}{\langle p[q!e; P] \mid q[p?x; Q], \Sigma \rangle \xrightarrow{p.v \rightarrow q} \langle p[P] \mid q[Q], \Sigma[q.x \mapsto v] \rangle} \text{COM} \\
 &\frac{\Sigma(p) \vdash e \downarrow v}{\langle p[x := e; P], \Sigma \rangle \xrightarrow{\tau @ p} \langle p[P], \Sigma[p.x \mapsto v] \rangle} \text{LOCAL} \\
 &\frac{i = 1 \text{ if } \Sigma(p) \vdash e \downarrow \text{true}, i = 2 \text{ otherwise}}{\langle p[\text{if } e \text{ then } P_1 \text{ else } P_2; Q], \Sigma \rangle \xrightarrow{\tau @ p} \langle p[P_i \circledast Q], \Sigma \rangle} \text{COND} \\
 &\frac{\langle N, \Sigma \rangle \xrightarrow{\mu} \langle N', \Sigma' \rangle}{\langle N \mid M, \Sigma \rangle \xrightarrow{\mu} \langle N' \mid M, \Sigma' \rangle} \text{PAR}
 \end{aligned}$$

Figure 5.5: Processes with conditionals, semantics.

5.1.2 Processes

Introducing conditionals to our process calculus follows the same principles as for choreographies. We will need a \circledast operator also for our process terms, to mimic what happens at the choreography level, so it is convenient to reformulate the syntax of processes by distinguishing instructions. The new syntax and semantics are given by the rules in figs. 5.4 to 5.6.

5.1.3 EndPoint Projection

Adding conditionals to choreographies has intriguing consequences for EPP. Consider the following choreography.

$$C_{\text{unproj}} \triangleq (\text{if } p.e \text{ then } p.\text{true} \rightarrow q.x; \mathbf{0} \text{ else } \mathbf{0}); \mathbf{0}$$

$$\mathbf{0} \circledast P = P \quad (I; P) \circledast Q = I; (P \circledast Q)$$

Figure 5.6: Processes with conditionals, operator \circledast .

5.2. Knowledge of choice

If p chooses the left branch in the conditional, then it sends the value `true` to q . Otherwise, the choreography terminates. What should the EPP of C_{unproj} look like? It seems obvious that the resulting network should consist of two processes, p and q :

$$\llbracket C_{\text{unproj}} \rrbracket = p \left[\llbracket C_{\text{unproj}} \rrbracket_p \right] \mid p \left[\llbracket C_{\text{unproj}} \rrbracket_q \right]. \quad (5.1)$$

The projection for p seems easy to achieve:

$$\llbracket C_{\text{unproj}} \rrbracket_p = (\text{if } e \text{ then } q!\text{true}; 0 \text{ else } 0); 0.$$

Instead, we run into trouble for projecting q . Process q is not involved in evaluating the conditional (since that is local at p), so its projection should just “skip” it. Indeed the only piece of code in the choreography that involves q is $p.\text{true} \rightarrow q.x$. If we choose to project that (and we should, since the choreography contains it), we obtain:

$$\llbracket C_{\text{unproj}} \rrbracket_q = p?x; 0. \quad (5.2)$$

If, instead, we choose *not to* project the receive action by q , we obtain:

$$\llbracket C_{\text{unproj}} \rrbracket_q = 0. \quad (5.3)$$

Neither of these decisions gives us a correct EPP, as we can check with two exercises.

Exercise 29. Assume Σ such that $\Sigma(p) \vdash e \downarrow \text{false}$. Show that, if we adopt the behaviour projection in eq. (5.2), then $\langle \llbracket C_{\text{unproj}} \rrbracket, \Sigma \rangle$ has a derivative that does not correspond to any derivative of $\langle C_{\text{unproj}}, \Sigma \rangle$.

Exercise 30. Assume Σ such that $\Sigma(p) \vdash e \downarrow \text{true}$. Show that, if we adopt the behaviour projection in eq. (5.3), then $\langle \llbracket C_{\text{unproj}} \rrbracket, \Sigma \rangle$ has a derivative that does not correspond to any derivative of $\langle C_{\text{unproj}}, \Sigma \rangle$.

These exercises clarify that a naïve definition of EPP is not going to cut it. We explore this issue in depth in the coming section.

5.2 Knowledge of choice

Recall our problematic example for projecting conditionals:

$$C_{\text{unproj}} \triangleq (\text{if } p.e \text{ then } p.\text{true} \rightarrow q.x; 0 \text{ else } 0); 0.$$

Observe that the choreography states that p and q have to behave differently depending on the branch chosen by the conditional. For p , this is not a problem,

because p is the process making the choice and thus “knows” whether we should run the then branch or the else branch. However, q does not have any information that it can use to determine which branch has been chosen. Should then q wait for a message from p (then branch) or simply terminate (else branch)? The problem is thus that the choreography does not specify a sufficient *flow of information* about choices among processes. In the literature, a choreography such as this is said to be *unrealisable*, or *unprojectable*, in the sense that if we project it naïvely as we attempted to do in part 3 we obtain an incorrect (network) implementation. Mentions of this kind of properties (sometimes with different terminology) were already present in the early days of formal methods for choreographies [Fu et al. 2005a, Carbone et al. 2007, Qiu et al. 2007, Lanese et al. 2008]. In more expressive choreography models, it is not only conditionals that can make a choreography unprojectable, as we will see later on. For now, let us focus on this particular construct.

There are different and useful theoretical tools that can be adopted to deal with unrealisability.

Detection First of all, we can develop mechanical methods to detect whether a choreography is realisable. Then, we can add realisability as a necessary assumption for the definition and/or the correctness of EPP, i.e., we guarantee that EPP is defined and/or correct only if the choreography given as input is realisable.

Amendment Given an unrealisable choreography due to some insufficient communication flow, we can attempt at automatically fixing the flow by adding extra communications to the choreography.

Smart Projection We can design EPP such that it can also project unrealisable choreographies, by adding extra communications in the generated network that are not defined in the original choreography.

We focus on detection first, and leave amendment and smart projections to later.

5.2.1 EPP with detection

We define EPP for choreographies with conditionals, including a requirement that detects unrealisable choreographies. The definition of EPP is the same as the last one, save that we need to add the case for conditionals to process projection, in fig. 5.7.

The rule for projecting a choreographic conditional includes a check that prevents projecting problematic choreographies. Namely, when we are projecting a

5.2. Knowledge of choice

$$\begin{aligned}
\llbracket p.e \rightarrow q.x; C \rrbracket_r &= \begin{cases} q!e; \llbracket C \rrbracket_r & \text{if } r = p \\ p?x; \llbracket C \rrbracket_r & \text{if } r = q \\ \llbracket C \rrbracket_r & \text{otherwise} \end{cases} \\
\llbracket p.x := e; C \rrbracket_r &= \begin{cases} x := e; \llbracket C \rrbracket_r & \text{if } r = p \\ \llbracket C \rrbracket_r & \text{otherwise} \end{cases} \\
\llbracket \text{if } p.e \text{ then } C_1 \text{ else } C_2; C \rrbracket_r &= \begin{cases} (\text{if } e \text{ then } \llbracket C_1 \rrbracket_r \text{ else } \llbracket C_2 \rrbracket_r); \llbracket C \rrbracket_r & \text{if } r = p \\ \llbracket C_1 \rrbracket_r \circ \llbracket C \rrbracket_r & \text{if } r \neq p \text{ and } \llbracket C_1 \rrbracket_r = \llbracket C_2 \rrbracket_r \end{cases} \\
\llbracket 0 \rrbracket_p &= 0
\end{aligned}$$

Figure 5.7: Process projection for choreographies with conditionals.

conditional $\text{if } p.e \text{ then } C_1 \text{ else } C_2$, we just proceed homomorphically if we are projecting the process that evaluates the guard (p)—by homomorphically, we mean that we follow the structure of the choreography and project a corresponding conditional with the same structure. If, instead, we are projecting some other process, we know that this process will not know the choice that p will make between the two branches C_1 and C_2 . Thus we require that the behaviour of this uninformed process is *the same* ($\llbracket C_1 \rrbracket_r = \llbracket C_2 \rrbracket_r$ in the rule).

Definition 11 (EndPoint Projection (EPP)). *Let C be a choreography. The end-point projection (EPP) of C , written $\llbracket C \rrbracket$, is the network such that, for all $p \in \text{Pid}$,*

$$\llbracket C \rrbracket(p) = \llbracket C \rrbracket_p .$$

So far, all definitions of EPP that we have seen were complete—as in a complete function, in the sense that EPP was defined for all possible choreographies. The check performed by the rule for projecting conditionals makes EPP a partial function instead, since we now have choreographies that may not respect our check. The unrealisable choreography from section 5.2 is an example of a choreography for which EPP is undefined. We recall its definition here:

$$C_{\text{unproj}} \triangleq (\text{if } p.e \text{ then } p.\text{true} \rightarrow q.x; 0 \text{ else } 0); 0 . \quad (5.4)$$

Then we can prove that it cannot be projected.

Proposition 5. *Let C_{unproj} be the choreography in eq. (5.4). Then, $\llbracket C_{\text{unproj}} \rrbracket$ is undefined.*

Proof. For $\llbracket C_{\text{unproj}} \rrbracket$ to be defined, we need $\llbracket C_{\text{unproj}} \rrbracket_q$ to be defined. Since C_{unproj} is a conditional, the only rule that we can apply for $\llbracket C_{\text{unproj}} \rrbracket_q$ is the third one in fig. 5.7. We proceed by cases according to the definition of the rule. Since $p \neq q$, we cannot apply the first case. The only remaining option is the second one, which requires the following.

- $p \neq q$. This holds.
- $\llbracket p.\text{true} \rightarrow q.x; 0 \rrbracket_q = \llbracket 0 \rrbracket_q$. This does not hold, because

$$\llbracket p.\text{true} \rightarrow q.x; 0 \rrbracket_q = p?x; 0,$$

which is not equal to $\llbracket 0 \rrbracket_q = 0$.

Thus, $\llbracket C_{\text{unproj}} \rrbracket_q$ is undefined and, consequently, $\llbracket C_{\text{unproj}} \rrbracket$. \square

When the EPP of a choreography is not defined, we say that the choreography is unprojectable.

Observe that the requirement on conditionals may seem strict, but we can still write quite meaningful choreographies. Here is a modified version of C_{unproj} that is projectable.

$$(\text{if } p.e \text{ then } p.\text{true} \rightarrow q.x; 0 \text{ else } p.\text{false} \rightarrow q.x; 0); 0 \quad (5.5)$$

Exercise 31. Write the EPP of the choreography in eq. (5.5).

Intuitively, the choreography in eq. (5.5) is projectable because q has the same behaviour in both branches of the conditional, i.e., a receive action on variable x . Observe that p , however, can send different values to q (true and false respectively, in our example), since p knows which branch is chosen. Now that q has this information, it can evaluate it internally with a conditional to know which branch we are in. Then, we could have that q does something different depending on this. For example, we may wish that q sends some money to p when p sends true , and no money when p sends false . We do this in the following choreography. As usual, we assume that $q.x$ returns the value stored in x at q . The symbol 0 in the choreography below is just the natural number 0 , representing no money.

$$\left(\begin{array}{l} \text{if } p.e \text{ then } p.\text{true} \rightarrow q.x; \\ \quad \left(\begin{array}{l} \text{if } q.x \text{ then } q.\text{money} \rightarrow p.m; 0 \\ \quad \text{else } q.0 \rightarrow p.m; 0 \end{array} \right); 0 \\ \text{else } p.\text{false} \rightarrow q.x; \\ \quad \left(\begin{array}{l} \text{if } q.x \text{ then } q.\text{money} \rightarrow p.m; 0 \\ \quad \text{else } q.0 \rightarrow p.m; 0 \end{array} \right); 0 \end{array} \right); 0 \quad (5.6)$$

5.3. Selections

Exercise 32. Write the process projection of q for the choreography in eq. (5.6).

Lemma 9. If $\langle C, \Sigma \rangle \xrightarrow{\mu} \langle C', \Sigma' \rangle$, then $\llbracket C \rrbracket_r = \llbracket C' \rrbracket_r$ for all $r \notin \text{pn}(\mu)$.

Lemma 10. If $\langle N, \Sigma \rangle \xrightarrow{\mu} \langle N', \Sigma' \rangle$ and $p \notin \text{pn}(\mu)$, then $\langle N \setminus p, \Sigma \rangle \xrightarrow{\mu} \langle N' \setminus p, \Sigma' \rangle$.

Exercise 33. Prove lemma 9.

Exercise 34. Prove lemma 10.

A few remarks There are two problems that become pretty evident when looking at choreographies such as that in eq. (5.6). First, the choreography is repetitive and tedious to write, because we have to copy-paste exactly the same code for q in both branches, to respect our condition for projecting conditionals. So it is much “bigger” than we would wish for. Second, we now have a problem with p inside of the conditional evaluated by q . Namely, since p does not know which branch q chooses, we had to insert a communication of 0 from q to p to represent the act of giving no money. This seems silly: from the choreography, we know that if p sends `false`, then p is not due any money. So sending a 0 from q to p is a waste of a communication: it would be better if we could write a choreography where q simply does not send anything when no money is due.

Summing up our considerations, we would like to be able to write (and project!) a choreography that looks like the following, which is a direct formalisation of what we would like to happen: if p wants some money, then q sends it to p , otherwise nothing happens.

$$(\text{if } p.e \text{ then } q.\text{money} \rightarrow p.m; 0 \text{ else } 0); 0 \quad (5.7)$$

Of course, the choreography in eq. (5.7) is unprojectable according to our rules so far. Our aim in the next section is to develop a choreography model that allows us to write things that are nearly as concise as this choreography—meaning that we avoid the two problems mentioned above—and are also projectable!

5.3 Selections

We add a new primitive to our choreography model, which can be used to explicitly (and efficiently) propagate information among processes about which branches have been chosen in conditionals.

$$\begin{aligned}
 C &::= I; C \mid 0 \\
 I &::= p.e \rightarrow q.x \mid p \rightarrow q.L \mid p.x := e \mid \text{if } p.e \text{ then } C_1 \text{ else } C_2
 \end{aligned}$$

Figure 5.8: Choreographies with selections, syntax.

5.3.1 Choreographies

Syntax The updated syntax of choreographies is given in fig. 5.8. The new primitive is $p \rightarrow q.L$, read “process p sends to process q the selection of label L ”. Labels, ranged over by L , are picked from an infinite set of label names. Intuitively, when we write a selection $p \rightarrow q.L$, p is *selecting* one of the behaviours (L in this case) that q offers. In programming languages, we can think of labels as abstractions of method names in object-oriented programming, or operations in service-oriented computing.

Extending function pn to selections is straightforward:

$$\begin{aligned}
 \text{pn}(I; C) &= \text{pn}(I) \cup \text{pn}(C) \\
 \text{pn}(0) &= \emptyset \\
 \text{pn}(p.e \rightarrow q.x) &= \{p, q\} \\
 \text{pn}(p \rightarrow q.L) &= \{p, q\} \\
 \text{pn}(p.x := e) &= \{p\} \\
 \text{pn}(\text{if } p.e \text{ then } C_1 \text{ else } C_2) &= \{p\} \cup \text{pn}(C_1) \cup \text{pn}(C_2) .
 \end{aligned}$$

Example Before we dive into the formal details of selections, let us see how they can help us with our example in eq. (5.7), where p needs to inform q of whether we chose the left or the right branch of the conditional, such that q can behave accordingly. We can choose two labels, say PAY and NO , and imagine that q should offer p a choice between the two behaviours in the respective branches of the conditional. When q receives a selection for PAY , then q knows that it should behave as specified in the left branch of the conditional and pay some money— $q.\text{money} \rightarrow p.m; 0$. Instead, when q receives a selection for NO , then it knows that it should behave as specified in the right branch and hence do nothing— 0 . We can formalise this intuition as the following choreography.

$$(\text{if } p.e \text{ then } p \rightarrow q.\text{PAY}; q.\text{money} \rightarrow p.m; 0 \text{ else } p \rightarrow q.\text{NO}; 0) ; 0 \quad (5.8)$$

In the choreography in eq. (5.8), p makes a choice as before (evaluating the conditional). Differently from before, however, right after making this choice p

5.3. Selections

$$\begin{array}{c}
\frac{\Sigma(p) \vdash e \downarrow v}{\langle p.x := e; C, \Sigma \rangle \xrightarrow{\tau @ p} \langle C, \Sigma [p.x \mapsto v] \rangle} \text{LOCAL} \\
\\
\frac{\Sigma(p) \vdash e \downarrow v}{\langle p.e \rightarrow q.x; C, \Sigma \rangle \xrightarrow{p.v \rightarrow q} \langle C, \Sigma [q.x \mapsto v] \rangle} \text{COM} \\
\\
\frac{}{\langle p \rightarrow q.L; C, \Sigma \rangle \xrightarrow{p \rightarrow q.L} \langle C, \Sigma \rangle} \text{SEL} \\
\\
\frac{i = 1 \text{ if } \Sigma(p) \vdash e \downarrow \text{true}, i = 2 \text{ otherwise}}{\langle \text{if } p.e \text{ then } C_1 \text{ else } C_2; C, \Sigma \rangle \xrightarrow{\tau @ p} \langle C_i \circ C, \Sigma \rangle} \text{COND} \\
\\
\frac{\langle C, \Sigma \rangle \xrightarrow{\mu} \langle C', \Sigma' \rangle \quad \text{pn}(I) \# \text{pn}(\mu)}{\langle I; C, \Sigma \rangle \xrightarrow{\mu} \langle I; C', \Sigma' \rangle} \text{DELAY} \\
\\
\frac{\langle C_1, \Sigma \rangle \xrightarrow{\mu} \langle C'_1, \Sigma' \rangle \quad \langle C_2, \Sigma \rangle \xrightarrow{\mu} \langle C'_2, \Sigma' \rangle \quad p \notin \text{pn}(\mu)}{\langle \text{if } p.e \text{ then } C_1 \text{ else } C_2; C, \Sigma \rangle \xrightarrow{\mu} \langle \text{if } p.e \text{ then } C'_1 \text{ else } C'_2; C, \Sigma' \rangle} \text{DELAYCOND}
\end{array}$$

Figure 5.9: Choreographies with selections, semantics.

now communicates a label to q . In the left branch, q receives label `PAY`. In the right branch, q receives label `NO`. We then assume that, thanks to the fact that q receives a *different* label for the two branches, it is able to use this information to know how to behave in the two branches.

Semantics The semantics of selections is straightforward, since they do not change the memory of any process. We just need to add the following rule.

$$\frac{}{\langle p \rightarrow q.L; C, \Sigma \rangle \xrightarrow{p \rightarrow q.L} \langle C, \Sigma \rangle} \text{SEL}$$

The complete set of rules that we obtain for choreographies with selections is displayed in fig. 5.9.

5.3.2 Processes

Regarding the process model, we need to add two primitives: one for sending selections, and one for receiving them. The updated syntax and semantics are given by the rules in figs. 5.10 and 5.11.

$$\begin{aligned}
 P &::= I; P \mid \mathbf{0} \\
 I &::= \mathbf{p}!e \mid \mathbf{p}?x \mid \mathbf{p} \oplus L \mid \mathbf{p} \& \{L_i : P_i\}_{i \in I} \mid x := e \mid \text{if } e \text{ then } P \text{ else } Q
 \end{aligned}$$

Figure 5.10: Processes with selections, syntax.

$$\begin{aligned}
 &\frac{\Sigma(\mathbf{p}) \vdash e \downarrow v}{\langle \mathbf{p}[\mathbf{q}!e; P] \mid \mathbf{q}[\mathbf{p}?x; Q], \Sigma \rangle \xrightarrow{\mathbf{p}.v \rightarrow \mathbf{q}} \langle \mathbf{p}[P] \mid \mathbf{q}[Q], \Sigma[\mathbf{q}.x \mapsto v] \rangle} \text{COM} \\
 &\frac{j \in I}{\langle \mathbf{p}[\mathbf{q} \oplus L_j; P] \mid \mathbf{q}[\mathbf{p} \& \{L_i : P_i\}_{i \in I}; Q], \Sigma \rangle \xrightarrow{\mathbf{p} \rightarrow \mathbf{q}.L_j} \langle \mathbf{p}[P] \mid \mathbf{q}[P_j; Q], \Sigma \rangle} \text{SEL} \\
 &\frac{\Sigma(\mathbf{p}) \vdash e \downarrow v}{\langle \mathbf{p}[x := e; P], \Sigma \rangle \xrightarrow{\tau @ \mathbf{p}} \langle \mathbf{p}[P], \Sigma[\mathbf{p}.x \mapsto v] \rangle} \text{LOCAL} \\
 &\frac{i = 1 \text{ if } \Sigma(\mathbf{p}) \vdash e \downarrow \text{true}, i = 2 \text{ otherwise}}{\langle \mathbf{p}[\text{if } e \text{ then } P_1 \text{ else } P_2; Q], \Sigma \rangle \xrightarrow{\tau @ \mathbf{p}} \langle \mathbf{p}[P_i; Q], \Sigma \rangle} \text{COND} \\
 &\frac{\langle N, \Sigma \rangle \xrightarrow{\mu} \langle N', \Sigma' \rangle}{\langle N \mid M, \Sigma \rangle \xrightarrow{\mu} \langle N' \mid M, \Sigma' \rangle} \text{PAR}
 \end{aligned}$$

Figure 5.11: Processes with selections, semantics.

The new primitives are $\mathbf{p} \oplus L$ and $\mathbf{p} \& \{L_i : P_i\}_{i \in I}$. A term $\mathbf{p} \oplus L$ sends the choice of a label L to \mathbf{p} . Dually, a term $\mathbf{p} \& \{L_i : P_i\}_{i \in I}$ (also called *branching term*, or simply a branching) offers the possibility to choose from many behaviours $\{P_i\}_{i \in I}$ for some finite set I . When a branching term receives a label, it runs the behaviour that the label is associated to. For example, we read $\mathbf{p} \& \{L_1 : P_1, L_2 : P_2\}$ as “receive a label, and then run P_1 if the received label is L_1 , or run P_2 if the received label is L_2 ”. This intuition is formalised by rule SEL in fig. 5.11, where the sender selects a label among those offered by the receiver ($j \in I$).

5.3. Selections

5.3.3 EPP for choreographies with selections

Consider again the choreography in eq. (5.8), which is again given below, for convenience.

$$(\text{ if } p.e \text{ then } p \rightarrow q.\text{PAY}; q.money \rightarrow p.m; \mathbf{0} \text{ else } p \rightarrow q.\text{NO}; \mathbf{0}) ; \mathbf{0}$$

We can now manually write a network that implements this choreography as follows.

$$\begin{array}{c} p[\text{if } e \text{ then } q \oplus \text{PAY}; q?m; \mathbf{0} \text{ else } q \oplus \text{NO}; \mathbf{0}] \\ | \\ q[p \&\{\text{PAY} : p!money; \mathbf{0}, \text{NO} : \mathbf{0}\}; \mathbf{0}] \end{array} \quad (5.9)$$

Tuning EPP to mechanically produce this kind of results requires adding a rule for projecting selections and updating the rule for projecting conditionals as follows.

$$\begin{aligned} \llbracket p \rightarrow q.L; C \rrbracket_r &= \begin{cases} q \oplus L; \llbracket C \rrbracket_r & \text{if } r = p \\ p \&\{L : \llbracket C \rrbracket_r\}; \mathbf{0} & \text{if } r = q \\ \llbracket C \rrbracket_r & \text{otherwise} \end{cases} \\ \llbracket \text{if } p.e \text{ then } C_1 \text{ else } C_2; C \rrbracket_r &= \begin{cases} (\text{if } e \text{ then } \llbracket C_1 \rrbracket_r \text{ else } \llbracket C_2 \rrbracket_r); \llbracket C \rrbracket_r & \text{if } r = p \\ (\llbracket C_1 \rrbracket_r \sqcup \llbracket C_2 \rrbracket_r) \ddagger \llbracket C \rrbracket_r & \text{otherwise} \end{cases} \end{aligned}$$

The projection of a selection $p \rightarrow q.L$ is simple: the sender is projected to the sending of the label— $q \oplus l$ —whereas the receiver is projected to a branching with a single branch with label l — $p \&\{l : \llbracket C \rrbracket_r\}$. Observe that we do not require equality of projections for the processes that do not evaluate the conditional. Instead, we have a new ingredient, the *merging operator* \sqcup . This is a partial operator on behaviours—meaning that it is not always defined—so EPP is still partial. However, it is now much more expressive, since we can define \sqcup to take advantage of selections. Formally, $P \sqcup Q$ is defined inductively on the structure of P and Q . The rules defining \sqcup are displayed in fig. 5.12.

Definition 12 (Merging). *The merging operator \sqcup is the smallest binary operator between process terms satisfying the rules in fig. 5.12.*

Merging proceeds homomorphically, requiring the two behaviours to be merged to have the same structure. For all terms but branchings, we require the identity (the two merged terms are the same). For branchings (last rule), we apply the following reasoning: all branches with the same label ($k \in I \cap J$) are merged, whereas branches with different labels are simply added to the result with no requirements ($i \in I \setminus J$ and $j \in J \setminus I$).

Example 14. *The network in eq. (5.9) is the EPP of the choreography in eq. (5.8).*

$$\begin{aligned}
 I; P_1 \sqcup I; P_2 &= I; (P_1 \sqcup P_2) \quad (I \text{ not a conditional or a branching}) \\
 \mathbf{0} \sqcup \mathbf{0} &= \mathbf{0} \\
 \text{if } e \text{ then } P_1 \text{ else } P'_1; P''_1 &= \text{if } e \text{ then } (P_1 \sqcup P_2) \text{ else } (P'_1 \sqcup P'_2); (P''_1 \sqcup P''_2) \\
 \text{if } e \text{ then } P_2 \text{ else } P'_2; P''_2 &= \\
 p \&\{l_i : P_i\}_{i \in I}; R \sqcup p \&\{l_j : Q_j\}_{j \in J}; S &= \\
 p \& \left(\{l_k : (P_k \sqcup Q_k)\}_{k \in I \cap J} \cup \{l_i : P_i\}_{i \in I \setminus J} \cup \{l_j : Q_j\}_{j \in J \setminus I} \right); (R \sqcup S)
 \end{aligned}$$

Figure 5.12: Merging operator for processes with selections.

Exercise 35. Write the EPP of the following choreography.

$$\left(\begin{array}{l} \text{if } p.e \text{ then } p \rightarrow q.\text{PAY}; q.\text{money} \rightarrow p.m; \mathbf{0} \\ \text{else} \\ \left(\begin{array}{l} \text{if } p.e' \text{ then } p \rightarrow q.\text{REIMBURSE}; p.\text{money} \rightarrow q.m; \mathbf{0} \\ \text{else } p \rightarrow q.\text{NO}; \mathbf{0} \end{array} \right); \mathbf{0} \end{array} \right); \mathbf{0}$$

Proposition 6. For all P , $P \sqcup P = P$.

Exercise 36. Prove proposition 6.

5.3.4 Operational correspondence for EPP with selections

Selections make stating the operational correspondence theorem for EPP trickier. Consider the following choreography.

$$C_{\text{cond}} \triangleq (\text{if } p.e \text{ then } p \rightarrow q.\text{LEFT}; \mathbf{0} \text{ else } p \rightarrow q.\text{RIGHT}; \mathbf{0}); \mathbf{0}$$

Its projection is the following.

$$\begin{aligned}
 \llbracket C_{\text{cond}} \rrbracket &= p[(\text{if } e \text{ then } q \oplus \text{LEFT}; \mathbf{0} \text{ else } q \oplus \text{RIGHT}; \mathbf{0}); \mathbf{0}] \\
 &\quad | \\
 &\quad q[p \&\{\text{LEFT} : \mathbf{0}, \text{RIGHT} : \mathbf{0}\}; \mathbf{0}]
 \end{aligned}$$

Let Σ be such that $\Sigma(p) \vdash e \downarrow \text{true}$. Then, by rule COND for choreographies we can execute the conditional at p as follows.

$$\langle C_{\text{cond}}, \Sigma \rangle \xrightarrow{\tau @ p} \langle p \rightarrow q.\text{LEFT}; \mathbf{0}, \Sigma \rangle \quad (5.10)$$

5.3. Selections

The corresponding transition in $\langle \llbracket C_{\text{cond}} \rrbracket, \Sigma \rangle$ executes the same conditional at p :

$$\langle \llbracket C_{\text{cond}} \rrbracket, \Sigma \rangle \xrightarrow{\tau @ p} \left\langle \begin{array}{c} p[q \oplus \text{LEFT}; 0] \\ | \\ q[p \& \{\text{LEFT} : 0, \text{RIGHT} : 0\}; 0] \end{array}, \Sigma \right\rangle. \quad (5.11)$$

Let C_{left} be the choreography on the right-hand side in eq. (5.10), i.e.,

$$C_{\text{left}} = p \rightarrow q.\text{LEFT}; 0,$$

and let N_{left} be the network on the right-hand side in eq. (5.11), i.e.,

$$N_{\text{left}} = \begin{array}{c} p[q \oplus \text{LEFT}; 0] \\ | \\ q[p \& \{\text{LEFT} : 0, \text{RIGHT} : 0\}; 0] \end{array}.$$

According to our previous statements of correctness of EPP, we should have that $\llbracket C_{\text{left}} \rrbracket = N_{\text{left}}$. But this is not the case, since we can see that the EPP of C_{left} is different:

$$\llbracket C_{\text{left}} \rrbracket = \llbracket p \rightarrow q.\text{LEFT}; 0 \rrbracket = \begin{array}{c} p[q \oplus \text{LEFT}; 0] \\ | \\ q[p \& \{\text{LEFT} : 0\}; 0] \end{array} \neq N_{\text{left}}.$$

Notice that the difference is in the branching term at process q : the projection of C_{left} has only the **LEFT** branch, whereas in N_{left} we still have both the **LEFT** and **RIGHT** branches. Generally speaking, this happens because when a conditional like *if* $p.e$ *then* C_1 *else* C_2 in a choreography is executed, all the code in the branch that is not chosen is “cut off” in a single step, and that code might mention many processes. Instead, at the level of networks, executing a conditional at process p removes code for p only, so the other processes may still have extra branches in their branching terms (as in our example here).

We now move to formalising our observations in a general way, obtaining a new statement for the correctness of EPP. First, we define formally what it means for processes and networks to have “extra branches” as the relation \sqsupseteq .

Definition 13. We write $P \sqsupseteq Q$ when P has at least as many branches in branchings as Q . Formally, \sqsupseteq is defined as follows.

- $P \sqsupseteq Q$ if and only if $P \sqcup Q = P$.

Definition 14. We write $N \sqsupseteq M$ when all processes in N have at least as many branches as in M . Formally, $N \sqsupseteq M$ if and only if, for all p , $N(p) \sqsupseteq M(p)$.

Proposition 7. *Relation \sqsubseteq for networks is a partial order; that is, it has the following properties.*

- *Reflexivity: $N \sqsubseteq N$ for all N .*
- *Antisymmetry: if $N \sqsubseteq M$ and $M \sqsubseteq N$, then $N = M$ for all N and M .*
- *Transitivity: if $N \sqsubseteq N'$ and $N' \sqsubseteq N''$, then $N \sqsubseteq N''$ for all N , N' , and N'' .*

Exercise 37 (!). *Prove proposition 7.*

Proposition 8. *If $N \sqsubseteq M$ or $M \sqsubseteq N$, then $\text{supp}(N) = \text{supp}(M)$.*

Lemma 11. *Let $N \sqsubseteq M$. If $\langle M, \Sigma \rangle \xrightarrow{\mu} \langle M', \Sigma' \rangle$ and $\langle N, \Sigma \rangle \xrightarrow{\mu} \langle N', \Sigma' \rangle$, then $N' \sqsubseteq M'$.*

Lemma 12. *Let $N \sqsubseteq M$. If $\langle M, \Sigma \rangle \xrightarrow{\mu} \langle M', \Sigma' \rangle$, then $\langle N, \Sigma \rangle \xrightarrow{\mu} \langle N', \Sigma' \rangle$ for some N' .*

Lemma 13. *Let $N \sqsubseteq \llbracket C \rrbracket$. If $\langle N, \Sigma \rangle \xrightarrow{\mu} \langle N', \Sigma' \rangle$ then $\langle \llbracket C \rrbracket, \Sigma \rangle \xrightarrow{\mu} \langle M, \Sigma' \rangle$ and $N' \sqsubseteq M$.*

Corollary 3. *Let $N \sqsubseteq M \sqsubseteq \llbracket C \rrbracket$. If $\langle N, \Sigma \rangle \xrightarrow{\mu} \langle N', \Sigma' \rangle$ then $\langle M, \Sigma \rangle \xrightarrow{\mu} \langle M', \Sigma' \rangle$ and $N' \sqsubseteq M'$.*

Proof. Say that $\langle N, \Sigma \rangle \xrightarrow{\mu} \langle N', \Sigma' \rangle$. By lemma 13, $\langle \llbracket C \rrbracket, \Sigma \rangle \xrightarrow{\mu} \langle N'', \Sigma' \rangle$ and $N' \sqsubseteq N''$. By lemma 12, $\langle M, \Sigma \rangle \xrightarrow{\mu} \langle M', \Sigma' \rangle$. By lemma 11, $N' \sqsubseteq M'$. \square

Lemma 14. *If $\langle C, \Sigma \rangle \xrightarrow{\mu} \langle C', \Sigma' \rangle$, then $\llbracket C \rrbracket_r \sqsubseteq \llbracket C' \rrbracket_r$ for all $r \notin \text{pn}(\mu)$.*

Lemma 15. *If $\langle N, \Sigma \rangle \xrightarrow{\mu} \langle N', \Sigma' \rangle$ and $p \notin \text{pn}(\mu)$, then $\langle N \setminus p, \Sigma \rangle \xrightarrow{\mu} \langle N' \setminus p, \Sigma' \rangle$.*

Exercise 38. *Prove lemma 14.*

Exercise 39. *Prove lemma 15.*

Lemma 16. *$P \sqsubseteq Q$ implies $I; P \sqsubseteq I; Q$ for all P , Q , and I .*

Proof. By definition of \sqsubseteq , we need to prove that

$$I; P \sqcup I; Q = I; P.$$

By the definition of \sqcup (fig. 5.12), this holds if $P \sqcup Q = P$, which is our initial assumption. \square

Lemma 17. *$P \sqsubseteq Q$ implies $R \circ P \sqsubseteq R \circ Q$ for all P , Q , and R .*

5.3. Selections

Proof. By induction on R .

Case $R = 0$. In this case, $R \circ P = P$ and $R \circ Q = Q$, so the thesis follows immediately by the assumption $P \sqsubseteq Q$.

Case $R = I; R'$. In this case, by definition of \circ ,

$$\begin{aligned} R \circ P &= I; (R' \circ P) \\ R \circ Q &= I; (R' \circ Q) \end{aligned} .$$

By induction hypothesis, $R' \circ P \sqsubseteq R' \circ Q$. The thesis follows by lemma 16.

□

Lemma 18. $\llbracket C \rrbracket_r \sqsubseteq \llbracket C' \rrbracket_r$ implies $\llbracket I; C \rrbracket_r \sqsubseteq \llbracket I; C' \rrbracket_r$ for all C, C', p , and I .

Proof. We proceed by cases on I .

Case $I = p.x := e$. If $r = p$, then the thesis follows by definition of projection and lemma 16. If $r \neq p$, then the thesis follows by definition of projection—since $\llbracket I; C \rrbracket_r = \llbracket C \rrbracket_r$ and $\llbracket I; C' \rrbracket_r = \llbracket C' \rrbracket_r$ —and the assumption $\llbracket C \rrbracket_r \sqsubseteq \llbracket C' \rrbracket_r$.

Case $I = p.e \rightarrow q.x$ or $I = p \rightarrow q.L$. Similar to the previous case, by distinguishing whether $r = p$, $r = q$, or $r \notin \{p, q\}$.

Case $I = \text{if } p.e \text{ then } C_1 \text{ else } C_2$. If $r = p$, then the thesis follows by definition of projection and lemma 16. If $r \neq p$, then

$$\begin{aligned} \llbracket I; C \rrbracket_r &= (\llbracket C_1 \rrbracket_r \sqcup \llbracket C_2 \rrbracket_r) \circ \llbracket C \rrbracket_r \\ \llbracket I; C' \rrbracket_r &= (\llbracket C_1 \rrbracket_r \sqcup \llbracket C_2 \rrbracket_r) \circ \llbracket C' \rrbracket_r . \end{aligned}$$

The thesis follows by lemma 17.

□

Lemma 19 (Associativity of \circ). For all P, Q , and R ,

$$P \circ (Q \circ R) = (P \circ Q) \circ R .$$

Proof. By induction on P .

Case $P = 0$. The thesis follows immediately by definition of \circ , since

$$\begin{aligned} P \circ (Q \circ R) &= Q \circ R \\ (P \circ Q) \circ R &= Q \circ R \quad (\text{because } P \circ Q = Q) . \end{aligned}$$

Case $P = I; P'$. By definition of \circ ,

$$\begin{aligned} P \circ (Q \circ R) &= I; (P' \circ (Q \circ R)) \\ (P \circ Q) \circ R &= (I; (P' \circ Q)) \circ R. \end{aligned}$$

Still by definition of \circ , we can rewrite the right-hand side of the second equation as follows.

$$(I; (P' \circ Q)) \circ R = I; ((P' \circ Q) \circ R)$$

So we are left with having to prove that

$$P' \circ (Q \circ R) = (P' \circ Q) \circ R,$$

which follows by induction hypothesis.

□

Lemma 20. For all C, C' , and r ,

$$\llbracket C \circ C' \rrbracket_r = \llbracket C \rrbracket_r \circ \llbracket C' \rrbracket_r.$$

Proof. By induction on C .

Case $C = 0$. The thesis follows from the definitions of projection and \circ , since in this case $C \circ C' = C'$ and $\llbracket C \rrbracket_r \circ \llbracket C' \rrbracket_r = \llbracket C' \rrbracket_r$.

Case $C = I; C_1$, where I is an assignment, a value communication, or a selection. We have a subcase for each possible shape that I can take.

If $I = p.x := e$, then

$$C \circ C' = p.x := e; (C_1 \circ C').$$

By induction hypothesis,

$$\llbracket C_1 \circ C' \rrbracket_r = \llbracket C_1 \rrbracket_r \circ \llbracket C' \rrbracket_r.$$

If $r = p$, then

$$\llbracket C \circ C' \rrbracket_r = (x := e; \llbracket C_1 \circ C' \rrbracket_r).$$

Otherwise, if $r \neq p$,

$$\llbracket C \circ C' \rrbracket_r = \llbracket C_1 \circ C' \rrbracket_r.$$

In both cases, the thesis follows by induction hypothesis.

The subcases when I is a communication or selection are similar to the previous one.

5.3. Selections

Case $C = \text{if } p.e \text{ then } C_1 \text{ else } C_2; C_3$. In this case,

$$C \circ C' = \text{if } p.e \text{ then } C_1 \text{ else } C_2; (C_3 \circ C').$$

We have two subcases now, either $r = p$ or $r \neq p$.

If $r = p$, then

$$\begin{aligned} \llbracket C \circ C' \rrbracket_r &= \text{if } e \text{ then } \llbracket C_1 \rrbracket_r \text{ else } \llbracket C_2 \rrbracket_r; \llbracket C_3 \circ C' \rrbracket_r \\ \llbracket C \rrbracket_r \circ \llbracket C' \rrbracket_r &= (\text{if } e \text{ then } \llbracket C_1 \rrbracket_r \text{ else } \llbracket C_2 \rrbracket_r; \llbracket C_3 \rrbracket_r) \circ \llbracket C' \rrbracket_r \end{aligned}$$

By definition of \circ , we can rewrite $\llbracket C \rrbracket_r \circ \llbracket C' \rrbracket_r$ as follows.

$$\begin{aligned} (\text{if } e \text{ then } \llbracket C_1 \rrbracket_r \text{ else } \llbracket C_2 \rrbracket_r; \llbracket C_3 \rrbracket_r) \circ \llbracket C' \rrbracket_r &= \\ \text{if } e \text{ then } \llbracket C_1 \rrbracket_r \text{ else } \llbracket C_2 \rrbracket_r; (\llbracket C_3 \rrbracket_r \circ \llbracket C' \rrbracket_r) \end{aligned}$$

The thesis follows by induction hypothesis.

If $r \neq p$, then

$$\begin{aligned} \llbracket C \circ C' \rrbracket_r &= (\llbracket C_1 \rrbracket_r \sqcup \llbracket C_2 \rrbracket_r) \circ \llbracket C_3 \circ C' \rrbracket_r \\ \llbracket C \rrbracket_r \circ \llbracket C' \rrbracket_r &= ((\llbracket C_1 \rrbracket_r \sqcup \llbracket C_2 \rrbracket_r) \circ \llbracket C_3 \rrbracket_r) \circ \llbracket C' \rrbracket_r \end{aligned}$$

By lemma 19, we can rewrite the right-hand side of the second equation as follows.

$$((\llbracket C_1 \rrbracket_r \sqcup \llbracket C_2 \rrbracket_r) \circ \llbracket C_3 \rrbracket_r) \circ \llbracket C' \rrbracket_r = (\llbracket C_1 \rrbracket_r \sqcup \llbracket C_2 \rrbracket_r) \circ (\llbracket C_3 \rrbracket_r \circ \llbracket C' \rrbracket_r)$$

Thus we need to prove that $\llbracket C_3 \circ C' \rrbracket_r = \llbracket C_3 \rrbracket_r \circ \llbracket C' \rrbracket_r$, which follows by induction hypothesis.

□

Theorem 5 (Correctness of EPP for stateful choreographies).

- (Completeness) For any C, Σ, μ, C' , and Σ' , $\langle C, \Sigma \rangle \xrightarrow{\mu} \langle C', \Sigma' \rangle$ implies $\langle \llbracket C \rrbracket, \Sigma \rangle \xrightarrow{\mu} \sqsupseteq \langle \llbracket C' \rrbracket, \Sigma' \rangle$.
- (Soundness) For any C, Σ, μ, N , and Σ' , $\langle \llbracket C \rrbracket, \Sigma \rangle \xrightarrow{\mu} \langle N, \Sigma' \rangle$ then $\langle C, \Sigma \rangle \xrightarrow{\mu} \langle C', \Sigma' \rangle$ for some C' such that $N \sqsupseteq \llbracket C' \rrbracket$.

Proof (Completeness part). We proceed by induction on the derivation of the transition $C \xrightarrow{\mu} C'$. We show only the interesting cases, i.e., those that involve merging. The other cases are similar to those in the previous proofs.

Case The derivation of the transition ends with an application of rule COND:

$$\frac{i = 1 \text{ if } \Sigma(\mathbf{p}) \vdash e \downarrow \text{true}, i = 2 \text{ otherwise}}{\langle \text{if } \mathbf{p}.e \text{ then } C_1 \text{ else } C_2; C_3, \Sigma \rangle \xrightarrow{\tau @ \mathbf{p}} \langle C_i \circ C_3, \Sigma \rangle} \text{COND}.$$

Let $\Sigma(\mathbf{p}) \vdash e \downarrow \text{true}$ (so we choose branch C_1); the other case (when we choose C_2) follows reasoning equivalent to the following.

By the definition of EPP,

$$\llbracket C \rrbracket = \llbracket C \rrbracket \setminus \mathbf{p} \mid \mathbf{p} \left[\text{if } e \text{ then } \llbracket C_1 \rrbracket_{\mathbf{p}} \text{ else } \llbracket C_2 \rrbracket_{\mathbf{p}}; \llbracket C_3 \rrbracket_{\mathbf{p}} \right].$$

By rules PAR and COND for networks, $\langle \llbracket C \rrbracket, \Sigma \rangle$ can mimic the choreographic transition:

$$\langle \llbracket C \rrbracket, \Sigma \rangle \xrightarrow{\tau @ \mathbf{p}} \langle \llbracket C \rrbracket \setminus \mathbf{p} \mid \mathbf{p} \left[\llbracket C_1 \rrbracket_{\mathbf{p}} \circ \llbracket C_3 \rrbracket_{\mathbf{p}} \right], \Sigma \rangle.$$

By lemma 14, $\llbracket C \rrbracket \setminus \mathbf{p} \sqsupseteq \llbracket C' \rrbracket \setminus \mathbf{p}$. By lemma 20, $\llbracket C_1 \rrbracket_{\mathbf{p}} \circ \llbracket C_3 \rrbracket_{\mathbf{p}} = \llbracket C_1 \circ C_3 \rrbracket_{\mathbf{p}}$.

Hence, $\langle \llbracket C \rrbracket, \Sigma \rangle \xrightarrow{\tau @ \mathbf{p}} \sqsupseteq \langle \llbracket C' \rrbracket, \Sigma \rangle$.

Case The derivation of the transition ends with an application of rule DELAY:

$$\frac{\langle C_1, \Sigma \rangle \xrightarrow{\mu} \langle C_2, \Sigma' \rangle \quad \text{pn}(I) \# \text{pn}(\mu)}{\langle I; C_1, \Sigma \rangle \xrightarrow{\mu} \langle I; C_2, \Sigma' \rangle} \text{DELAY}.$$

Let $\text{pn}(I) = \{\vec{\mathbf{p}}\}$, for some $\vec{\mathbf{p}}$ and n such that $\vec{\mathbf{p}} = \mathbf{p}_1, \dots, \mathbf{p}_n$. By definition of EPP,

$$\llbracket C \rrbracket = \llbracket C_1 \rrbracket \setminus \vec{\mathbf{p}} \mid \llbracket C_1 \rrbracket \upharpoonright_{\vec{\mathbf{p}}}.$$

By induction hypothesis,

$$\langle \llbracket C_1 \rrbracket, \Sigma \rangle \xrightarrow{\mu} \sqsupseteq \langle \llbracket C_2 \rrbracket, \Sigma' \rangle.$$

Since $\vec{\mathbf{p}}$ do not appear in μ (premise of DELAY above), by lemma 15 we obtain

$$\langle \llbracket C_1 \rrbracket \setminus \vec{\mathbf{p}}, \Sigma \rangle \xrightarrow{\mu} \langle N, \Sigma' \rangle \text{ for some } N \text{ such that } N \sqsupseteq \llbracket C_2 \rrbracket \setminus \vec{\mathbf{p}}. \quad (5.12)$$

We use this to make the following derivation:

$$\frac{\langle \llbracket C_1 \rrbracket \setminus \vec{\mathbf{p}}, \Sigma \rangle \xrightarrow{\mu} \langle N, \Sigma' \rangle}{\langle \llbracket C_1 \rrbracket \setminus \vec{\mathbf{p}} \mid \llbracket I; C_1 \rrbracket \upharpoonright_{\vec{\mathbf{p}}}, \Sigma \rangle \xrightarrow{\mu} \langle N \mid \llbracket I; C_1 \rrbracket \upharpoonright_{\vec{\mathbf{p}}}, \Sigma' \rangle} \text{PAR}. \quad (5.13)$$

5.3. Selections

From the premises of rule DELAY at the beginning and lemma 14, we have that

$$\llbracket C_1 \rrbracket \vdash_{\vec{p}} \sqsupseteq \llbracket C_2 \rrbracket \vdash_{\vec{p}} . \quad (5.14)$$

By eq. (5.14) and lemma 18,

$$\llbracket I; C_1 \rrbracket \vdash_{\vec{p}} \sqsupseteq \llbracket I; C_2 \rrbracket \vdash_{\vec{p}} . \quad (5.15)$$

By eq. (5.12), eq. (5.15), and the definition of \sqsupseteq :

$$N \mid \llbracket I; C_1 \rrbracket \vdash_{\vec{p}} \sqsupseteq \llbracket C_2 \rrbracket \setminus \vec{p} \mid \llbracket I; C_2 \rrbracket \vdash_{\vec{p}} ,$$

which proves the thesis.

□

Chapter 6

Recursion

All the choreographies that we have seen so far necessarily terminate, in the sense that their execution eventually reaches the terminated term 0 . In this chapter, we shall extend choreographies with the capability of expressing recursive behaviours—“repetitions” of communication structures—including possibly infinite behaviours.

Lemma 21. *For all C and p , if $p \notin \text{pn}(C)$ then $\llbracket C \rrbracket_p = 0$.*

6.1 Choreographies

6.1.1 Syntax

To express recursion, we add two ingredients to the syntax of choreographies: procedure definitions, ranged over by \mathcal{C} , and a new term for procedure calls in choreographies. The new syntax is displayed in fig. 6.1.

A set of procedure definitions \mathcal{C} is a (possibly empty) set of equations of the form $X(\vec{p}) = C$, read “procedure X has parameters \vec{p} and body C ”. We call the parameters \vec{p} of a procedure definition the *formal parameters* of the procedure. We make a few assumptions on sets of procedure definitions.

- All procedures defined in a set \mathcal{C} must have distinct names. This means that we can abstract from the order in which procedures are defined, and thus write $X(\vec{p}) = C \in \mathcal{C}$ to state that the definition $X(\vec{p}) = C$ is in \mathcal{C} .
- The list of formal parameters of a procedure never contains repetitions, i.e., all formal parameters are distinct.
- In every procedure definition $X(\vec{p}) = C$, C can mention only its formal parameters, formally $\text{pn}(C) \subseteq \{\vec{p}\}$.

$$\begin{aligned}
 C &::= I; C \mid \mathbf{0} \\
 I &::= p.e \rightarrow q.x \mid p.x := e \mid \text{if } p.e \text{ then } C_1 \text{ else } C_2 \mid X(\vec{p}) \\
 \mathcal{C} &::= X(\vec{p}) = C, \mathcal{C} \mid \emptyset
 \end{aligned}$$

Figure 6.1: Recursive choreographies, syntax.

- All procedure definitions have at least two parameters, i.e., for every $X(\vec{p}) = C$, \vec{p} is a sequence of at least two processes.

We also add a new kind of choreographic instruction, namely $X(\vec{p})$, read “run procedure X with arguments \vec{p} ”. Here as well, we make a few assumptions.

- In all procedure calls $X(\vec{p})$, all process names in \vec{p} are distinct.
- In the context of a set of procedure definitions \mathcal{C} , all procedure calls $X(\vec{p})$ are such that:
 - X is defined in \mathcal{C} , formally $X(\vec{q}) = C \in \mathcal{C}$ for some \vec{q} and C ;
 - the lengths of \vec{p} and \vec{q} are the same (every procedure invocation provides the right number of arguments).

Example 15. A typical way of transmitting a large file over a network is to split the file into multiple parts (or “chunks”, or packets) that are then re-assembled at the destination. We implement this strategy here.

First, we write a procedure S that streams a series of packets from a server s to a client c .

$$\begin{aligned}
 S(c, s) = & \text{ if } s.(n \leq \text{packets}(\text{file})) \text{ then} \\
 & \quad s \rightarrow c.\text{NEXT}; \\
 & \quad s.\text{mkPacket}(\text{file}, n) \rightarrow c.\text{packet}; \\
 & \quad c.\text{file} := \text{append}(\text{file}, \text{packet}); \\
 & \quad s.n := n + 1; \\
 & \quad S(c, s); \\
 & \quad \mathbf{0} \\
 & \text{ else} \\
 & \quad s \rightarrow c.\text{END}; \\
 & \quad \mathbf{0} \\
 & ; \mathbf{0}
 \end{aligned}$$

Procedure S is recursive. The guard of the conditional checks that variable n at s is smaller or equal than the number of packets that need to be sent to transmit

6.1. Choreographies

the local file at s . If so, s informs the client c that it will receive a packet. The expression $mkPacket(file, n)$ returns the n -th packet to send, which c stores in its local variable $packet$. Then, c appends the received packet to its local file (where it is aggregating the received packets), and s increments its local counter of sent packets n by 1. Finally, we recursively invoke S to send the remaining packets. When s finishes the packets to send ($n \leq packets(file)$), s sends the label `END` to c to inform it that the procedure will terminate.

Using S , we can write a choreography for sending a large file as chunks.

```
c.filename → s.filename;
s.file := readFile(filename);
c.file := emptyFile();
s.n := 1;
S(c, s); 0
```

Here, the client sends the filename of the file it wishes to download to s , which s uses to read the file from its filesystem. Then c initialises its own variable $file$ (for storing the received packets) and s initialises n to 1. Finally, we invoke S to perform the streaming.

Since we extended the syntax of choreographies, we have to update our definition of pn .

$$\begin{aligned}
pn(I; C) &= pn(I) \cup pn(C) \\
pn(0) &= \emptyset \\
pn(p.e \rightarrow q.x) &= \{p, q\} \\
pn(p \rightarrow q.L) &= \{p, q\} \\
pn(p.x := e) &= \{p\} \\
pn(\text{if } p.e \text{ then } C_1 \text{ else } C_2) &= \{p\} \cup pn(C_1) \cup pn(C_2) \\
pn(X(\vec{p})) &= \{\vec{p}\}.
\end{aligned}$$

6.1.2 Semantics

We extend the semantics of our choreography model to consider also the set of procedure definitions under which we are executing. This means that the definition of our reduction relation now depends on the set of procedure definitions \mathcal{C} that we are considering. We thus denote transitions as $\langle C, \Sigma, \mathcal{C} \rangle \xrightarrow{\mu} \langle C', \sigma', \mathcal{C} \rangle$ —notice the new ingredient \mathcal{C} .

Before delving into the new rules, it is useful to explore the path that led to their design. The simplest way of modelling a procedure call would be to have a

single rule, which just unfolds the definition of the procedure (its “body”) in the choreography that we are running. Here is how such a rule could be formulated.

$$\frac{X(\vec{q}) = C \in \mathcal{C}}{X(\vec{p}); C' \xrightarrow{\tau @ \vec{p}} C[\vec{p}/\vec{q}] \circ C'}$$

In the conclusion, $C[\vec{p}/\vec{q}]$ is the choreography C where each occurrence of a parameter q_i (for every i such that q_i is in \vec{q}) is replaced by the corresponding argument p_i . While the rule is pleasantly simple, the fact that we have a single transition with a label that involves multiple processes ($\tau @ \vec{p}$) is fishy: it denotes that all processes are entering the procedure at the same time, as if there were some underlying global coordination mechanism among processes. Such coordination violates the principle that processes run independently of one another. Instead, in practice we would observe that each process would begin the execution of (its part of) the procedure independently. This means that we should be able to observe a transition for each process “entering” the procedure, and that these transitions should be able to take place in any order.

To capture the fact that each process begins the execution of a procedure call independently, we shall use an auxiliary syntactic term, or *runtime* term. Intuitively, the purpose of a runtime term is just to represent an intermediate state in the execution of a program. Specifically, in our semantics, the syntax of choreographies is extended as follows.

$$I ::= \dots \mid X\langle\vec{p}\rangle^C$$

The new term $X\langle\vec{p}\rangle^C$ denotes that the processes \vec{p} still have to begin execution of procedure X . The choreography C in $X\langle\vec{p}\rangle^C$ is a mere annotation, in the sense that it will not influence our semantics. The reason for having annotations comes from a technical detail in the definition of projection, which we shall discuss later when we reach the definition of EPP for recursive choreographies. For now, it is fine to ignore them: as far as the semantics of choreographies is concerned, they are completely immaterial.

We can now define the semantics of recursive choreographies; the rules are displayed in fig. 6.2. The three new rules are CALLFIRST, CALLENTER, and CALLDONE. In essence, rule CALLFIRST states that any process involved in a procedure call may enter first and the procedure invocation term is transformed into the special runtime term that we have introduced. Rule CALLENTER captures that any process that still has to begin executing a procedure might start doing so. Finally, rule CALLDONE eliminates a runtime term for a procedure invocation when the last process initiates the execution of the procedure.

6.1. Choreographies

$$\begin{array}{c}
\frac{\Sigma(p) \vdash e \downarrow v}{\langle p.x := e; C, \Sigma, \mathcal{C} \rangle \xrightarrow{\tau @ p} \langle C, \Sigma [p.x \mapsto v], \mathcal{C} \rangle} \text{LOCAL} \\
\\
\frac{\Sigma(p) \vdash e \downarrow v}{\langle p.e \rightarrow q.x; C, \Sigma, \mathcal{C} \rangle \xrightarrow{p.v \rightarrow q} \langle C, \Sigma [q.x \mapsto v], \mathcal{C} \rangle} \text{COM} \\
\\
\frac{}{\langle p \rightarrow q.L; C, \Sigma, \mathcal{C} \rangle \xrightarrow{p \rightarrow q.L} \langle C, \Sigma, \mathcal{C} \rangle} \text{SEL} \\
\\
\frac{i = 1 \text{ if } \Sigma(p) \vdash e \downarrow \text{true}, i = 2 \text{ otherwise}}{\langle \text{if } p.e \text{ then } C_1 \text{ else } C_2; C, \Sigma, \mathcal{C} \rangle \xrightarrow{\tau @ p} \langle C_i; C, \Sigma, \mathcal{C} \rangle} \text{COND} \\
\\
\frac{X(\vec{q}) = C \in \mathcal{C} \quad r \in \vec{p}}{\langle X(\vec{p}); C', \Sigma, \mathcal{C} \rangle \xrightarrow{\tau @ r} \langle X(\vec{p} \setminus r)^{C'}; C[\vec{p}/\vec{q}]; C', \Sigma, \mathcal{C} \rangle} \text{CALLFIRST} \\
\\
\frac{r \in \vec{p} \quad \vec{p} \setminus r \text{ nonempty}}{\langle X(\vec{p})^{C'}; C, \Sigma, \mathcal{C} \rangle \xrightarrow{\tau @ r} \langle X(\vec{p} \setminus r)^{C'}; C, \Sigma, \mathcal{C} \rangle} \text{CALLEENTER} \\
\\
\frac{}{\langle X(p)^{C'}; C, \Sigma, \mathcal{C} \rangle \xrightarrow{\tau @ p} \langle C, \Sigma, \mathcal{C} \rangle} \text{CALLDONE} \\
\\
\frac{\langle C, \Sigma, \mathcal{C} \rangle \xrightarrow{\mu} \langle C', \Sigma', \mathcal{C} \rangle \quad \text{pn}(I) \# \text{pn}(\mu)}{\langle I; C, \Sigma, \mathcal{C} \rangle \xrightarrow{\mu} \langle I; C', \Sigma', \mathcal{C} \rangle} \text{DELAY} \\
\\
\frac{\langle C_1, \Sigma, \mathcal{C} \rangle \xrightarrow{\mu} \langle C'_1, \Sigma', \mathcal{C} \rangle \quad \langle C_2, \Sigma, \mathcal{C} \rangle \xrightarrow{\mu} \langle C'_2, \Sigma', \mathcal{C} \rangle \quad p \notin \text{pn}(\mu)}{\langle \text{if } p.e \text{ then } C_1 \text{ else } C_2; C, \Sigma, \mathcal{C} \rangle \xrightarrow{\mu} \langle \text{if } p.e \text{ then } C'_1 \text{ else } C'_2; C, \Sigma', \mathcal{C} \rangle} \text{DELAYCOND}
\end{array}$$

Figure 6.2: Recursive choreographies, semantics.

Example 16. The following procedure, PP , implements a ping-pong protocol between two processes p and q , which take turns in pinging each other by sending an empty message (represented by the constant value unit).

$$PP(p, q) = p.\text{unit} \rightarrow q.x; PP(q, p); 0$$

Notice how, in the recursive invocation of PP , we invert the order of arguments to make the two processes take turns. To see this with our semantics, say that we

have a choreography C and a set of procedure definitions \mathcal{C} defined as follows.

$$\begin{aligned} C &\triangleq PP(p, q); 0 \\ \mathcal{C} &\triangleq PP(p, q) = p.\text{unit} \rightarrow q.x; PP(q, p); 0 \end{aligned}$$

Then, for any store Σ , by rule **CALLFIRST** we have that the following two transitions are possible.

$$\begin{aligned} \langle C, \Sigma, \mathcal{C} \rangle &\xrightarrow{\tau@p} \langle PP\langle q \rangle^C; p.\text{unit} \rightarrow q.x; PP(q, p); 0, \Sigma, \mathcal{C} \rangle \\ \langle C, \Sigma, \mathcal{C} \rangle &\xrightarrow{\tau@q} \langle PP\langle p \rangle^C; p.\text{unit} \rightarrow q.x; PP(q, p); 0, \Sigma, \mathcal{C} \rangle \end{aligned}$$

Let us consider the derivative of the first transition. By rule **CALLDONE**, we have that

$$\langle PP\langle q \rangle^C; p.\text{unit} \rightarrow q.x; PP(q, p); 0, \Sigma, \mathcal{C} \rangle \xrightarrow{\tau@q} \langle p.\text{unit} \rightarrow q.x; PP(q, p); 0, \Sigma, \mathcal{C} \rangle .$$

We can now execute the communication (the first “ping”), by rule **COM**.

$$\langle p.\text{unit} \rightarrow q.x; PP(q, p); 0, \Sigma, \mathcal{C} \rangle \xrightarrow{p.\text{unit} \rightarrow q} \langle PP(q, p); 0, \Sigma[q.x \mapsto \text{unit}], \mathcal{C} \rangle .$$

We are now in a choreography similar to the one we started from, with the only difference that PP is invoked with arguments in reverse order. By rule **CALLENTER**, we can see that this means that now it is q that is going to ping p .

$$\begin{aligned} \langle PP(q, p); 0, \Sigma[q.x \mapsto \text{unit}], \mathcal{C} \rangle &\xrightarrow{\tau@q} \\ \langle PP\langle p \rangle^C; q.\text{unit} \rightarrow p.x; PP(p, q); 0, \Sigma[q.x \mapsto \text{unit}], \mathcal{C} \rangle \end{aligned}$$

By rules **CALLDONE** and **COM**, we get the following multi-step derivative.

$$\begin{aligned} \langle PP\langle p \rangle^C; q.\text{unit} \rightarrow p.x; PP(p, q); 0, \Sigma[q.x \mapsto \text{unit}], \mathcal{C} \rangle &\xrightarrow{\tau@q} \xrightarrow{q.\text{unit} \rightarrow p} \\ \langle PP(p, q); 0, \Sigma[q.x \mapsto \text{unit}][p.x \mapsto \text{unit}], \mathcal{C} \rangle \end{aligned}$$

So we got back to exactly our initial choreography C . The choreography is now going to continue indefinitely as shown above, alternating pings between p and q .

Remark 4. A transition never changes the set of procedure definitions (\mathcal{C}). This is different from choreographies (C) and stores (Σ), which can be updated by transitions. Indeed, in most programming systems, procedure definitions are fixed and never change during execution.

There are, however, examples of choreographic programming models where the code of procedures might change at runtime. This is useful, for example, when a system needs to adapt its code to a set of conditions that become clear only during execution. A model where the definitions of choreographies can evolve dynamically was presented by [Dalla Preda et al. \[2017\]](#).

6.2. Processes

$$\begin{aligned} P &::= I; P \mid 0 \\ I &::= p!e \mid p?x \mid p \oplus L \mid p \& \{L_i : P_i\}_{i \in I} \mid x := e \mid \text{if } e \text{ then } P \text{ else } Q \mid X(\vec{p}) \\ \mathcal{P} &::= X(\vec{p}) = P, \mathcal{P} \mid \emptyset \end{aligned}$$

Figure 6.3: Recursive processes, syntax.

6.2 Processes

The extension to our process model to include recursion is very similar to that for choreographies. The new syntax and semantics of processes are displayed in figs. 6.3 and 6.4. The additions are the new syntax term for invoking procedures and a rule in the semantics for unfolding the body of a procedure call. Notice that, in this case, only one rule is needed for the semantics of procedure calls (rule CALL). This is because a procedure call in the language of processes involves only one process.

$$\begin{array}{c}
\frac{\Sigma(\mathbf{p}) \vdash e \downarrow v}{\langle \mathbf{p}[\mathbf{q}!e; P] \mid \mathbf{q}[\mathbf{p}?x; Q], \Sigma, \mathcal{P} \rangle \xrightarrow{\mathbf{p}.v \rightarrow \mathbf{q}} \langle \mathbf{p}[P] \mid \mathbf{q}[Q], \Sigma[\mathbf{q}.x \mapsto v], \mathcal{P} \rangle} \text{COM} \\
\\
\frac{j \in I}{\langle \mathbf{p}[\mathbf{q} \oplus L_j; P] \mid \mathbf{q}[\mathbf{p} \& \{L_i : P_i\}_{i \in I}; Q], \Sigma, \mathcal{P} \rangle \xrightarrow{\mathbf{p} \rightarrow \mathbf{q}.L_j} \langle \mathbf{p}[P] \mid \mathbf{q}[P_j \circ Q], \Sigma, \mathcal{P} \rangle} \text{SEL} \\
\\
\frac{\Sigma(\mathbf{p}) \vdash e \downarrow v}{\langle \mathbf{p}[x := e; P], \Sigma, \mathcal{P} \rangle \xrightarrow{\tau @ \mathbf{p}} \langle \mathbf{p}[P], \Sigma[\mathbf{p}.x \mapsto v], \mathcal{P} \rangle} \text{LOCAL} \\
\\
\frac{i = 1 \text{ if } \Sigma(\mathbf{p}) \vdash e \downarrow \text{true}, i = 2 \text{ otherwise}}{\langle \mathbf{p}[\text{if } e \text{ then } P_1 \text{ else } P_2; Q], \Sigma, \mathcal{P} \rangle \xrightarrow{\tau @ \mathbf{p}} \langle \mathbf{p}[P_i \circ Q], \Sigma, \mathcal{P} \rangle} \text{COND} \\
\\
\frac{X(\vec{r}) = Q \in \mathcal{P}}{\langle \mathbf{p}[X(\vec{q}); P], \Sigma, \mathcal{P} \rangle \xrightarrow{\tau @ \mathbf{p}} \langle \mathbf{p}[Q[\vec{q}/\vec{r}] \circ P], \Sigma, \mathcal{P} \rangle} \text{CALL} \\
\\
\frac{\langle N, \Sigma, \mathcal{P} \rangle \xrightarrow{\mu} \langle N', \Sigma', \mathcal{P} \rangle}{\langle N \mid M, \Sigma, \mathcal{P} \rangle \xrightarrow{\mu} \langle N' \mid M, \Sigma', \mathcal{P} \rangle} \text{PAR}
\end{array}$$

Figure 6.4: Recursive processes, semantics.

Appendix A

Solution to selected exercises

We give the solutions to some selected exercises, pointing out the main aspects. Some solutions are given in full detail, to serve as examples of exposition.

Solution of exercise 3. We prove only the direction from the system in fig. 1.1 to the system in fig. 1.2.

To prove this direction, we actually prove the stronger statement:

- if $\text{conn}(A, B)$ is derivable in the system in fig. 1.1, then $\text{conn}(A, B)$ is derivable in the system in fig. 1.2;
- if $\text{path}(A, B)$ is derivable in the system in fig. 1.1, then there exists a natural number n such that $\text{path}(A, B, n)$ is derivable in the system in fig. 1.2.

We proceed by induction on the structure of the derivation of $\text{conn}(A, B)$ or $\text{path}(A, B)$ in the system in fig. 1.1. We get a case for each rule that could be applied last in the derivation.

Base cases (axioms) If the last applied rule is an axiom, then the proof is valid also in the other system directly, since the two systems share the same axioms.

Case SYM The derivation has this shape:

$$\frac{\mathcal{D} \text{ conn}(B, A)}{\text{conn}(A, B)} \text{ SYM} .$$

By induction hypothesis on the sub-derivation \mathcal{D} , we know that there exists \mathcal{D}' in the other system such that:

$$\mathcal{D}' \text{ conn}(B, A) .$$

The thesis follows by applying rule SYM:

$$\frac{\mathcal{D}'}{\frac{\text{conn}(B, A)}{\text{conn}(A, B)}} \text{SYM} .$$

Case DIR The derivation has this shape:

$$\frac{\mathcal{D}}{\frac{\text{conn}(A, B)}{\text{path}(A, B)}} \text{DIR} .$$

By induction hypothesis, we know that there exists \mathcal{D}' in the other system such that:

$$\frac{\mathcal{D}'}{\text{conn}(A, B)} .$$

The thesis follows by applying rule DIRW.

$$\frac{\mathcal{D}'}{\frac{\text{conn}(A, B)}{\text{path}(A, B, 1)}} \text{DIRW} .$$

Case TRANS The derivation has this shape:

$$\frac{\frac{\mathcal{D}}{\text{path}(A, B)} \quad \frac{\mathcal{E}}{\text{path}(B, C)}}{\text{path}(A, C)} \text{TRANS} .$$

By induction hypothesis on \mathcal{D} and by induction hypothesis on \mathcal{E} , we know that there exist natural numbers n and m , and derivations \mathcal{D}' and \mathcal{E}' in the other system such that:

$$\frac{\mathcal{D}'}{\text{path}(A, B, n)} \quad \frac{\mathcal{E}'}{\text{path}(A, B, m)} .$$

The thesis follows by applying rule TRANSW.

$$\frac{\frac{\mathcal{D}'}{\text{path}(A, B, n)} \quad \frac{\mathcal{E}'}{\text{path}(A, B, m)}}{\text{path}(A, B, n + m)} \text{TRANSW} .$$

■

Solution of exercise 10. We prove the stronger result that, for any choreography C , either:

-
1. $C = 0$; or,
 2. $C \neq 0$, in which case there exists $\vec{\mu}$ such that $C \xrightarrow{\vec{\mu}} 0$.

We proceed by induction on the structure of C .

Case $C = 0$. In this case, the thesis follows trivially.

Case $C = p \rightarrow q; C'$ for some C' . By rule COM, we obtain $C \xrightarrow{p \rightarrow q} C'$.

If $C' = 0$, then the thesis follows by $\vec{\mu} = p \rightarrow q$. Otherwise, if $C' \neq 0$, by induction hypothesis there exists $\vec{\mu}'$ such that $C' \xrightarrow{\vec{\mu}'} 0$. Hence, the thesis follows for the sequence of labels $\vec{\mu} = p \rightarrow q, \vec{\mu}'$.

■

Solution of exercise 16. By induction on the derivation of the transition $N \xrightarrow{\mu} N'$.

Case The derivation of the transition ends with an application of rule COM:

$$\frac{}{p[q!; P] \mid q[p?; Q] \xrightarrow{p \rightarrow q} p[P] \mid q[Q]} \text{COM},$$

for $N = p[q!; P] \mid q[p?; Q]$.

Trivially, in this case $N(r) = 0 = N'(r)$ for all r different than p and q .

Case The derivation of the transition ends with an application of rule PAR:

$$\frac{N_1 \xrightarrow{\mu} N_2}{N_1 \mid M \xrightarrow{\mu} N_2 \mid M} \text{PAR},$$

for $N = N_1 \mid M$.

By induction hypothesis, $N_1(r) = N_2(r)$ for all r that do not appear in μ . The thesis follows because parallel composition requires disjoint supports and the context M does not change in the derivative.

■

Solution of exercise 22. By induction on the derivation of the transition $C \xrightarrow{\mu} C'$.

Case The derivation of the transition ends with an application of rule COM:

$$\frac{}{p \rightarrow q; C' \xrightarrow{p \rightarrow q} C'} \text{COM}.$$

The thesis follows by definition 7, since for all $r \notin \{p, q\}$, $\llbracket p \rightarrow q; C' \rrbracket_r = \llbracket C' \rrbracket_r$.

Case The derivation of the transition ends with an application of rule DELAY:

$$\frac{C_1 \xrightarrow{\mu} C_2 \quad \{p, q\} \# \text{pn}(\mu)}{p \rightarrow q; C_1 \xrightarrow{\mu} p \rightarrow q; C_2} \text{DELAY} .$$

By induction hypothesis, $\llbracket C_1 \rrbracket_r = \llbracket C_2 \rrbracket_r$. We have three cases, depending on whether r is p , q , or another process name.

For $r = p$, by definition 7 we obtain

$$\llbracket p \rightarrow q; C_1 \rrbracket_p = q!; \llbracket C_1 \rrbracket_p = q!; \llbracket C_2 \rrbracket_p = \llbracket p \rightarrow q; C_2 \rrbracket_p .$$

Similarly, for $r = q$, by definition 7 we obtain

$$\llbracket p \rightarrow q; C_1 \rrbracket_q = p?; \llbracket C_1 \rrbracket_q = p?; \llbracket C_2 \rrbracket_q = \llbracket p \rightarrow q; C_2 \rrbracket_q .$$

When $r \notin \{p, q\}$ we have the simplest case. Again, by definition 7,

$$\llbracket p \rightarrow q; C_1 \rrbracket_r = \llbracket C_1 \rrbracket_r = \llbracket C_2 \rrbracket_r = \llbracket p \rightarrow q; C_2 \rrbracket_r .$$

■

List of Figures

1.1	An inference system for flights.	13
1.2	Weighted rules for flight paths.	17
1.3	A limited and weighted flight system.	19
1.4	An alternative way of constructing paths.	20
1.5	A weighted version of the system in fig. 1.4.	26
2.1	Simple choreographies, syntax.	30
2.2	Simple choreographies, semantics.	32
3.1	Simple processes, syntax.	36
3.2	Simple processes, semantics.	39
3.3	Process projection for simple choreographies.	42
3.4	Simple concurrent choreographies, syntax.	46
3.5	Simple concurrent choreographies, semantics.	47
4.1	Expressions.	56
4.2	Expression evaluation.	57
4.3	Stateful choreographies, syntax.	58
4.4	Stateful choreographies, semantics.	59
4.5	Stateful processes, syntax.	61
4.6	Stateful processes, semantics.	62
4.7	Process projection for stateful choreographies.	62
5.1	Choreographies with conditionals, syntax.	68
5.2	Choreographies with conditionals, semantics.	68
5.3	Choreographies with conditionals, § operator (“fat semicolon”).	69
5.4	Processes with conditionals, syntax.	70
5.5	Processes with conditionals, semantics.	70
5.6	Processes with conditionals, operator §.	70
5.7	Process projection for choreographies with conditionals.	73
5.8	Choreographies with selections, syntax.	76
5.9	Choreographies with selections, semantics.	77

LIST OF FIGURES

5.10	Processes with selections, syntax.	78
5.11	Processes with selections, semantics.	78
5.12	Merging operator for processes with selections.	80
6.1	Recursive choreographies, syntax.	90
6.2	Recursive choreographies, semantics.	93
6.3	Recursive processes, syntax.	95
6.4	Recursive processes, semantics.	96

List of Notations

C A choreography. 30

N A network. 37

$\llbracket C \rrbracket_p$ The projection of the behaviour of process p in choreography C . 42

$\llbracket C \rrbracket$ The EndPoint Projection (EPP) of choreography C . 43

μ A label of a labelled transition system. 31

τ The internal action of a labelled transition system. 60

\longrightarrow The transition relation. 31

! A (possibly) difficult exercises. 10

\hookrightarrow An exercise for which a solution is provided in appendix A. 10

p A process name, also called process identifier (pid for short). 30

pn The function that computes the set of process names in a term. 43

supp The support of a function. 37

Index

μ -derivative, 33

admissible rule, 22

axiom, 11

choreographic store, 55

complete derivation, 16

configuration, 59

derivable rule, 20

derivation, 16

derivative, 33

EndPoint Projection (EPP), 42

expression, 56

expression evaluation, 56

finite support, 37

function identifier, 56

inference rule, 11

inference system, 11

internal action, 60

labelled transition system (lts), 31

multi-step derivative, 33

parallel operator, 37

pid, 30

process identifier, 30

process projection, 42

process store, 55

support, 37

variables, 55

Bibliography

- Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral types in programming languages. *Foundations and Trends in Programming Languages*, 3(2-3):95–230, 2016.
- Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *POPL*, pages 263–274. ACM, 2013.
- Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. In Rocco De Nicola, editor, *ESOP*, volume 4421 of *LNCS*, pages 2–17. Springer, 2007. doi: 10.1007/978-3-540-71316-6_2.
- Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2):8, 2012.
- Luís Cruz-Filipe and Fabrizio Montesi. Procedural choreographic programming. In *FORTE*, LNCS. Springer, 2017.
- Luís Cruz-Filipe and Fabrizio Montesi. A core model for choreographic programming. *Theoretical Computer Science*, 2019. ISSN 0304-3975. doi: <https://doi.org/10.1016/j.tcs.2019.07.005>. URL <http://www.sciencedirect.com/science/article/pii/S0304397519304311>.
- Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Dynamic choreographies: Theory and implementation. *Logical Methods in Computer Science*, 13(2), 2017.
- Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In *ICALP (2)*, pages 174–186, 2013.

BIBLIOGRAPHY

- Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*, pages 195–216. Springer, 2017.
- Xiang Fu, Tevfik Bultan, and Jianwen Su. Realizability of conversation protocols with message contents. *Int. J. Web Service Res.*, 2(4):68–93, 2005a.
- Xiang Fu, Tevfik Bultan, and Jianwen Su. Realizability of conversation protocols with message contents. *International Journal on Web Service Res.*, 2(4):68–93, 2005b.
- Matthew Hennessy. *A distributed Pi-calculus*. Cambridge University Press, 2007.
- Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. *J. ACM*, 63(1):9, 2016. doi: 10.1145/2827695. URL <http://doi.acm.org/10.1145/2827695>.
- Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniérou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016.
- International Telecommunication Union. Recommendation Z.120: Message sequence chart, 1996.
- Ivan Lanese, Claudio Guidi, Fabrizio Montesi, and Gianluigi Zavattaro. Bridging the gap between interaction- and process-oriented choreographies. In *SEFM*, pages 323–332, 2008.
- Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic journal of philosophical logic*, 1(1):11–60, 1996.
- Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, Berlin, 1980.
- R.M. Needham and M.D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, December 1978. ISSN 0001-0782. doi: 10.1145/359657.359659.
- Object Management Group. Business Process Model and Notation. <http://www.omg.org/spec/BPMN/2.0/>, 2011.

BIBLIOGRAPHY

- OECD. Horizon scan of megatrends and technology trends in the context of future research policy, 2016. <http://ufm.dk/en/publications/2016/an-oecd-horizon-scan-of-megatrends-and-technology-trends-in-the-context-of-future-research-policy>.
- F. Pfenning. Lecture Notes on Deductive Inference, 2012. <https://www.cs.cmu.edu/fp/courses/15816-s12/lectures/01-inference.pdf>.
- Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.
- Z. Qiu, X. Zhao, C. Cai, and H. Yang. Towards the theoretical foundation of choreography. In *WWW*, pages 973–982. ACM, 2007.
- R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978. ISSN 0001-0782. doi: 10.1145/359340.359342. URL <http://doi.acm.org/10.1145/359340.359342>.
- Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2011. doi: 10.1017/CBO9780511777110.
- W3C WS-CDL Working Group. Web services choreography description language version 1.0. <http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/>, 2004.