Introduction to Choreographies

Fabrizio Montesi
fmontesi@imada.sdu.dk
Department of Mathematics and Computer Science
University of Southern Denmark

Last update: November 13, 2018

This document contains lecture notes on choreographies for the course on DM861 Concurrency Theory (2018) at the University of Southern Denmark. It is organised as a working book. Different parts get updated during the course.

The chapters in advance of the lectures are given as a preview on what is to come, but **remember to download the updated version of this book every week to stay up to date!**

Contents

Preface:
Alice, Bob, and Choreographies

We live in the era of concurrency and distribution. Concurrency gives us the ability to perform multiple tasks at a time. Distribution allows us to do so at a distance. It is the era of the web, mobile computers, cloud computing, and microservices. The number of computer programs that communicate with other programs over a network is exploding. By 2025, the Internet alone might is expected to connect from 25 to 100 billion devices [OECD, 2016].

Modern computer networks and their applications are key 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.

Modern computer networks and their applications are complex. Services are becoming increasingly dependent on other services. 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 brower, 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 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, because each party needs to know what it is supposed to do such that integration is successful and the whole system works. At the same time, it is important that protocols are as concise as possible: the bigger a protocol, the higher the chance that we made 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 might say that we need a good *notation*.

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

$$A \texttt{->} B : M.$$

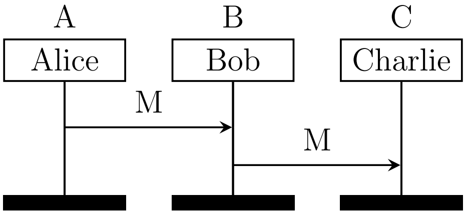
To define a protocol where A sends a message M to B, and then B passes the same message to C, we can just compose communications in a sequence:

$$\begin{array}{l} A \texttt{->} B : M \\ B \texttt{->} C : M. \end{array}$$

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 for 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 the case in general: not all notations are equally expressive, as we will see in the rest of this book.[1]

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 is essentially still a protocol, but the emphasis is on detail. Choreographies might include details like the following.

- The kind of data being transmitted, or even the actual functions used to compute the data to be transmitted.
- Explicit cause-effect relations (also known as causal dependencies) between the data being transmitted and the choices made by participants in a protocol.
- The state of participants, e.g., their memory states.

Choreographies are typically written in languages designed to be readable *mechanically*, which also make them amenable to be used by a computer. 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 started emerging 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. In 2011, the global technology standards consortium Object Management Group (OMG) introduced choreographies in their notation for business processes [BPMN]. The recent paradigm of microservices [Dragoni et al., 2017] advocates for 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].

It is the time of Alice and Bob. It is the time of choreographies.

---

[1] Message sequence charts in particular support many features, including timeouts and alternative behaviours.

This book

This book is an introduction to the theory of choreographies. It explains what choreographies are and how we can model them mathematically. Its primary audiences are computer science students and researchers. However, the book should be approachable by anybody interested in studying choreographies, e.g., for theoretical purposes or the development of choreography-based tools. The aim of this book is to be pedagogical. It is not an aim to be comprehensive. References to alternative notations and techniques to model choreographies are given where appropriate. It is assumed that the reader is familiar with the notion of concurrency and 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 in terms of context-free grammars. To give meaning to choreographies, we are going to use extensively Plotkin's structural approach to operational semantics.

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 not a requirement to read this book: chapter 1 provides an 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 ideas from Milner's seminal calculus of communicating systems [Milner, 1980].

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]

An inference system is a set of *inference rules* (also called rules of inference). An inference rule has the form

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

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.

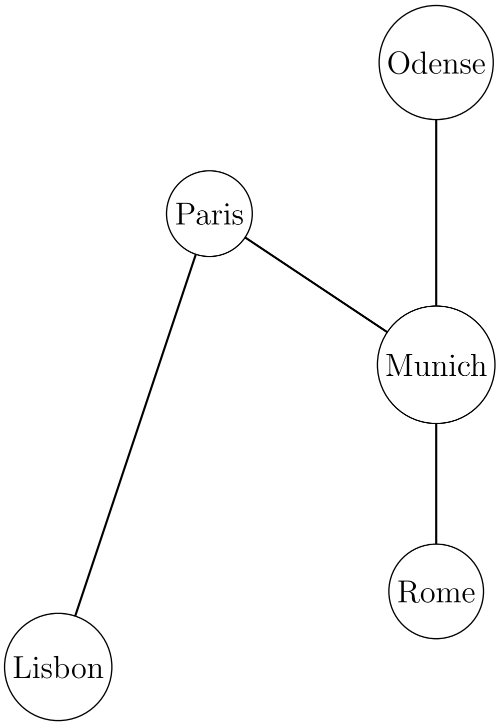
1.1 Example: Flight connections

This example is inspired by the usage of rules of inference to model graphs by Pfenning [2012].

Consider the following undirected graph of direct flights between cities.

---

[1] For 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* $\mathsf{conn}(A, B)$. Then, we can represent our graph as the set of axioms below.

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

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

Notice that our graph is undirected: in an ideal world, all direct flights are available also in the opposite direction. This is not faithfully represented by our axioms: if $\mathsf{conn}(A, B)$, it should also be the case that $\mathsf{conn}(B, A)$. One option to solve this discrepancy is to double the number of our axioms, to include their symmetric versions—e.g., for $\mathsf{conn}(\text{Munich}, \text{Odense})$). A more elegant option is to include a rule of inference for symmetry, as follows.

$$\frac{\mathsf{conn}(A, B)}{\mathsf{conn}(B, A)}\ \textsc{Sym}$$

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 us that if we have a connection from any A to any B (premise),

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

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

$$\frac{\mathsf{conn}(A, B)}{\mathsf{conn}(B, A)}\ \textsc{Sym} \qquad \frac{\mathsf{conn}(A, B)}{\mathsf{path}(A, B)}\ \textsc{Dir} \qquad \frac{\mathsf{path}(A, B) \quad \mathsf{path}(B, C)}{\mathsf{path}(A, C)}\ \textsc{Trans}$$

Figure 1.1: An inference system for flights.

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.

Now that we are satisfied with how our rule system captures the graph, we can use it to find flight paths from a city to another. For any two cities A and B, the proposition $\mathsf{path}(A, B)$ 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. (DIR stands for direct.)

$$\frac{\mathsf{conn}(A, B)}{\mathsf{path}(A, B)}\ \textsc{Dir}$$

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 in the following rule stands for transitivity.)

$$\frac{\mathsf{path}(A, B) \quad \mathsf{path}(B, C)}{\mathsf{path}(A, C)}\ \textsc{Trans}$$

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 $\mathsf{path}(\text{Odense}, \text{Rome})$ holds. To do this, we build a *derivation tree* (also simply called *derivation*, or *proof tree*). The problem tackled in the following is also known as proof search.

We start our derivation from what we want to conclude with (our conclusion is what we want to prove).

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

Observe that we have only two inference rules that can conclude something of this form: Dir and Trans. Dir cannot be applied: we would need to prove $\mathsf{conn}(\text{Odense}, \text{Rome})$, which is impossible. So our only choice is rule Trans, by instantiating A as Odense and C as Rome. How should we instantiate B in rule Trans, though?

$$\dfrac{\mathsf{path}(\text{Odense}, ??B??) \qquad \mathsf{path}(??B??, \text{Rome})}{\mathsf{path}(\text{Odense}, \text{Rome})}\ \textsc{Trans}$$

We need to guess a correct instantiation for B and hope that it will allow us to continue our derivation. Looking at the definition of our graph, it is easy to see that the right choice is to pick Munich.

$$\dfrac{\mathsf{path}(\text{Odense}, \text{Munich}) \qquad \mathsf{path}(\text{Munich}, \text{Rome})}{\mathsf{path}(\text{Odense}, \text{Rome})}\ \textsc{Trans}$$

Our derivation is not complete yet, because we are left with the tasks of proving (deriving) $\mathsf{path}(\text{Odense}, \text{Munich})$ and $\mathsf{path}(\text{Munich}, \text{Rome})$. Thanks to the notation of inference rules, we can just "dig deeper" with further rule applications. Since we know that Odense and Munich are connected, we can try using rule Dir.

$$\dfrac{\dfrac{\mathsf{conn}(\text{Odense}, \text{Munich})}{\mathsf{path}(\text{Odense}, \text{Munich})}\ \textsc{Dir} \qquad \mathsf{path}(\text{Munich}, \text{Rome})}{\mathsf{path}(\text{Odense}, \text{Rome})}\ \textsc{Trans}$$

We now have to prove $\mathsf{conn}(\text{Odense}, \text{Munich})$. 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.

$$\dfrac{\dfrac{\dfrac{}{\mathsf{conn}(\text{Odense}, \text{Munich})}}{\mathsf{path}(\text{Odense}, \text{Munich})}\ \textsc{Dir} \qquad \mathsf{path}(\text{Munich}, \text{Rome})}{\mathsf{path}(\text{Odense}, \text{Rome})}\ \textsc{Trans}$$

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

$$\dfrac{\dfrac{\dfrac{}{\mathsf{conn}(\text{Odense}, \text{Munich})}}{\mathsf{path}(\text{Odense}, \text{Munich})}\,\textsc{Dir} \qquad \dfrac{\dfrac{}{\mathsf{conn}(\text{Munich}, \text{Rome})}}{\mathsf{path}(\text{Munich}, \text{Rome})}\,\textsc{Dir}}{\mathsf{path}(\text{Odense}, \text{Rome})}\,\textsc{Trans}$$

Our derivation is done! We can tell by the fact that there are no more premises left to prove.

If you look carefully, you can see that our derivation is a tree. The conclusion $\mathsf{path}(\text{Odense}, \text{Rome})$ is the root of the tree, and the leaves are empty nodes (the premises of our axioms). Rule applications connect the nodes of the tree. In fact, even our previous partial derivations are all trees. This is a very useful property that we will use throughout the book.

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

1.3 Proof non-existence

Proof, or proof of no proof. There is no try.
- Yoda, to an exhausted Luke

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.

What about showing that a proposition *cannot* be derived? This can be far trickier, because it requires us to reason about all the possible proofs that could, potentially, conclude with our proposition and showing that none of those proofs can actually be built.

Consider a simple example: showing that $\mathsf{conn}(\text{Lisbon}, \text{Rome})$ 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 $\mathsf{conn}(A, B)$ for any A and B are our axioms and rule Sym. None of the axioms can be applied for $A = \text{Lisbon}$ and $B = \text{Rome}$, as in our case. We are left with rule Sym: any search of a derivation of $\mathsf{conn}(\text{Lisbon}, \text{Rome})$ must begin as follows.

$$\dfrac{\mathsf{conn}(\text{Rome}, \text{Lisbon})}{\mathsf{conn}(\text{Lisbon}, \text{Rome})}\,\textsc{Sym}$$

By similar reasoning (there is no rule to apply for $\mathsf{conn}(\text{Rome}, \text{Lisbon})$ but Sym), we obtain that the proof *must* continue with another application of Sym.

$$\dfrac{\dfrac{\mathsf{conn}(\text{Lisbon}, \text{Rome})}{\mathsf{conn}(\text{Rome}, \text{Lisbon})}\ \textsc{Sym}}{\mathsf{conn}(\text{Lisbon}, \text{Rome})}\ \textsc{Sym}$$

We got back to where we started: we have to prove $\mathsf{conn}(\text{Lisbon}, \text{Rome})$. 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, $\mathsf{conn}(\text{Lisbon}, \text{Rome})$ cannot be proven.

Let us be more formal, to convince ourselves that $\mathsf{conn}(\text{Lisbon}, \text{Rome})$ 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 $\mathsf{conn}(\text{Lisbon}, \text{Rome})$, there is at least one of minimal height, in the sense that there is no other proof of $\mathsf{conn}(\text{Lisbon}, \text{Rome})$ 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.

$$\dfrac{\dfrac{\mathsf{conn}(\text{Lisbon}, \text{Rome})}{\mathsf{conn}(\text{Rome}, \text{Lisbon})}\ \textsc{Sym}}{\mathsf{conn}(\text{Lisbon}, \text{Rome})}\ \textsc{Sym}$$

So we have to find some proof of our premise $\mathsf{conn}(\text{Lisbon}, \text{Rome})$ on top. But if such a proof exists, it would be a proof of $\mathsf{conn}(\text{Lisbon}, \text{Rome})$ 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.

Another way of proving that $\mathsf{conn}(\text{Lisbon}, \text{Rome})$ is by looking at how proofs are constructed from the top, rather than the bottom. For our system, we can prove the following result. We use $\mathcal{P}$ to range over proofs (derivations).

Proposition 1. *For any natural number n, there exists no proof of* $\mathsf{conn}(\mathit{Lisbon}, \mathit{Rome})$ *of height n.*

Proof. We prove the stronger result that there exists no proof of $\mathsf{conn}(\text{Lisbon}, \text{Rome})$ and there exists no proof of $\mathsf{conn}(\text{Rome}, \text{Lisbon})$, either.

We proceed by induction on n.

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

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

$$\frac{\mathsf{conn}(A, B)}{\mathsf{conn}(B, A)}\ \textsc{Sym}$$

$$\frac{\mathsf{conn}(A, B)}{\mathsf{path}(A, B, 1)}\ \textsc{DirW} \qquad \frac{\mathsf{path}(A, B, n) \quad \mathsf{path}(B, C, m)}{\mathsf{path}(A, C, n + m)}\ \textsc{TransW}$$

Figure 1.2: Weighted rules for flight paths.

Base case: $n = 1$. All proofs with height 1 must necessarily be an application of an axiom, and there is no axiom that can prove either $\mathsf{conn}(\text{Lisbon}, \text{Rome})$ or $\mathsf{conn}(\text{Rome}, \text{Lisbon})$.

Inductive case: $n = m + 1$ for some natural number m. By induction hypothesis, we know that there is no proof $\mathcal{P}$ such that the height of $\mathcal{P}$ is m and that the conclusion of $\mathcal{P}$ is $\mathsf{conn}(\text{Lisbon}, \text{Rome})$ or $\mathsf{conn}(\text{Rome}, \text{Lisbon})$. Thus, all proofs $\mathcal{Q}$ of height m conclude with either: i) $\mathsf{conn}(A, B)$ for some A and B such that the set $\{A, B\}$ is different from $\{\text{Lisbon}, \text{Rome}\}$; or ii) $\mathsf{path}(A, B)$ for some A and B. For i), we observe that there is no rule that allows us to derive $\mathsf{conn}(\text{Lisbon}, \text{Rome})$ from a premise that is not $\mathsf{conn}(\text{Rome}, \text{Lisbon})$, and likewise for the symmetric case where the premise is $\mathsf{conn}(\text{Rome}, \text{Lisbon})$. For ii), we observe that there is no rule that, given a conn proposition as one of its premises, allows us to conclude $\mathsf{conn}(\text{Lisbon}, \text{Rome})$ or $\mathsf{conn}(\text{Rome}, \text{Lisbon})$.

$\square$

It follows from proposition 1 that there is no proof of $\mathsf{conn}(\text{Lisbon}, \text{Rome})$. Assume that there were such a proof. Since it would have to be finite, it would have a height n. Thus we reach a contradiction, because proposition 1 states that for all n there is no proof of $\mathsf{conn}(\text{Lisbon}, \text{Rome})$.

Exercise 1. *Consider the system in fig. 1.2, which replaces rules* Dir *and* Trans *with alternative rules that measure the length of a path (paths are "weighted", with each connection having weight* 1*).*

$$\frac{}{\mathsf{conn}(\text{Odense}, \text{Munich})} \qquad \frac{}{\mathsf{conn}(\text{Munich}, \text{Rome})} \qquad \frac{}{\mathsf{conn}(\text{Paris}, \text{Lisbon})}$$

$$\frac{\mathsf{conn}(A, B)}{\mathsf{conn}(B, A)}\ \textsc{Sym}$$

$$\frac{\mathsf{conn}(A, B)}{\mathsf{path}(A, B, 1)}\ \textsc{DirW} \qquad \frac{\mathsf{path}(A, B, n) \quad \mathsf{path}(B, C, m)}{\mathsf{path}(A, C, n + m)}\ \textsc{TransW}$$

Figure 1.3: A limited and weighted flight system.

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

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

$$\frac{\mathsf{conn}(A, B)}{\mathsf{conn}(B, A)}\ \textsc{Sym} \qquad \frac{\mathsf{conn}(A, B)}{\mathsf{path}(A, B)}\ \textsc{Dir} \qquad \frac{\mathsf{conn}(A, B) \quad \mathsf{path}(B, C)}{\mathsf{path}(A, C)}\ \textsc{Step}$$

Figure 1.4: An alternative way of constructing paths.

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

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

Exercise 2 (!). *Consider the system in fig. 1.3, which removes the direct flight from Paris to Munich.*

Prove that it is not possible to derive $\mathsf{path}(\mathit{Lisbon}, \mathit{Munich}, n)$ *for any* n.

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, 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: $\mathsf{conn}(A, B)$ and $\mathsf{path}(A, B)$. It is easy to see that, for any A and B, $\mathsf{conn}(A, B)$ 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 $\mathsf{path}(A, B)$, 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).

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.

$$\frac{\mathsf{conn}(A, B) \quad \mathsf{path}(B, C)}{\mathsf{path}(A, C)}\ \textsc{Step}$$

The key observation here is that we can build a derivation that, from the premises $\mathsf{conn}(A, B)$ and $\mathsf{path}(B, C)$, concludes $\mathsf{path}(A, C)$ by using the rules in fig. 1.1. Here it is.

$$\frac{\dfrac{\mathsf{conn}(A, B)}{\mathsf{path}(A, B)}\ \textsc{Dir} \quad \mathsf{path}(B, C)}{\mathsf{path}(A, C)}\ \textsc{Trans}$$

The derivation above 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.

1.4.2 Rule admissibility

Let us look at the other direction: proving that if adding rule Trans to the system with rule Step would not add any new derivable propositions.

Recall that rule Trans is defined as follows.

$$\frac{\mathsf{path}(A,B) \quad \mathsf{path}(B,C)}{\mathsf{path}(A,C)}\ \textsc{Trans}$$

As a first attempt, we could try to adopt the same strategy that we followed in section 1.4.1: deriving the conclusion $\mathsf{path}(A,C)$ from the premises $\mathsf{path}(A,B)$ and $\mathsf{path}(B,C)$ 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 for rule Trans). It is sometimes convenient to mark admissible rules explicitly when defining them. We will use dashed horizontal lines to denote admissible inference rules.

Theorem 1. *The rule*

$$\frac{\mathsf{path}(A,B) \quad \mathsf{path}(B,C)}{\mathsf{path}(A,C)}\ \textsc{Trans}$$

is admissible in the system in fig. 1.4.

Proof. Let $\mathcal{P}$ and $\mathcal{Q}$ be the derivations of $\mathsf{path}(A,B)$ and $\mathsf{path}(B,C)$, respectively. We proceed by induction on the size of the derivation

$$\frac{\dfrac{\mathcal{P}}{\mathsf{path}(A,B)} \quad \dfrac{\mathcal{Q}}{\mathsf{path}(B,C)}}{\mathsf{path}(A,C)}\ \textsc{Trans}$$

and prove that we can build a derivation using the rules in fig. 1.4 *that is not bigger* (this is going to be important for the last case of the proof). We have three cases, depending on the last applied rule in $\mathcal{P}$ (one for each rule that can conclude a path proposition).

Case Dir $\mathcal{P}$ ends with an application of rule Dir. Thus, for some $\mathcal{P}'$ that does not contain any applications of rule Trans, we are in the following situation.

$$\begin{prooftree}
\AxiomC{$\mathcal{P}'$}
\UnaryInfC{$\mathsf{conn}(A,B)$}
\RightLabel{\textsc{Dir}}
\UnaryInfC{$\mathsf{path}(A,B)$}
\AxiomC{$\mathcal{Q}$}
\UnaryInfC{$\mathsf{path}(B,C)$}
\RightLabel{\textsc{Trans}}
\dottedLine
\BinaryInfC{$\mathsf{path}(A,C)$}
\end{prooftree}$$

We rewrite the proof as follows.

$$\begin{prooftree}
\AxiomC{$\mathcal{P}'$}
\UnaryInfC{$\mathsf{conn}(A,B)$}
\AxiomC{$\mathcal{Q}$}
\UnaryInfC{$\mathsf{path}(B,C)$}
\RightLabel{\textsc{Step}}
\BinaryInfC{$\mathsf{path}(A,C)$}
\end{prooftree}$$

If $\mathcal{Q}$ does not contain applications of rule Trans, then this is the base case and the thesis follows immediately. Otherwise, the thesis follows by induction hypothesis.

Case Step $\mathcal{P}$ ends with an application of rule Step. Thus, for some $\mathcal{P}'$ and $\mathcal{P}''$, we are in the following situation.

$$\begin{prooftree}
\AxiomC{$\mathcal{P}'$}
\UnaryInfC{$\mathsf{conn}(A,B')$}
\AxiomC{$\mathcal{P}''$}
\UnaryInfC{$\mathsf{path}(B',B)$}
\RightLabel{\textsc{Step}}
\BinaryInfC{$\mathsf{path}(A,B')$}
\AxiomC{$\mathcal{Q}$}
\UnaryInfC{$\mathsf{path}(B,C)$}
\RightLabel{\textsc{Trans}}
\dottedLine
\BinaryInfC{$\mathsf{path}(A,C)$}
\end{prooftree}$$

We rewrite the proof as follows.

$$\begin{prooftree}
\AxiomC{$\mathcal{P}'$}
\UnaryInfC{$\mathsf{conn}(A,B')$}
\AxiomC{$\mathcal{P}''$}
\UnaryInfC{$\mathsf{path}(B',B)$}
\AxiomC{$\mathcal{Q}$}
\UnaryInfC{$\mathsf{path}(B,C)$}
\RightLabel{\textsc{Trans}}
\dottedLine
\BinaryInfC{$\mathsf{path}(B',C)$}
\RightLabel{\textsc{Step}}
\BinaryInfC{$\mathsf{path}(A,C)$}
\end{prooftree}$$

The thesis now follows by induction hypothesis.

Case Trans $\mathcal{P}$ ends with an application of rule Trans. Thus, for some $\mathcal{P}'$ and $\mathcal{P}''$, we are in the following situation.

$$\begin{prooftree}
\AxiomC{$\mathcal{P}'$}
\UnaryInfC{$\mathsf{path}(A,B')$}
\AxiomC{$\mathcal{P}''$}
\UnaryInfC{$\mathsf{path}(B',B)$}
\RightLabel{\textsc{Trans}}
\dottedLine
\BinaryInfC{$\mathsf{path}(A,B)$}
\AxiomC{$\mathcal{Q}$}
\UnaryInfC{$\mathsf{path}(B,C)$}
\RightLabel{\textsc{Trans}}
\dottedLine
\BinaryInfC{$\mathsf{path}(A,C)$}
\end{prooftree}$$

From the proof of the left-hand premise and the induction hypothesis, we know that there exists a proof $\mathcal{R}$ that does not contain applications of rule Trans, ends with $\mathsf{path}(A, B)$, and is not bigger than the left-hand side proof

$$\dfrac{\dfrac{\mathcal{P}'}{\mathsf{path}(A, B')} \qquad \dfrac{\mathcal{P}''}{\mathsf{path}(B', B)}}{\mathsf{path}(A, B)}\ \textsc{Trans}.$$

Thus we can build the following smaller derivation, by eliminating at least one application of rule Trans.

$$\dfrac{\dfrac{\mathcal{R}}{\mathsf{path}(A, B)} \qquad \dfrac{\mathcal{Q}}{\mathsf{path}(B, C)}}{\mathsf{path}(A, C)}\ \textsc{Trans}$$

Since the derivation is smaller than the one we started with, we can conclude by invoking the induction hypothesis on this derivation.

□

Exercise 3 (!). *Prove that if* $\mathsf{path}(A, B)$ *is provable in the system in fig. 1.4, then there exists a nonempty sequence of provable propositions* $\mathsf{conn}(C_1, C_1')$, $\dots$, $\mathsf{conn}(C_n, C_n')$ *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 $\mathsf{path}(A, B)$ *is provable in the system in fig. 1.1, then there exists a nonempty sequence of provable propositions* $\mathsf{conn}(C_1, C_1')$, $\dots$, $\mathsf{conn}(C_n, C_n')$ *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*).*

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.

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

$$C ::= \mathsf{p} \texttt{ -> } \mathsf{q}; C \mid \mathbf{0}$$

Figure 2.1: Simple choreographies, syntax.

2.1 Syntax

Our first choreography model is called simple choreographies. The syntax of simple choreographies is given by the grammar in fig. 2.1, where C ranges over a choreography. Choreographies describe interactions between processes. We refer to processes by using process names, ranged over by p, q.

The syntax of simple choreographies is minimalistic. A choreography can either be a term $\mathsf{p} \texttt{ -> } \mathsf{q}; C$ or term $\mathbf{0}$. Term $\mathsf{p} \texttt{ -> } \mathsf{q}; C$ is an interaction and reads "process p sends a message to process q; then, the choreography proceeds as C". We always assume that p and q are different in interactions $\mathsf{p} \texttt{ -> } \mathsf{q}$, i.e., $\mathsf{p} \neq \mathsf{q}$ (meaning that a process cannot send a message to itself). Term $\mathbf{0}$ is the terminated choreography (no interactions, or end of program, if you like). Sometimes, we refer to terms as *programs* in the remainder.

Example 2. *The following choreography defines the behaviour that we informally described in example 1.*

$$\mathsf{Buyer} \texttt{ -> } \mathsf{Seller}; \mathsf{Seller} \texttt{ -> } \mathsf{Buyer}; \mathbf{0}$$

Note that what we have is actually a rather coarse abstraction *of what we described in example 1, 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 semantic interpretation. We use a relation to do that. Recall from set theory that a relation $\mathcal{R}$ on two sets S and S' is a subset of the product $S \times S'$ (the set of all pairs of elements from S and S'). Given an element $s \in S$ and an element $s' \in S'$, we write $s\mathcal{R}s'$ for $(s, s') \in \mathcal{R}$.

$$\frac{}{\mathsf{p} \texttt{->} \mathsf{q}; C \to C}\ \textsc{Com}$$

Figure 2.2: Simple choreographies, semantics.

Formally, the semantics for simple choreographies is given in terms of a *reduction relation* $\to$, which is defined as the smallest relation satisfying the rule displayed in fig. 2.2. Whenever two choreographies C and C' are related by $\to$, written $C \to C'$, we say that there is a reduction from C to C'. A reduction models executing a step of a choreography.

There is only one rule, called Com, which is an axiom: it always allows us to reduce 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 $\mathsf{p} \neq \mathsf{q}$ in communication terms, so this is true also here.)

The sets over which the relation $\to$ is defined are given implicitly: they are evident from the inference rules that define the relation. Specifically, let *SimpleChor* be the set of all choreography terms in our grammar for simpler choreographies—or, equivalently, the *language* generated by that grammar. There is only one inference rule in fig. 2.2, rule Com. The rule relates any choreography of the form $\mathsf{p} \texttt{->} \mathsf{q}; C$ to the choreography C. Thus, we know that $\to \subseteq \mathit{SimpleChor} \times \mathit{SimpleChor}$.

It is important that the set of departure of relation $\to$ (*SimpleChor*) is the same as its set of destination (*SimpleChor*), i.e., that $\to$ goes from choreographies to choreographies. This property allows us to define the useful concept of strong reduction chain, which permits to observe multiple steps of execution. The "strong" qualifier indicates that there is at least one step in the chain.

Definition 1 (Strong reduction chain)**.** *We say that there is a strong reduction chain from C_1 to C_n whenever there exists a sequence of choreographies $(C_1, \ldots, C_n)$ such that $C_i \to C_{i+1}$ for each $i \in [1, n-1]$.*

When there is a strong reduction chain from C_1 to C_n, we write $C_1 \to^+ C_n$ (when showing the intermediate steps is not necessary, only their existance) or $C_1 \to \cdots \to C_n$ (when we want to show the intermediate steps).

Example 3. *The program in example 2 has the following strong reduction chain:*

$$\mathsf{Buyer} \texttt{->} \mathsf{Seller}; \mathsf{Seller} \texttt{->} \mathsf{Buyer}; \mathbf{0} \quad \to \quad \mathsf{Seller} \texttt{->} \mathsf{Buyer}; \mathbf{0} \quad \to \quad \mathbf{0}.$$

So we first reduce an interaction where Buyer *sends a message to* Seller *and then we reduce an interaction where* Seller *sends a message to* Buyer*, which is exactly the communication flow that we wanted in example 1.*

Another way to define relation $\rightarrow^+$ is: $\rightarrow^+$ is the transitive closure of $\rightarrow$. In the next sections, we will also use the reflexive and transitive closure of $\rightarrow$, written $\rightarrow^*$ and defined as follows. Whenever $C \rightarrow^* C'$ for some C and C', we say that there is a weak reduction chain from C to C'.

Definition 2 (Weak reduction chain)**.** *We write* $C \rightarrow^* C'$ *if either:*

Base case: $C = C'$*; or,*

Inductive case: *there exists* C'' *such that* $C \rightarrow C''$ *and* $C'' \rightarrow^* C'$*.*

If you are familiar with regular expressions, the use of the symbols $+$ and $*$ should ring a bell: $\rightarrow^+$ means "one or more reductions" and $\rightarrow^*$ means "zero or more reductions".

Exercise 4. *Prove that* $C \rightarrow^+ C'$ *implies* $C \rightarrow^* C'$*. Prove that the converse does not hold.*

Exercise 5. *Prove the following statements.*

1. *For all* $C \neq \mathbf{0}$ *(all* C *that are not* $\mathbf{0}$*), there exists* C' *such that* $C \rightarrow C'$*.*
2. *For all* $C \neq \mathbf{0}$*, there exists* C' *such that* $C \rightarrow^* C'$*.*
3. *For all* $C \neq \mathbf{0}$*,* $C \rightarrow^+ \mathbf{0}$*.*
4. *For all* C*,* $C \rightarrow^* \mathbf{0}$*.*

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 with each other, 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 4. *To implement our scenario from example 1 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*;*

$$
\begin{aligned}
N &::= \mathsf{p} \triangleright B \mid N \,|\, N \mid \mathbf{0} \\
B &::= \mathsf{p}!; B \mid \mathsf{p}?; B \mid \mathbf{0}
\end{aligned}
$$

Figure 3.1: Simple processes, syntax.

- *receive the price of the book from* Seller*.*

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

- *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. We model systems as networks, ranged over by N. A network N can be: a single process term $\mathsf{p} \triangleright B$, where p is the name of the process and B its behaviour; a parallel composition of two networks N and N', written $N \mid N'$, which enables the processes in N and N' to communicate; or the empty network $\mathbf{0}$. A process behaviour, ranged over by B, can be: a send action $\mathsf{p}!; B$, read "send a message to process p and then do B"; a receive action $\mathsf{p}?; B$, read "receive a message from process p and then do B"; or the terminated behaviour $\mathbf{0}$.

Example 5. *The two process programs informally described in example 4 can be formalised as follows.*

$$\mathsf{Buyer} \triangleright \mathsf{Seller}!; \mathsf{Seller}?; \mathbf{0} \ \mid\ \mathsf{Seller} \triangleright \mathsf{Buyer}?; \mathsf{Buyer}!; \mathbf{0}$$

3.1.2 Semantics: discussion

Similarly to what we have done for simple choreographies, we can equip simple processes with a semantics based on reductions of the form $N \to N'$. The idea is that each reduction should model a step in the execution of a network. However, this requires a few extra ingredients compared to the semantics of simple choreographies, so we make a few considerations before giving the general definition.

The essential rule defining reductions for simple processes is the following axiom, rule Com.

$$\frac{}{\mathsf{p} \triangleright \mathsf{q}!; B \ \mid\ \mathsf{q} \triangleright \mathsf{p}?; B' \quad \to \quad \mathsf{p} \triangleright B \ \mid\ \mathsf{q} \triangleright B'}\ \textsc{Com}$$

Rule Com models communications, by matching a send action by process p towards process q with a receive action at q waiting for a message from p. Each process then proceeds with its respective continuation (B for p, B' for q).

Rule Com is simple, because it focuses only on the components that it needs (the sender and the receiver). However, if we designed a semantics that includes only that rule, the result would be too limited. For example, consider the network from example 5:

$$\mathsf{Buyer} \triangleright \mathsf{Seller}!; \mathsf{Seller}?; \mathbf{0} \ \mid\ \mathsf{Seller} \triangleright \mathsf{Buyer}?; \mathsf{Buyer}!; \mathbf{0}.$$

By rule Com, we have the following reduction:

$$\begin{array}{l} \mathsf{Buyer} \triangleright \mathsf{Seller}!; \mathsf{Seller}?; \mathbf{0} \ \mid\ \mathsf{Seller} \triangleright \mathsf{Buyer}?; \mathsf{Buyer}!; \mathbf{0} \\ \to \\ \mathsf{Buyer} \triangleright \mathsf{Seller}?; \mathbf{0} \ \mid\ \mathsf{Seller} \triangleright \mathsf{Buyer}!; \mathbf{0}. \end{array}$$

So far so good. But what about the reductum (the term of the reduction)? Intuitively, we would expect to be able to reduce the reductum

$$\mathsf{Buyer} \triangleright \mathsf{Seller}?; \mathbf{0} \ \mid\ \mathsf{Seller} \triangleright \mathsf{Buyer}!; \mathbf{0} \tag{3.1}$$

as follows, because Buyer and Seller are performing compatible actions—Buyer wants to receive from Seller, and Seller wants to send to Buyer.

$$\mathsf{Buyer} \triangleright \mathsf{Seller}?; \mathbf{0} \ \mid\ \mathsf{Seller} \triangleright \mathsf{Buyer}!; \mathbf{0} \quad \to \quad \mathsf{Buyer} \triangleright \mathbf{0} \ \mid\ \mathsf{Seller} \triangleright \mathbf{0}$$

Sadly, this does not work, because rule Com requires the sender to appear on the left-hand side of the parallel composition.

One solution could be to have another rule, Moc (the reverse of Com), that allows us to reduce networks where the sender appears on the right-hand side.

$$\frac{}{\mathsf{p} \triangleright \mathsf{q}?; B \ \mid\ \mathsf{q} \triangleright \mathsf{p}!; B' \quad \to \quad \mathsf{p} \triangleright B \ \mid\ \mathsf{q} \triangleright B'}\ \textsc{Moc}$$

While Moc might look like a good idea at first, we quickly run into trouble again with other networks. What if we have a terminated process Bystander in between Buyer and Seller in our example?

$$\mathsf{Buyer} \triangleright \mathsf{Seller}?; \mathbf{0} \ \mid\ \mathsf{Bystander} \triangleright \mathbf{0} \ \mid\ \mathsf{Seller} \triangleright \mathsf{Buyer}!; \mathbf{0} \tag{3.2}$$

We cannot apply rule Moc to the network above, because the rule does not allow for any terms to appear in between the two interacting processes.

Instead of designing an ad-hoc rule for each corner case, we adopt the standard technique of defining a *structural relation* [Sangiorgi and Walker, 2001]. A structural relation relates terms that are intuitively equivalent and should behave in the same way, but are not syntactically equal. We will denote our structural relation for processes with $\preceq$.

To gain some intuition on what terms $\preceq$ should relate, consider again the network with the bystander in eq. (3.2). Since process Bystander is not performing any actions at all, we should be able to disregard it entirely when reasoning about our network. In other words, we wish for the following to hold.

$$\begin{array}{l} \mathsf{Buyer} \triangleright \mathsf{Seller}?; \mathbf{0} \\ \mid\ \mathsf{Bystander} \triangleright \mathbf{0} \\ \mid\ \mathsf{Seller} \triangleright \mathsf{Buyer}!; \mathbf{0} \end{array} \quad \preceq \quad \begin{array}{l} \mathsf{Buyer} \triangleright \mathsf{Seller}?; \mathbf{0} \\ \mid\ \mathsf{Seller} \triangleright \mathsf{Buyer}!; \mathbf{0} \end{array}$$

Observe that, in the right-hand side, we ended up in the same situation as in eq. (3.1): we cannot apply rule Com because the sender and the receiver do not appear in the right order. However, it does not make sense that order matters in parallel compositions. If two processes wish to speak to each other, then they should just be able to do so. Why should the order in which they are "placed in the network" matter? Formally, we wish for the following.

$$\mathsf{Buyer} \triangleright \mathsf{Seller}?;\mathbf{0} \;\mid\; \mathsf{Seller} \triangleright \mathsf{Buyer}!;\mathbf{0} \;\preceq\; \mathsf{Seller} \triangleright \mathsf{Buyer}!;\mathbf{0} \;\mid\; \mathsf{Buyer} \triangleright \mathsf{Seller}?;\mathbf{0}$$

Another wish is that $\preceq$ is not finer than syntactic equality, i.e., if two terms are exactly the same, then they are also structurally related. Formally, for any two networks N and N', $N = N'$ implies that $N \preceq N'$.

If we had a relation $\preceq$ that grants our wishes (and we are going to define it), then we could use it in our reduction semantics by adopting the following rule.

$$\frac{N \preceq N_1 \qquad N_1 \to N_2 \qquad N_2 \preceq N'}{N \to N'}\ \textsc{Struct}$$

Rule Struct closes $\to$ under $\preceq$. With this rule, we can finally derive that our network with the bystander from eq. (3.2) can reduce and terminate.

$$\frac{\begin{array}{l}\mathsf{Buyer} \triangleright \mathsf{Seller}?;\mathbf{0}\\ \mid\ \mathsf{Bystander} \triangleright \mathbf{0}\\ \mid\ \mathsf{Seller} \triangleright \mathsf{Buyer}!;\mathbf{0}\end{array} \preceq \begin{array}{l}\mathsf{Seller} \triangleright \mathsf{Buyer}!;\mathbf{0}\\ \mid\ \mathsf{Buyer} \triangleright \mathsf{Seller}?;\mathbf{0}\end{array} \qquad \begin{array}{l}\mathsf{Seller} \triangleright \mathsf{Buyer}!;\mathbf{0}\\ \mid\ \mathsf{Buyer} \triangleright \mathsf{Seller}?;\mathbf{0}\end{array} \to \begin{array}{l}\mathsf{Seller} \triangleright \mathbf{0}\\ \mid\ \mathsf{Buyer} \triangleright \mathbf{0}\end{array} \qquad \begin{array}{l}\mathsf{Seller} \triangleright \mathbf{0}\\ \mid\ \mathsf{Buyer} \triangleright \mathbf{0}\end{array} \to \begin{array}{l}\mathsf{Buyer} \triangleright \mathbf{0}\\ \mid\ \mathsf{Seller} \triangleright \mathbf{0}\end{array}}{\begin{array}{l}\mathsf{Buyer} \triangleright \mathsf{Seller}?;\mathbf{0}\\ \mid\ \mathsf{Bystander} \triangleright \mathbf{0}\\ \mid\ \mathsf{Seller} \triangleright \mathsf{Buyer}!;\mathbf{0}\end{array} \to \begin{array}{l}\mathsf{Buyer} \triangleright \mathbf{0}\\ \mid\ \mathsf{Seller} \triangleright \mathbf{0}\end{array}}\ \textsc{Struct}$$

3.1.3 Semantics: definition

We move to the formal definition of the semantics of simple processes. Since the definition of the reduction relation $\to$ depends on the definition of the structural relation $\preceq$, we define the latter first.

Structural relation

First, a useful abbreviation: we will write $N \equiv N'$ whenever $N \preceq N'$ and $N' \preceq N$ (the symmetric cases).

We define the structural relation $\preceq$ as the smallest relation that satifies the axioms displayed in fig. 3.2.

Rules PC and PA capture that parallel composition is commutative (PC) and associative (PA), so that we can disregard the order in which networks are composed. Rule GCN garbage collects a terminated network, and rule GCP transforms a terminated process into a terminated network (which can be later collected by rule GCN).

$$\frac{}{N_1 \mid N_2 \;\equiv\; N_2 \mid N_1}\ \text{PC} \qquad \frac{}{(N_1 \mid N_2) \mid N_3 \;\equiv\; N_1 \mid (N_2 \mid N_3)}\ \text{PA}$$

$$\frac{}{N \mid \mathbf{0} \;\preceq\; N}\ \text{GCN} \qquad \frac{}{\mathsf{p} \triangleright \mathbf{0} \;\preceq\; \mathbf{0}}\ \text{GCP}$$

$$\frac{}{N \preceq N}\ \textsc{Refl} \qquad \frac{N \preceq N' \quad N' \preceq N''}{N \preceq N''}\ \textsc{Trans} \qquad \frac{N \preceq N'}{\mathscr{C}[N] \preceq \mathscr{C}[N']}\ \textsc{Ctx}$$

Figure 3.2: Simple processes, structural relation.

Rule Refl makes the structural relation reflexive, which makes it include syntactic equality (a network is always structurally related to itself).

Rule Trans makes the structural relation transitive, so that we can "use it" multiple times, e.g., to permute the processes in a parallel composition until we reach the configuration that we desire.

Rule Ctx closes the structural relation up to any context $\mathscr{C}$, which we formally define now.

Definition 3 (Context). *A context, denoted $\mathscr{C}$, is obtained when the hole $\bullet$ (a special terminal symbol) replaces one occurrence of $\mathbf{0}$ in a network.*

Given a context $\mathscr{C}$ and a network N, then $\mathscr{C}[N]$ is the network obtained by replacing the hole $\bullet$ in $\mathscr{C}$ with N.

Example 6. *Let the context $\mathscr{C}$ be defined as follows. We put explicit parentheses for grammatical clarity.*

$$\mathscr{C} \triangleq (\mathsf{Buyer} \triangleright \mathsf{Seller}?; \mathbf{0} \;\mid\; \bullet) \;\mid\; \mathsf{Seller} \triangleright \mathsf{Buyer}!; \mathbf{0}$$

Then, the network with the bystander in eq. (3.2) *can be obtained by replacing the hole in the context $\mathscr{C}$ with the bystander process:*

$$\mathscr{C}[\mathsf{Bystander} \triangleright \mathbf{0}] = (\mathsf{Buyer} \triangleright \mathsf{Seller}?; \mathbf{0} \;\mid\; \mathsf{Bystander} \triangleright \mathbf{0}) \;\mid\; \mathsf{Seller} \triangleright \mathsf{Buyer}!; \mathbf{0}.$$

Example 7. *Thanks to contexts, rule* Ctx *allows us to apply the structural relation to sub-terms of networks.*

Let $\mathscr{C}$ be defined as in example 6 and $\mathscr{D}$ be defined as follows.

$$\mathscr{D} \triangleq \bullet \;\mid\; \mathsf{Seller} \triangleright \mathsf{Buyer}!; \mathbf{0}$$

$$\frac{}{\mathsf{p} \triangleright \mathsf{q}!; B \ \mid\ \mathsf{q} \triangleright \mathsf{p}?; B' \quad \to \quad \mathsf{p} \triangleright B \ \mid\ \mathsf{q} \triangleright B'}\ \textsc{Com}$$

$$\frac{N_1 \to N_1'}{N_1 \mid N_2 \ \to\ N_1' \mid N_2}\ \textsc{Par} \qquad \frac{N \preceq N_1 \quad N_1 \to N_2 \quad N_2 \preceq N'}{N \to N'}\ \textsc{Struct}$$

Figure 3.3: Simple processes, semantics.

Notice that $\mathscr{C}[\mathbf{0}] = \mathscr{D}[\mathsf{Buyer} \triangleright \mathsf{Seller}?; \mathbf{0} \mid \mathbf{0}]$. *Then, we can derive:*

$$\frac{\dfrac{\dfrac{}{\mathsf{Bystander} \triangleright \mathbf{0} \preceq \mathbf{0}}\ \text{GCP}}{\mathscr{C}[\mathsf{Bystander} \triangleright \mathbf{0}] \preceq \mathscr{C}[\mathbf{0}]}\ \textsc{Ctx} \qquad \dfrac{\dfrac{}{\mathsf{Buyer} \triangleright \mathsf{Seller}?; \mathbf{0} \mid \mathbf{0} \preceq \mathsf{Buyer} \triangleright \mathsf{Seller}?; \mathbf{0}}\ \text{GCN}}{\mathscr{D}[\mathsf{Buyer} \triangleright \mathsf{Seller}?; \mathbf{0} \mid \mathbf{0}] \preceq \mathscr{D}[\mathsf{Buyer} \triangleright \mathsf{Seller}?; \mathbf{0}]}\ \textsc{Ctx}}{\begin{array}{l}(\mathsf{Buyer} \triangleright \mathsf{Seller}?; \mathbf{0} \ \mid\ \mathsf{Bystander} \triangleright \mathbf{0}) \\ \mid\ \mathsf{Seller} \triangleright \mathsf{Buyer}!; \mathbf{0}\end{array} \preceq \mathsf{Buyer} \triangleright \mathsf{Seller}?; \mathbf{0} \ \mid\ \mathsf{Seller} \triangleright \mathsf{Buyer}!; \mathbf{0}}\ \textsc{Trans} \quad .$$

In the remainder, we will not show explicit derivations for the structural relation. It should be evident, for example, that from the conclusion that we have just derived, we can obtain the following by applying rules Trans *(transitivity) and* PC *(commutativity of parallel).*

$$\begin{array}{l}(\mathsf{Buyer} \triangleright \mathsf{Seller}?; \mathbf{0} \ \mid\ \mathsf{Bystander} \triangleright \mathbf{0}) \\ \mid\ \mathsf{Seller} \triangleright \mathsf{Buyer}!; \mathbf{0}\end{array} \preceq \ \mathsf{Seller} \triangleright \mathsf{Buyer}!; \mathbf{0} \ \mid\ \mathsf{Buyer} \triangleright \mathsf{Seller}?; \mathbf{0}$$

Reductions

The semantics of simple processes is given by the reduction rules displayed in fig. 3.3.

Rule Com models communications, by matching a send action by process p towards process q with a receive action at q waiting for a message from p. Each process then proceeds with its respective continuation (B for p, B' for q).

Rule Par allows for reductions to happen inside of a sub-network.

Rule Struct allows us to apply the structural relation $\preceq$ when deriving reductions.

Rule Par allows for reductions to happen in a sub-network.

Rule Struct closes reductions under the structural relation $\preceq$, allowing us to disregard structural differences in networks.

Exercise 6. *Prove that the following reductions hold.*

$$\begin{array}{l}\mathsf{Buyer} \triangleright \mathsf{Seller}!; \mathsf{Seller}?; \mathbf{0} \;\mid\; \mathsf{Seller} \triangleright \mathsf{Buyer}?; \mathsf{Buyer}!; \mathbf{0} \\ \to \\ \mathsf{Buyer} \triangleright \mathsf{Seller}?; \mathbf{0} \;\mid\; \mathsf{Seller} \triangleright \mathsf{Buyer}!; \mathbf{0}\end{array}$$

$$\mathsf{Buyer} \triangleright \mathsf{Seller}?; \mathbf{0} \;\mid\; \mathsf{Seller} \triangleright \mathsf{Buyer}!; \mathbf{0} \;\to\; \mathbf{0}$$

$$(\mathsf{Buyer} \triangleright \mathsf{Seller}?; \mathbf{0} \;\mid\; \mathsf{Bystander} \triangleright \mathbf{0}) \;\mid\; \mathsf{Seller} \triangleright \mathsf{Buyer}!; \mathbf{0} \;\to\; \mathbf{0}$$

Exercise 7 (!)**.** *Prove the following statement.*

If $N_1 \preceq N_2$ *and* $N_1 \preceq N_3$*, then there exists* N' *such that* $N_2 \preceq N'$ *and* $N_3 \preceq N'$*.*

3.2 Endpoint Projection (EPP)

The network in example 5 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

To gain some intuition on how we could develop the method we want, we can look at our examples. Let us see our choreography from example 2 again:

$$\mathsf{Buyer} \texttt{->} \mathsf{Seller}; \mathsf{Seller} \texttt{->} \mathsf{Buyer}; \mathbf{0}.$$

It is evident, albeit informally, that our network from example 5 implements exactly the interactions defined in the choreography:

$$\mathsf{Buyer} \triangleright \mathsf{Seller}!; \mathsf{Seller}?; \mathbf{0} \;\mid\; \mathsf{Seller} \triangleright \mathsf{Buyer}?; \mathsf{Buyer}!; \mathbf{0}.$$

This informal correspondence is preserved by reductions—remember, reductions model execution, if you think in computational terms. Indeed, whenever we take a step in the reduction chain shown in example 3 (for the choreography), we can "mimic" it by following the reduction chain of example 5 (for the process network), and vice versa (if we take a step for the process network, we can mimic it for the choreography).

The intuition that we can gain from our examples is that a network "implements" a choreography if the actions performed by processes give rise to the interactions described in the choreography. Therefore, an automatic method that produces networks from choreographies should generate a network that consists of the processes described in the choreography, and the behaviour of each of these processes should be the actions that the process needs to perform to implement the interactions that it is involved in in the choreography.

We can now move to formally defining our desired method, 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 $\mathsf{Buyer} \mathrel{\text{->}} \mathsf{Seller}$ 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.

Let $\mathsf{procs}(C)$ be the set of process names used in C. We can define this function inductively on the structure of C, as follows.

$$
\begin{aligned}
\mathsf{procs}\,(\mathsf{p} \mathrel{\text{->}} \mathsf{q}; C) &= \{\mathsf{p}, \mathsf{q}\} \cup \mathsf{procs}(C) \\
\mathsf{procs}(\mathbf{0}) &= \emptyset
\end{aligned}
$$

We write $[\![C]\!]$ for the EPP of a choreography C.

Definition 4 (EndPoint Projection (EPP))**.** *The EPP of a choreography C, denoted $[\![C]\!]$, is defined as:*

$$
[\![C]\!] = \prod_{\mathsf{p} \in \mathsf{procs}(C)} \mathsf{p} \triangleright [\![C]\!]_{\mathsf{p}}\,.
$$

In definition 5, the notation $\prod_{\mathsf{p}\in\mathsf{procs}(C)} \mathsf{p} \triangleright [\![C]\!]_{\mathsf{p}}$ stands for "the parallel composition of all $\mathsf{p} \triangleright [\![C]\!]_{\mathsf{p}}$ such that p is in $\mathsf{procs}(C)$". The auxiliary function $[\![C]\!]_{\mathsf{p}}$—not to be confused with $[\![C]\!]$—projects the actions that process p should perform in order to implement its part in choreography C. We call $[\![C]\!]_{\mathsf{p}}$ a behaviour projection, since it outputs a behaviour B. It is inductively defined by the rules given in fig. 3.4.

Example 8. *Let $C_{\mathsf{BuyerSeller}}$ be the choreography in example 2. We recall it here ($\triangleq$ stands for "defined as"):*

$$
C_{\mathsf{BuyerSeller}} \quad \triangleq \quad \mathsf{Buyer} \mathrel{\text{->}} \mathsf{Seller}; \mathsf{Seller} \mathrel{\text{->}} \mathsf{Buyer}; \mathbf{0}.
$$

$$
\llbracket \mathsf{p} \mathrel{\texttt{->}} \mathsf{q}; C \rrbracket_{\mathsf{r}} = \begin{cases} \mathsf{q}!; \llbracket C \rrbracket_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{p} \\ \mathsf{p}?; \llbracket C \rrbracket_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{q} \\ \llbracket C \rrbracket_{\mathsf{r}} & \text{otherwise} \end{cases}
$$

$$
\llbracket \mathbf{0} \rrbracket_{\mathsf{p}} = \mathbf{0}
$$

Figure 3.4: Behaviour projection for simple choreographies.

The process names in $C_{\mathsf{BuyerSeller}}$ *are:*

$$
\mathsf{procs}(C_{\mathsf{BuyerSeller}}) = \{\mathsf{Buyer}, \mathsf{Seller}\}.
$$

The behaviour projection for Buyer *is:*

$$
\llbracket C_{\mathsf{BuyerSeller}} \rrbracket_{\mathsf{Buyer}} = \mathsf{Seller}!; \mathsf{Seller}?; \mathbf{0}.
$$

The EPP of $C_{\mathsf{BuyerSeller}}$ *is exactly the network that we defined in example 5:*

$$
\llbracket C_{\mathsf{BuyerSeller}} \rrbracket = \mathsf{Buyer} \triangleright \mathsf{Seller}!; \mathsf{Seller}?; \mathbf{0} \mid \mathsf{Seller} \triangleright \mathsf{Buyer}?; \mathsf{Buyer}!; \mathbf{0}.
$$

Exercise 8. *Write the outputs of* procs *and* $\llbracket\rrbracket$ *(EPP) for the choreography*

$$
\mathsf{p} \mathrel{\texttt{->}} \mathsf{q}; \mathsf{r} \mathrel{\texttt{->}} \mathsf{q}; \mathsf{r} \mathrel{\texttt{->}} \mathsf{s}; \mathsf{q} \mathrel{\texttt{->}} \mathsf{p}; \mathbf{0}.
$$

Exercise 9. *What are the reduction chains for the choreography in exercise 8 and (the network resulting from) its EPP? Do you think that they informally correspond to one another? (We have not formally defined correspondence yet.)*

Exercise 10. *Is the following statement true?*
Let $N = \llbracket C \rrbracket$*. If* $N \to N'$ *for some* N'*, then there exists* C' *such that* $C \to C'$ *and* $N' \preceq \llbracket C' \rrbracket$*.*

3.2.2 Towards a correct EPP

Unfortunately, the statement in exercise 10 does not hold. More specifically, the framework of simple choreographies does not support a key desirable property: the EPP of a choreography should only do what the original choreography prescribes.

The counterexample is simple. Take the following choreography:

$$
C_{\mathsf{problem}} \triangleq \mathsf{p} \mathrel{\texttt{->}} \mathsf{q}; \mathsf{r} \mathrel{\texttt{->}} \mathsf{s}; \mathbf{0}. \tag{3.3}
$$

The EPP of this choreography is:

$$[\![C_{\mathsf{problem}}]\!] \;=\; \mathsf{p} \triangleright \mathsf{q}!; \mathbf{0} \,|\, \mathsf{q} \triangleright \mathsf{p}?; \mathbf{0} \,|\, \mathsf{r} \triangleright \mathsf{s}!; \mathbf{0} \,|\, \mathsf{s} \triangleright \mathsf{r}?; \mathbf{0}.$$

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

$$\mathsf{p} \triangleright \mathsf{q}!; \mathbf{0} \,|\, \mathsf{q} \triangleright \mathsf{p}?; \mathbf{0} \,|\, \mathsf{r} \triangleright \mathsf{s}!; \mathbf{0} \,|\, \mathsf{s} \triangleright \mathsf{r}?; \mathbf{0} \quad \to \quad \mathsf{p} \triangleright \mathsf{q}!; \mathbf{0} \,|\, \mathsf{q} \triangleright \mathsf{p}?; \mathbf{0}.$$

However, this reduction cannot be mimicked by C_{problem}, which can only reduce the first interaction between p and q according to the semantics given in the previous lecture notes.

The example above shows that our framework is not sound yet, because the projection of a choreography can perform "extra" reductions with respect to the choreography. There are two ways to fix this.

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

Exercise 11. *Check that the EPP of* $\mathsf{p} \texttt{ -> } \mathsf{q}; \mathsf{r} \texttt{ -> } \mathsf{q}; \mathbf{0}$ *reduces as expected by the choreography (i.e., their respective reduction chains match).*

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 following reduction

$$\mathsf{p} \texttt{ -> } \mathsf{q}; \mathsf{r} \texttt{ -> } \mathsf{s}; \mathbf{0} \quad \to \quad \mathsf{p} \texttt{ -> } \mathsf{q}; \mathbf{0}$$

$$C ::= \mathsf{p} \rightarrow \mathsf{q}; C \mid \mathbf{0}$$

Figure 3.5: Simple concurrent choreographies, syntax.

$$\frac{}{\mathsf{p} \rightarrow \mathsf{q}; C \to C}\ \textsc{Com} \qquad \frac{C \preceq C_1 \quad C_1 \to C_2 \quad C_2 \preceq C'}{C \to C'}\ \textsc{Struct}$$

Figure 3.6: Simple concurrent choreographies, semantics.

then we would be fine, because that is the reduction that we need to match the problematic one that the EPP of the choreography can do. Observe that this reduction 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 ininfluential 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]. In the next section, where we develop the technical details, we borrow the formulation by Cruz-Filipe and Montesi [2016].

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. 3.5.

The semantics of choreographies, instead, needs some updating to capture out-of-order execution of interactions. We obtain this by adding a structural relation $\preceq$ for choreographies[1]. The reduction rules are given in fig. 3.6. The only change is the addition of rule Struct, which closes reductions under $\preceq$.

---

[1] We use the same symbol as for the structural relation for networks, since we can easily distinguish them from the context (they operate on different domains, one choreographies and the other networks).

$$\dfrac{\{\mathsf{p},\mathsf{q}\}\,\#\{\mathsf{r},\mathsf{s}\}}{\mathsf{p}\ \mathsf{\text{->}}\ \mathsf{q};\mathsf{r}\ \mathsf{\text{->}}\ \mathsf{s} \quad\equiv\quad \mathsf{r}\ \mathsf{\text{->}}\ \mathsf{s};\mathsf{p}\ \mathsf{\text{->}}\ \mathsf{q}}\ \textsc{Swap}$$

Figure 3.7: Simple concurrent choreographies, structural precongruence.

We define $\preceq$ as the smallest relation that satisfies the rule displayed in fig. 3.7 and is reflexive, transitive, and closed under (choreographic) contexts. We do not give explicit inference rules for the last three properties (reflexive, transitive, context closure), as these should be intuitive by now. Recall that $C \equiv C'$ stands for $C \preceq C'$ and $C' \preceq C$. (Later on, we will see rules that use $\preceq$ in only one direction also for choreographies.) Rule Swap states that two interactions $\mathsf{p}\ \mathsf{\text{->}}\ \mathsf{q}$ and $\mathsf{r}\ \mathsf{\text{->}}\ \mathsf{s}$ can be exchanged in a choreography if the processes p, q, r, and s are distinct (they are all different). This is captured by the premise $\{\mathsf{p},\mathsf{q}\}\,\#\{\mathsf{r},\mathsf{s}\}$, which states that the sets $\{\mathsf{p},\mathsf{q}\}$ and $\{\mathsf{r},\mathsf{s}\}$ must be disjoint. Formally, given two sets S and S', $S \,\#\, S'$ is a shortcut notation for $S \cap S' = \emptyset$ (empty intersection). Notice that we slightly abuse notation in rule Swap: a term $\mathsf{p}\ \mathsf{\text{->}}\ \mathsf{q};\mathsf{r}\ \mathsf{\text{->}}\ \mathsf{s}$ is only "partial", in the sense that it is not a valid complete term for the grammar of choreographies, it can only be part of a bigger complete term. Context closure of $\preceq$ means that we can apply that rule to all subterms of a choreography.

Example 9. *This is the reduction we wished we could do in section 3.2.2.*

$$\mathsf{p}\ \mathsf{\text{->}}\ \mathsf{q};\mathsf{r}\ \mathsf{\text{->}}\ \mathsf{s};\mathbf{0} \quad\rightarrow\quad \mathsf{p}\ \mathsf{\text{->}}\ \mathsf{q};\mathbf{0}.$$

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

$$\dfrac{\begin{array}{c}\mathsf{p}\ \mathsf{\text{->}}\ \mathsf{q};\\ \mathsf{r}\ \mathsf{\text{->}}\ \mathsf{s};\mathbf{0}\end{array} \preceq \begin{array}{c}\mathsf{r}\ \mathsf{\text{->}}\ \mathsf{s};\\ \mathsf{p}\ \mathsf{\text{->}}\ \mathsf{q};\mathbf{0}\end{array} \qquad \dfrac{}{\begin{array}{c}\mathsf{r}\ \mathsf{\text{->}}\ \mathsf{s};\\ \mathsf{p}\ \mathsf{\text{->}}\ \mathsf{q};\mathbf{0}\end{array} \rightarrow \mathsf{p}\ \mathsf{\text{->}}\ \mathsf{q};\mathbf{0}}\ \textsc{Com} \qquad \mathsf{p}\ \mathsf{\text{->}}\ \mathsf{q};\mathbf{0} \preceq \mathsf{p}\ \mathsf{\text{->}}\ \mathsf{q};\mathbf{0}}{\mathsf{p}\ \mathsf{\text{->}}\ \mathsf{q};\mathsf{r}\ \mathsf{\text{->}}\ \mathsf{s};\mathbf{0} \quad\rightarrow\quad \mathsf{p}\ \mathsf{\text{->}}\ \mathsf{q};\mathbf{0}}\ \textsc{Struct}.$$

Exercise 12. *Show all the possible reduction chains of the following choreography.*

$$\mathsf{p}\ \mathsf{\text{->}}\ \mathsf{q};\mathsf{r}\ \mathsf{\text{->}}\ \mathsf{s};\mathsf{q}\ \mathsf{\text{->}}\ \mathsf{r};\mathbf{0}$$

Exercise 13. *Prove the following statement.*
Let $[\![C]\!] = N$*. If* $C \preceq C'$ *for some* C'*, then* $N \preceq [\![C']\!]$*.*

We can now formally state that EPP is correct, in the sense that the behaviour implemented by the network projected from a choreography is exactly the one defined in the choreography.

Theorem 2 (Operational Correspondence). *Let* $[\![C]\!] = N$*. Then,*

Completeness *If* $C \to C'$ *for some* C'*, then there exists* N' *such that* $N \to N'$ *and* $N' \preceq [\![C']\!]$*.*

Soundness *If* $N \to N'$ *for some* N'*, then there exists* C' *such that* $C \to C'$ *and* $N' \preceq [\![C']\!]$*.*

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, finally!

The correctness of EPP gives us a powerful result for free: the EPP of a choreography never gets stuck (for example, there cannot be deadlocks). 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. To have a deadlock, one typically needs a language where send and receive actions are defined separately (hence the opportunity for mistakes).

Let us formalise this result. First, we observe that choreographies never get stuck.

Theorem 3 (Progress for choreographies). *Let* C *be a choreography. Then, either* $C = \mathbf{0}$ *(*C *has terminated) or there exists* C' *such that* $C \to C'$*.*

Proof. By cases on C. If $C = \mathbf{0}$, the thesis follows immediately. Otherwise, we can just apply rule COM. $\square$

We now combine theorem 3 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.

Corollary 1 (Progress for EPP). *Let* $[\![C]\!] = N$*. Then, either* $N = \mathbf{0}$ *(*N *has terminated) or there exists* N' *such that* $N \to N'$*.*

Proof. Direct consequence of theorems 2 and 3. $\square$

Chapter 4

Statefulness

4.1 Local Computation

Now that we know the basic soundness principles of choreographies and EPP, we can play with extending our model such that we can capture more interesting—and realistic—examples.

In this section, we equip processes with the ability to perform local computation. This will enable us to capture our initial scenario from example 1 more precisely, i.e., we want to define the *content* of the messages exchanged by Buyer and Seller.

4.1.1 Stateful Choreographies

Syntax We augment the syntax of choreographies with *local functions*—ranged over by f, g, ...—which can be used by processes to perform local computation. The new syntax is given in fig. 4.1.

The key idea is that now processes are equipped with their own local memories, which they can manipulate through computation. The new term $\mathsf{p}.f$ reads "process p stores the result of function f in its memory". The new communication term $\mathsf{p}.f \texttt{ -> } \mathsf{q}.g; C$ reads "process p sends the result of computing function f to q; q then computes function g according to the

$$
\begin{aligned}
C &::= \eta; C \mid \mathbf{0} \\
\eta &::= \mathsf{p}.f \texttt{ -> } \mathsf{q}.g \mid \mathsf{p}.f
\end{aligned}
$$

Figure 4.1: Stateful choreographies, syntax.

$$\frac{f(\sigma(\mathsf{p}))\downarrow v}{\langle \mathsf{p}.f; C, \sigma\rangle \to \langle C, \sigma[\mathsf{p} \mapsto v]\rangle}\ \textsc{Local}$$

$$\frac{f(\sigma(\mathsf{p}))\downarrow v \qquad g(\sigma(\mathsf{q}), v)\downarrow u}{\langle \mathsf{p}.f \;\texttt{->}\; \mathsf{q}.g; C, \sigma\rangle \to \langle C, \sigma[\mathsf{q} \mapsto u]\rangle}\ \textsc{Com}$$

$$\frac{C \preceq C_1 \qquad \langle C_1, \sigma\rangle \to \langle C_2, \sigma'\rangle \qquad C_2 \preceq C'}{\langle C, \sigma\rangle \to \langle C', \sigma'\rangle}\ \textsc{Struct}$$

Figure 4.2: Stateful choreographies, semantics.

received message and its local memory, and updates its memory with the result".

Notice that we now use η to range over choreographic statements, i.e., communications ($\mathsf{p}.f \;\texttt{->}\; \mathsf{q}.g$) and internal computation steps ($\mathsf{p}.f$). This is convenient for reasoning about process names in statements. We update the definition of procs as follows.

$$\begin{aligned}
\mathsf{procs}\,(\eta; C) &= \mathsf{procs}(\eta) \cup \mathsf{procs}(C)\\
\mathsf{procs}(\mathbf{0}) &= \emptyset\\
\mathsf{procs}\,(\mathsf{p}.f \;\texttt{->}\; \mathsf{q}.g) &= \{\mathsf{p}, \mathsf{q}\}\\
\mathsf{procs}\,(\mathsf{p}.f) &= \{\mathsf{p}\}
\end{aligned}$$

Semantics Now that processes have functions that may refer to their memories, the execution of a choreography depends on the state of process memories. To formalise this, we need to formulate reductions on more than just choreographies: we will generalise reductions to consider pairs of choreographies and memory states.

It is convenient to abstract from how process memory is concretely implemented, since different processes may use different kinds of data structures. Let v, u, ... range over memory states (or values), which we leave unspecified. Also, let σ range over global memory states, mapping process names to values. Intuitively, σ maps each process to its memory state. For example, $\sigma(\mathsf{p}) = v$ means that process p has v as memory. We assume that σs are total functions (they are never undefined).

We define a semantics for stateful choreographies in terms of reductions $\langle C, \sigma\rangle \to \langle C', \sigma'\rangle$, where $\langle C, \sigma\rangle$ is a *runtime configuration.* The rules defining $\to$ are given in fig. 4.2. It is based as usual on a structural relation $\preceq$, defined by the rule in fig. 4.3.

$$\frac{\mathsf{procs}(\eta) \mathrel{\#} \mathsf{procs}(\eta')}{\eta; \eta' \quad \equiv \quad \eta'; \eta} \ \textsc{Swap}$$

Figure 4.3: Stateful choreographies, structural precongruence.

Let us look at rule Local first, which gives a semantics for local computations. The premise uses the evaluation operator $\downarrow$. We write $f(v_1, \ldots, v_n) \downarrow u$ when the evaluation of function f—where its parameters are instantiated with the values $v_1, \ldots, v_n$—returns the value u. We do not define how $\downarrow$ works concretely, since that depends on the language in which functions are written, which is an implementation detail from our perspective here. In rule Local, we pass the current memory state of p—$\sigma(\mathsf{p})$—to f and get a value v, which then becomes the new state for process p. The notation $\sigma[\mathsf{p} \mapsto v]$ is a mapping update and means "σ, but where p is now mapped to v". Formally:

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

Rule Com is the natural extension of interactions to include internal computation. In the first premise, we evaluate the function f used by the sender p under its memory state, getting a value v. Then, we evaluate the function g used by the receiver under the memory state of the receiver and the value v (the message received from p), obtaining a value u. The memory of the receiver q is then updated to become this value.

We assume that evaluating a local function always terminates (it never takes infinite time). In practice, this means that local computation may yield error values, and that infininte computations may be interrupted by timeouts. We abstract from such details, since what we are interested in here is communications, not the algorithmic details of internal computation. It is easy to plug in existing techniques for ensuring that the parameters passed to local functions are always of the right type, as shown in [Cruz-Filipe and Montesi, 2017].

Rule Swap is updated to deal with all kinds of choreographic statements, be they interactions or internal computations.

Modelling variables Mainstream programming languages typically allow programmers to manipulate different *variables* whose values reside in memory. Luckily, we do not need to extend our model to capture this style, since we can already capture it by using functions and a few conventions.

Let x, y, z, ... range over variable names (variables for short). A variable mapping h is a total function that maps variables to values. From now on, we adopt the following shortcut notation[1]: $\mathsf{p}.f \text{ -> } \mathsf{q}.x$ stands for $\mathsf{p}.f \text{ -> } \mathsf{q}.set^x$, where set^x is the function that replaces x in the variable mapping of q with the value received from p. Formally, given x, the evaluation of set^x is defined as:

$$set^x(h, v) \downarrow h[x \mapsto v].$$

Exercise 14. *Prove the following statement.*

For all $\langle \mathsf{p}.f \text{ -> } \mathsf{q}.x; C, \sigma \rangle$ *such that* $\sigma(\mathsf{q}) = h$ *and* h *is a variable mapping, we have that*

$$\langle \mathsf{p}.f \text{ -> } \mathsf{q}.x; C, \sigma \rangle \quad \rightarrow \quad \langle C, \sigma[\mathsf{q} \mapsto h'] \rangle$$

where $f(\sigma(\mathsf{p})) = v$ *and* $h' = h[x \mapsto v]$*.*

Conversely, it is useful to have a shortcut notation for *sending* the content of a variable. Thus, from now on, we adopt also another shortcut notation: $\mathsf{p}.x \text{ -> } \mathsf{q}.g$ stands for $\mathsf{p}.get^x \text{ -> } \mathsf{q}.g$, where get^x is the function that returns the value of variable x from the variable mapping of p. Formally, given x, the evaluation of get^x is defined as:

$$get^x(h) \downarrow h(x).$$

Example 10. *We can finally give a precise choreography for our example introduced in Part 1, 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

$$\mathsf{Buyer}.title \text{ -> } \mathsf{Seller}.x; \mathsf{Seller}.cat(x) \text{ -> } \mathsf{Buyer}.price; \mathbf{0}$$

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

Exercise 15. *Let* σ *be such that* $\sigma(\mathsf{p}) = h_{\mathsf{p}}$ *and* $\sigma(\mathsf{q}) = h_{\mathsf{q}}$ *for some variable mappings* h_{p} *and* h_{q}*. Also, let* $h_{\mathsf{p}}(title) =$ *"Flowers for Algernon" and cat be a function such that cat("Flowers for Algernon")* $= 100$*. Show the reduction chain for the choreography in the previous example:*

$$\mathsf{Buyer}.title \text{ -> } \mathsf{Seller}.x; \mathsf{Seller}.cat(x) \text{ -> } \mathsf{Buyer}.price; \mathbf{0}.$$

---

[1] Shortcut notations for programming languages are sometimes called syntactic sugar.

$$\begin{aligned} N &::= \mathsf{p} \triangleright_v B \mid N \,|\, N \mid \mathbf{0} \\ B &::= \mathsf{p}!f; B \mid \mathsf{p}?f; B \mid f; B \mid \mathbf{0} \end{aligned}$$

Figure 4.4: Stateful processes, syntax.

$$\frac{f(v) \downarrow v' \qquad g(u, v') \downarrow u'}{\mathsf{p} \triangleright_v \mathsf{q}!f; B \ \mid\ \mathsf{q} \triangleright_u \mathsf{p}?g; B' \quad \to \quad \mathsf{p} \triangleright_v B \ \mid\ \mathsf{q} \triangleright_{u'} B'} \ \textsc{Com}$$

$$\frac{f(v) \downarrow u}{\mathsf{p} \triangleright_v f; B \ \to\ \mathsf{p} \triangleright_u B} \ \textsc{Local} \qquad \frac{N_1 \to N_1'}{N_1 \,|\, N_2 \ \to\ N_1' \,|\, N_2} \ \textsc{Par}$$

$$\frac{N \preceq N_1 \qquad N_1 \to N_2 \qquad N_2 \preceq N'}{N \to N'} \ \textsc{Struct}$$

Figure 4.5: Stateful processes, semantics.

4.1.2 Stateful Processes

Since we updated our choreography model, we also need to update our process model to describe the implementations of choreographies. This is a straightforward extension of our previous calculus of simple processes, obtained by adding memories to processes. The new syntax is given in fig. 4.4.

A process term $\mathsf{p} \triangleright_v B$ now holds a value v, representing the memory state of the process. Send and receive actions are now extended to applying functions. A send action $\mathsf{p}!f$ sends the result of computing f in the local state of the process. Conversely, a receive action $\mathsf{p}?f$ computes f by using the value received from the other process and the local process memory, and then stores the result in the local process memory. An action f executes function f and updates the local memory of the process according to the result.

The semantics of stateful processes is also a straightforward extension, which uses the same evaluation function used for choreographies. The rules are given in fig. 4.5. The rules for the structural precongruence are the same (modulo the addition of values v in process terms, but they are ininfluential), but we report them for the reader's convenience anyway in fig. 4.6.

4.1.3 EndPoint Projection

We have to update our definition of EPP to our new language model.

$$\frac{}{(N_1 \mid N_2) \mid N_3 \;\equiv\; N_1 \mid (N_2 \mid N_3)}\ \text{PA}$$

$$\frac{}{N_1 \mid N_2 \;\equiv\; N_2 \mid N_1}\ \text{PC} \qquad \frac{}{N \mid \mathbf{0} \;\preceq\; N}\ \text{GCN} \qquad \frac{}{\mathsf{p} \triangleright_v \mathbf{0} \;\preceq\; \mathbf{0}}\ \text{GCP}$$

Figure 4.6: Stateful processes, structural precongruence.

$$\llbracket \mathsf{p}.f \text{ -> } \mathsf{q}.g; C \rrbracket_{\mathsf{r}} \;=\; \begin{cases} \mathsf{q}!f; \llbracket C \rrbracket_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{p} \\ \mathsf{p}?g; \llbracket C \rrbracket_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{q} \\ \llbracket C \rrbracket_{\mathsf{r}} & \text{otherwise} \end{cases}$$

$$\llbracket \mathsf{p}.f; C \rrbracket_{\mathsf{r}} \;=\; \begin{cases} f; \llbracket C \rrbracket_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{p} \\ \llbracket C \rrbracket_{\mathsf{r}} & \text{otherwise} \end{cases}$$

$$\llbracket \mathbf{0} \rrbracket_{\mathsf{p}} \;=\; \mathbf{0}$$

Figure 4.7: Behaviour projection for stateful choreographies.

First, as usual, let us gain some intuition. Given any σ, the network implementation of the choreography given in example 10 should look like the following.

$$\begin{array}{l} \mathsf{Buyer} \triangleright_{\sigma(\mathsf{Buyer})} \mathsf{Seller}!title; \mathsf{Seller}?price; \mathbf{0} \\ \mid \\ \mathsf{Seller} \triangleright_{\sigma(\mathsf{Seller})} \mathsf{Buyer}?x; \mathsf{Buyer}!cat(x); \mathbf{0} \end{array}$$

We generalise this intuition in the following definition.

Definition 5 (EndPoint Projection (EPP))**.** *The EPP of a configuration* $\langle C, \sigma \rangle$*, denoted* $\llbracket \langle C, \sigma \rangle \rrbracket$*, is defined as:*

$$\llbracket \langle C, \sigma \rangle \rrbracket = \prod_{\mathsf{p} \in \mathsf{procs}(C)} \mathsf{p} \triangleright_{\sigma(\mathsf{p})} \llbracket C \rrbracket_{\mathsf{p}} \,.$$

We also need to update the definition of behaviour projection—$\llbracket C \rrbracket_{\mathsf{p}}$—since the language of stateful choreographies is different from that of simple choreographies. We display the new rules in fig. 4.7.

Chapter 5

Conditionals and Realisability

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 proceeding 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., they do not have a natural 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.

$$
\begin{aligned}
C &::= I; C \mid \mathbf{0} \\
I &::= \mathsf{p}.f \;\text{->}\; \mathsf{q}.g \mid \mathsf{p}.f \mid \mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \mid \mathbf{0}
\end{aligned}
$$

Figure 5.1: Choreographies with conditionals, syntax.

$$\frac{f(\sigma(\mathsf{p}))\downarrow v \qquad g(\sigma(\mathsf{q}),v)\downarrow u}{\langle \mathsf{p}.f \;\text{->}\; \mathsf{q}.g; C, \sigma\rangle \to \langle C, \sigma[\mathsf{q}\mapsto u]\rangle}\ \textsc{Com}$$

$$\frac{f(\sigma(\mathsf{p}))\downarrow v}{\langle \mathsf{p}.f; C, \sigma\rangle \to \langle C, \sigma[\mathsf{p}\mapsto v]\rangle}\ \textsc{Local}$$

$$\frac{i = 1 \text{ if } f(\sigma(\mathsf{p}))\downarrow \mathtt{true},\ i = 2 \text{ otherwise}}{\langle \mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2; C, \sigma\rangle \to \langle C_i; C, \sigma\rangle}\ \textsc{Cond}$$

$$\frac{C \preceq C_1 \qquad \langle C_1, \sigma\rangle \to \langle C_2, \sigma'\rangle \qquad C_2 \preceq C'}{\langle C, \sigma\rangle \to \langle C', \sigma'\rangle}\ \textsc{Struct}$$

Figure 5.2: Choreographies with conditionals, semantics.

The new term $\mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$ means "process p runs function f, and the choreography proceeds as C_1 if the result is the value `true`, or proceeds as C_2 otherwise". (So we now assume that the set of possible values contains booleans.) The condition used in a conditional term is typically called a *guard*.

Extending function procs to the new syntax is easy:

$$\begin{aligned}
\mathsf{procs}\,(I; C) &= \mathsf{procs}(I) \cup \mathsf{procs}(C)\\
\mathsf{procs}(\mathbf{0}) &= \emptyset\\
\mathsf{procs}\,(\mathsf{p}.f \;\text{->}\; \mathsf{q}.g) &= \{\mathsf{p}, \mathsf{q}\}\\
\mathsf{procs}\,(\mathsf{p}.f) &= \{\mathsf{p}\}\\
\mathsf{procs}\,(\mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2) &= \{\mathsf{p}\} \cup \mathsf{procs}(C_1) \cup \mathsf{procs}(C_2).
\end{aligned}$$

Semantics The reduction semantics of choreographies with conditionals is given by the rules in fig. 5.2.

The new rule Cond formalises the intended meaning of conditionals, choosing the right branch depending on the result of the guard.

Updating structural precongruence is a bit more involved. Let us do the easy part first. Observe that an I can be $\mathbf{0}$. This is necessary for our semantics to be defined, since a conditional may contain a $\mathbf{0}$ branch. Consider the choreography $\mathsf{if}\ \mathsf{p}.\mathtt{true}\ \mathsf{then}\ \mathbf{0}\ \mathsf{else}\ \mathbf{0}; C$. By rule Cond, this reduces to $\mathbf{0}; C$. That $\mathbf{0}$ is now "garbage", in the sense that it does not specify any behaviour so we should just get rid of it. For this purpose, we introduce the following rule.

$$\frac{}{\mathbf{0}; C \preceq C}\ \textsc{GCNil}$$

$$\frac{\mathsf{procs}(I) \mathrel{\#} \mathsf{procs}(I')}{I; I' \quad \equiv \quad I'; I}\ \text{I-I}$$

$$\frac{\mathsf{p} \notin \mathsf{procs}(I)}{I; \mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \quad \equiv \quad \mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ (I; C_1)\ \mathsf{else}\ (I; C_2)}\ \textsc{I-Cond}$$

$$\frac{\mathsf{p} \neq \mathsf{q}}{\begin{array}{c} \mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ (\mathsf{if}\ \mathsf{q}.g\ \mathsf{then}\ C_1^1\ \mathsf{else}\ C_2^1)\,; C\ \mathsf{else}\ (\mathsf{if}\ \mathsf{q}.g\ \mathsf{then}\ C_1^2\ \mathsf{else}\ C_2^2)\,; C' \\ \equiv \\ \mathsf{if}\ \mathsf{q}.g\ \mathsf{then}\ (\mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ C_1^1; C\ \mathsf{else}\ C_1^2; C')\ \mathsf{else}\ (\mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ C_2^1; C\ \mathsf{else}\ C_2^2; C') \end{array}}\ \textsc{Cond-Cond}$$

$$\frac{}{\mathbf{0}; C \preceq C}\ \text{GC}\textsc{Nil}$$

Figure 5.3: Choreographies with conditionals, structural precongruence.

Now for the more sophisticated part. Consider the following choreography.

$$(\mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ (\mathsf{if}\ \mathsf{q}.g\ \mathsf{then}\ C_{11}\ \mathsf{else}\ C_{12})\,; \mathbf{0}\ \mathsf{else}\ (\mathsf{if}\ \mathsf{q}.g\ \mathsf{then}\ C_{21}\ \mathsf{else}\ C_{22})\,; \mathbf{0})\,; \mathbf{0}$$

Since p and q are different, there is no causal dependency that gives an ordering in which the two conditionals to be executed! This means that we should define precongruence rules that capture out-of-order execution of conditionals, too. Another revealing example is the following.

$$\mathsf{p}.f \ \text{->}\ \mathsf{q}.g; \mathsf{if}\ \mathsf{r}.f'\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2; \mathbf{0}$$

Since p, q, and r are all different, it may happen that r evaluates its conditional before p and q interact. Our structural precongruence should capture this kind of out-of-order behaviour too.

The new rules for structural precongruence that follow the intuition that we have just built are displayed in fig. 5.3.

Rule Cond-I allows us to swap instructions inside and outside of conditionals, in case that they do not involve the process in the guard. Observe that when we bring a term inside it gets duplicated in both branches of the conditionals, since we need to ensure that it will be executed regardless of which branch is chosen at runtime.

Rule Cond-Cond allows us to swap two independent conditionals. Notice that branches C_2^1 and C_1^2 get exchanged, in order to preserve the combined effects of evaluating the two guards. We can check that this exchange

$$
\begin{aligned}
N &::= \mathsf{p} \triangleright_v B \mid N \mid N \mid \mathbf{0} \\
B &::= \mathsf{p}!f; B \mid \mathsf{p}?f; B \mid f; B \mid \mathsf{if}\ f\ \mathsf{then}\ B_1\ \mathsf{else}\ B_2; B \mid \mathbf{0}; B \mid \mathbf{0}
\end{aligned}
$$

Figure 5.4: Processes with conditionals, syntax.

makes sense with the following exercise, which verifies that swapping the two conditionals does not introduce new behaviour.

Exercise 16. *Prove the following statement.*

Let σ be a global memory state. We have the following reduction chain without using rule Struct *for some j and i in $\{1, 2\}$*

$$\left\langle \mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ \left(\mathsf{if}\ \mathsf{q}.g\ \mathsf{then}\ C^1_1\ \mathsf{else}\ C^1_2\right)\ \mathsf{else}\ \left(\mathsf{if}\ \mathsf{q}.g\ \mathsf{then}\ C^2_1\ \mathsf{else}\ C^2_2\right), \sigma \right\rangle \rightarrow\!\!\rightarrow \left\langle C^j_i, \sigma \right\rangle$$

if and only if we have also the following reduction chain

$$\left\langle \mathsf{if}\ \mathsf{q}.g\ \mathsf{then}\ \left(\mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ C^1_1\ \mathsf{else}\ C^2_1\right)\ \mathsf{else}\ \left(\mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ C^1_2\ \mathsf{else}\ C^2_2\right), \sigma \right\rangle \rightarrow\!\!\rightarrow \left\langle C^j_i, \sigma \right\rangle .$$

Suggestion: proceed by cases on $f(\sigma(\mathsf{p})) \downarrow v$ (what can v be?) and $g(\sigma(\mathsf{q})) \downarrow u$ (what can u be?) and their consequences on the reduction chains.

5.1.2 Processes

Introducing conditionals to our process calculus follows the same principles as for choreographies. 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. Therefore, before we dive into a general definition, it is useful to look at an example. Consider the following choreography.

$$C_{\mathsf{unproj}} \triangleq\ (\mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ \mathsf{p}.\texttt{true}\ \texttt{->}\ \mathsf{q}.x; \mathbf{0}\ \mathsf{else}\ \mathbf{0})\ ; \mathbf{0}$$

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, so for any σ we get:

$$[\![\langle C_{\mathsf{unproj}}, \sigma\rangle]\!] = \ \mathsf{p} \triangleright_{\sigma(\mathsf{p})} [\![C_{\mathsf{unproj}}]\!]_{\mathsf{p}} \mid \mathsf{q} \triangleright_{\sigma(\mathsf{q})} [\![C_{\mathsf{unproj}}]\!]_{\mathsf{q}} . \tag{5.1}$$

$$\frac{f(v)\downarrow v' \qquad g(u,v')\downarrow u'}{\mathsf{p}\triangleright_v \mathsf{q}!f;B \ \mid\ \mathsf{q}\triangleright_u \mathsf{p}?g;B' \quad \rightarrow \quad \mathsf{p}\triangleright_v B \ \mid\ \mathsf{q}\triangleright_{u'} B'}\ \textsc{Com}$$

$$\frac{f(v)\downarrow u}{\mathsf{p}\triangleright_v f;B \ \rightarrow\ \mathsf{p}\triangleright_u B}\ \textsc{Local}$$

$$\frac{i=1 \text{ if } f(v)\downarrow \texttt{true},\ i=2 \text{ otherwise}}{\mathsf{p}\triangleright_v (\mathsf{if}\ f\ \mathsf{then}\ B_1\ \mathsf{else}\ B_2)\,;B \rightarrow \mathsf{p}\triangleright_v B_i;B}\ \textsc{Cond}$$

$$\frac{N_1 \rightarrow N_1'}{N_1 \mid N_2 \ \rightarrow\ N_1' \mid N_2}\ \textsc{Par} \qquad \frac{N \preceq N_1 \quad N_1 \rightarrow N_2 \quad N_2 \preceq N'}{N \rightarrow N'}\ \textsc{Struct}$$

Figure 5.5: Processes with conditionals, semantics.

$$\frac{}{(N_1 \mid N_2) \mid N_3 \ \equiv\ N_1 \mid (N_2 \mid N_3)}\ \text{PA} \qquad \frac{}{\mathbf{0};B \preceq B}\ \text{GCB}$$

$$\frac{}{N_1 \mid N_2 \ \equiv\ N_2 \mid N_1}\ \text{PC} \qquad \frac{}{N \mid \mathbf{0} \ \preceq\ N}\ \text{GCN} \qquad \frac{}{\mathsf{p}\triangleright_v \mathbf{0} \ \preceq\ \mathbf{0}}\ \text{GCP}$$

Figure 5.6: Processes with conditionals, structural precongruence.

The projection for p seems easy to achieve:

$$[\![C_{\mathsf{unproj}}]\!]_{\mathsf{p}} = (\mathsf{if}\ f\ \mathsf{then}\ \mathsf{q}!\mathtt{true};\mathbf{0}\ \mathsf{else}\ \mathbf{0})\,;\mathbf{0}.$$

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 $\mathsf{p}.\mathtt{true}$ -> $\mathsf{q}.x$. If we choose to project that (and we should, since the choreography contains it), we obtain:

$$[\![C_{\mathsf{unproj}}]\!]_{\mathsf{q}} = \mathsf{p}?x;\mathbf{0};\mathbf{0}. \tag{5.2}$$

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

$$[\![C_{\mathsf{unproj}}]\!]_{\mathsf{q}} = \mathbf{0};\mathbf{0}. \tag{5.3}$$

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

Exercise 17. *Show that, if we adopt the behaviour projection in eq.* (5.2)*, the network in eq.* (5.1) *may reduce to a network that is not the EPP of what* C_{unproj} *reduces to.*

Hint: consider the case of $\langle C_{\mathsf{unproj}}, \sigma\rangle$ *for some* σ *such that* $f(\sigma(\mathsf{p}))\downarrow \mathtt{false}$*.*

Exercise 18. *Show that, if we adopt the behaviour projection in eq.* (5.3)*, the network in eq.* (5.1) *may reduce to a network that is not the EPP of what* C_{unproj} *reduces to.*

Hint: consider the case of $\langle C_{\mathsf{unproj}}, \sigma\rangle$ *for some* σ *such that* $f(\sigma(\mathsf{p}))\downarrow \mathtt{true}$*.*

5.2 Introduction to realisability

Recall our problematic example for projecting conditionals.

$$C_{\mathsf{unproj}} \triangleq (\mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ \mathsf{p}.\mathtt{true}\ \text{->}\ \mathsf{q}.x;\mathbf{0}\ \mathsf{else}\ \mathbf{0})\,;\mathbf{0}$$

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.

5.3 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 behaviour projection, in fig. 5.7.

Definition 6 (EndPoint Projection (EPP))**.** *The EPP of a configuration* $\langle C, \sigma \rangle$*, denoted* $[\![\langle C, \sigma \rangle]\!]$*, is defined as:*

$$[\![\langle C, \sigma \rangle]\!] = \prod_{\mathsf{p} \in \mathsf{procs}(C)} \mathsf{p} \triangleright_{\sigma(\mathsf{p})} [\![C]\!]_{\mathsf{p}}$$

.

$$\llbracket \mathsf{p}.f \;\texttt{->}\; \mathsf{q}.g; C \rrbracket_{\mathsf{r}} \;=\; \begin{cases} \mathsf{q}!f; \llbracket C \rrbracket_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{p} \\ \mathsf{p}?g; \llbracket C \rrbracket_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{q} \\ \llbracket C \rrbracket_{\mathsf{r}} & \text{otherwise} \end{cases}$$

$$\llbracket \mathsf{p}.f; C \rrbracket_{\mathsf{r}} \;=\; \begin{cases} f; \llbracket C \rrbracket_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{p} \\ \llbracket C \rrbracket_{\mathsf{r}} & \text{otherwise} \end{cases}$$

$$\llbracket \mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2; C \rrbracket_{\mathsf{r}} \;=\; \begin{cases} (\mathsf{if}\ f\ \mathsf{then}\ \llbracket C_1 \rrbracket_{\mathsf{r}}\ \mathsf{else}\ \llbracket C_2 \rrbracket_{\mathsf{r}})\,; \llbracket C \rrbracket_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{p} \\ \llbracket C_1 \rrbracket_{\mathsf{r}}\,; \llbracket C \rrbracket_{\mathsf{r}} \quad \text{if } \mathsf{r} \neq \mathsf{p} \text{ and } \llbracket C_1 \rrbracket_{\mathsf{r}} = \llbracket C_2 \rrbracket_{\mathsf{r}} \end{cases}$$

$$\llbracket \mathbf{0}; C \rrbracket_{\mathsf{p}} \;=\; \llbracket C \rrbracket_{\mathsf{p}}$$

$$\llbracket \mathbf{0} \rrbracket_{\mathsf{p}} \;=\; \mathbf{0}$$

Figure 5.7: Behaviour projection for choreographies with conditionals.

The rule for projecting a choreographic conditional includes a check that prevents projecting problematic choreographies. Namely, when we are projecting a conditional $\mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$, 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* ($\llbracket C_1 \rrbracket_{\mathsf{r}} = \llbracket C_2 \rrbracket_{\mathsf{r}}$ in the rule).

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:

$$C_{\mathsf{unproj}} \triangleq (\mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ \mathsf{p}.\mathtt{true} \;\texttt{->}\; \mathsf{q}.x; \mathbf{0}\ \mathsf{else}\ \mathbf{0})\,; \mathbf{0}. \tag{5.4}$$

Then we can prove that it cannot be projected.

Proposition 2. *Let C_{unproj} be the choreography in eq.* (5.12)*. For all σ, $\llbracket \langle C_{\mathsf{unproj}}, \sigma \rangle \rrbracket$ is undefined.*

Proof. For $[\![\langle C_{\mathsf{unproj}}, \sigma\rangle]\!]$ to be defined, we need $[\![C_{\mathsf{unproj}}]\!]_{\mathsf{q}}$ to be defined (by definition 6). Since C_{unproj} is a conditional, the only rule that we can apply for $[\![C_{\mathsf{unproj}}]\!]_{\mathsf{q}}$ is the third one in fig. 5.7. We proceed by cases according to the definition of the rule. Since $\mathsf{p} \neq \mathsf{q}$, we cannot apply the first case. The only remaining option is the second one, which requires the following.

- $\mathsf{p} \neq \mathsf{q}$. This holds.
- $[\![\mathsf{p}.\mathtt{true} \texttt{->} \mathsf{q}.x; \mathbf{0}]\!]_{\mathsf{q}} = [\![\mathbf{0}]\!]_{\mathsf{q}}$. This does not hold, because $[\![\mathsf{p}.\mathtt{true} \texttt{->} \mathsf{q}.x; \mathbf{0}]\!]_{\mathsf{q}} = \mathsf{p}?x; \mathbf{0}$ which is not equal to $[\![\mathbf{0}]\!]_{\mathsf{q}} = \mathbf{0}$.

Thus, $[\![C_{\mathsf{unproj}}]\!]_{\mathsf{q}}$ is undefined and, consequently, the thesis follows. ☐

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.

$$(\mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ \mathsf{p}.\mathtt{true} \texttt{->} \mathsf{q}.x; \mathbf{0}\ \mathsf{else}\ \mathsf{p}.\mathtt{false} \texttt{->} \mathsf{q}.x; \mathbf{0})\ ; \mathbf{0} \tag{5.5}$$

Exercise 19. *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 $\mathsf{q}.x$ returns the value stored in x at q. The symbol 0 in the choreography below is just the natural number 0, representing no money.

$$\left(\begin{array}{rl} \mathsf{if}\ \mathsf{p}.f\ \mathsf{then} & \mathsf{p}.\mathtt{true} \texttt{->} \mathsf{q}.x; \\ & \left(\begin{array}{rl} \mathsf{if}\ \mathsf{q}.x\ \mathsf{then} & \mathsf{q}.\mathit{money} \texttt{->} \mathsf{p}.m; \mathbf{0} \\ \mathsf{else} & \mathsf{q}.0 \texttt{->} \mathsf{p}.m; \mathbf{0} \end{array}\right); \mathbf{0} \\ \mathsf{else} & \mathsf{p}.\mathtt{false} \texttt{->} \mathsf{q}.x; \\ & \left(\begin{array}{rl} \mathsf{if}\ \mathsf{q}.x\ \mathsf{then} & \mathsf{q}.\mathit{money} \texttt{->} \mathsf{p}.m; \mathbf{0} \\ \mathsf{else} & \mathsf{q}.0 \texttt{->} \mathsf{p}.m; \mathbf{0} \end{array}\right); \mathbf{0} \end{array}\right); \mathbf{0} \tag{5.6}$$

Exercise 20. *Write the behaviour projection of* q *for the choreography in eq.* (5.6).

$$
\begin{aligned}
C &::= I; C \mid \mathbf{0} \\
I &::= \mathsf{p}.f \texttt{ -> } \mathsf{q}.g \mid \mathsf{p} \texttt{->} \mathsf{q}[l] \mid \mathsf{p}.f \mid \mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \mid \mathbf{0}
\end{aligned}
$$

Figure 5.8: Choreographies with selections, syntax.

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.

$$
(\mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ \mathsf{q}.money \texttt{ -> } \mathsf{p}.m; \mathbf{0}\ \mathsf{else}\ \mathbf{0})\,; \mathbf{0} \tag{5.7}
$$

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!

5.4 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.

5.4.1 Choreographies

Syntax The updated syntax of choreographies is given in fig. 5.8. The new primitive is $\mathsf{p} \,\texttt{->}\, \mathsf{q}[l]$, 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 $\mathsf{p} \,\texttt{->}\, \mathsf{q}[l]$, 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.

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—$\mathsf{q}.\mathit{money} \,\texttt{->}\, \mathsf{p}.m; \mathbf{0}$. 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—$\mathbf{0}$. We can formalise this intuition as the following choreography.

$$\Big(\ \mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ \ \mathsf{p} \,\texttt{->}\, \mathsf{q}[\textsc{pay}]; \mathsf{q}.\mathit{money} \,\texttt{->}\, \mathsf{p}.m; \mathbf{0}\ \ \mathsf{else}\ \ \mathsf{p} \,\texttt{->}\, \mathsf{q}[\textsc{no}]; \mathbf{0}\ \ \Big)\,; \mathbf{0} \tag{5.8}$$

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 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.

$$\frac{}{\langle \mathsf{p} \,\texttt{->}\, \mathsf{q}[l]; C, \sigma\rangle \to \langle C, \sigma\rangle}\ \textsc{Sel}$$

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

The rules defining structural precongruence are the same (see fig. 5.10), but since now an I can be a selection term—$\mathsf{p} \,\texttt{->}\, \mathsf{q}[l]$—we need to update the definition of procs as follows.

$$\frac{f(\sigma(\mathsf{p}))\downarrow v \quad g(\sigma(\mathsf{q}),v)\downarrow u}{\langle \mathsf{p}.f \rightarrow \mathsf{q}.g; C, \sigma\rangle \to \langle C, \sigma[\mathsf{q}\mapsto u]\rangle}\ \textsc{Com} \qquad \frac{}{\langle \mathsf{p} \rightarrow \mathsf{q}[l]; C, \sigma\rangle \to \langle C, \sigma\rangle}\ \textsc{Sel}$$

$$\frac{f(\sigma(\mathsf{p}))\downarrow v}{\langle \mathsf{p}.f; C, \sigma\rangle \to \langle C, \sigma[\mathsf{p}\mapsto v]\rangle}\ \textsc{Local}$$

$$\frac{i = 1 \text{ if } f(\sigma(\mathsf{p}))\downarrow \mathtt{true},\ i = 2 \text{ otherwise}}{\langle \mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2; C, \sigma\rangle \to \langle C_i; C, \sigma\rangle}\ \textsc{Cond}$$

$$\frac{C \preceq C_1 \quad \langle C_1, \sigma\rangle \to \langle C_2, \sigma'\rangle \quad C_2 \preceq C'}{\langle C, \sigma\rangle \to \langle C', \sigma'\rangle}\ \textsc{Struct}$$

Figure 5.9: Choreographies with selections, semantics.

$$\begin{aligned}
\mathsf{procs}\,(I; C) &= \mathsf{procs}(I) \cup \mathsf{procs}(C)\\
\mathsf{procs}(\mathbf{0}) &= \emptyset\\
\mathsf{procs}\,(\mathsf{p}.f \rightarrow \mathsf{q}.g) &= \{\mathsf{p},\mathsf{q}\}\\
\mathsf{procs}\,(\mathsf{p} \rightarrow \mathsf{q}[l]) &= \{\mathsf{p},\mathsf{q}\}\\
\mathsf{procs}\,(\mathsf{p}.f) &= \{\mathsf{p}\}\\
\mathsf{procs}\,(\mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2) &= \{\mathsf{p}\} \cup \mathsf{procs}(C_1) \cup \mathsf{procs}(C_2)
\end{aligned}$$

5.4.2 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. 5.11 to 5.13.

The new primitives are $\mathsf{p} \oplus l; B$ and $\mathsf{p}\,\&\{l_i : B_i\}_{i\in I}; B$. A term $\mathsf{p} \oplus l; B$ sends the choice of a label l to p and then proceeds as B. Dually, a term $\mathsf{p}\,\&\{l_i : B_i\}_{i\in I}; B$ (also called *branching term*, or simply a branching) offers the possibility to choose from many behaviours $\{B_i\}_{i\in I}$ 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 $\mathsf{p}\,\&\{l_1 : B_1, l_2 : B_2\}; B$ as "receive a label, and then run $B_1; B$ if the received label is l_1, or run $B_2; B$ if the received label is l_2". This intuition is formalised by rule Sel in fig. 5.12, where the sender selects a label among those offered by the receiver ($j \in I$).

$$\frac{\mathsf{procs}(I) \mathrel{\#} \mathsf{procs}(I')}{I; I' \quad\equiv\quad I'; I}\ \textsc{I-I}$$

$$\frac{\mathsf{p} \notin \mathsf{procs}(I)}{I; \mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \quad\equiv\quad \mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ (I; C_1)\ \mathsf{else}\ (I; C_2)}\ \textsc{I-Cond}$$

$$\frac{\mathsf{p} \notin \mathsf{procs}(I) \qquad I \neq \mathbf{0}}{\mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2; I \quad\equiv\quad \mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ (C_1; I)\ \mathsf{else}\ (C_2; I)}\ \textsc{Cond-I}$$

$$\frac{}{\mathbf{0}; C \preceq C}\ \textsc{GCNil}$$

Figure 5.10: Choreographies with selections, structural precongruence.

$$\begin{aligned}
N &::= \mathsf{p} \triangleright_v B \mid N \mid N \mid \mathbf{0} \\
B &::= \mathsf{p}!f; B \mid \mathsf{p}?f; B \mid \mathsf{p} \oplus l; B \mid \mathsf{p} \,\&\{l_i : B_i\}_{i \in I}; B \\
&\quad\ \mid \mathsf{if}\ f\ \mathsf{then}\ B_1\ \mathsf{else}\ B_2; B \mid \mathbf{0}; B \mid \mathbf{0}
\end{aligned}$$

Figure 5.11: Processes with selections, syntax.

5.4.3 EPP for choreographies with selections

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

$$\big(\ \mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ \ \mathsf{p} \mathbin{\texttt{->}} \mathsf{q}[\textsc{pay}]; \mathsf{q}.\mathit{money} \mathbin{\texttt{->}} \mathsf{p}.m; \mathbf{0}\ \ \mathsf{else}\ \ \mathsf{p} \mathbin{\texttt{->}} \mathsf{q}[\textsc{no}]; \mathbf{0}\ \ \big); \mathbf{0}$$

Given any σ, we can now manually write an operationally-equivalent network, as follows.

$$\begin{array}{ll}
\mathsf{p} \triangleright_{\sigma(\mathsf{p})} & \mathsf{if}\ f\ \mathsf{then}\ \mathsf{q} \oplus \textsc{pay}; \mathsf{q}?m; \mathbf{0}\ \mathsf{else}\ \mathsf{q} \oplus \textsc{no}; \mathbf{0} \\
| & \\
\mathsf{q} \triangleright_{\sigma(\mathsf{q})} & \mathsf{p} \,\&\{\textsc{pay} : \mathsf{p}!\mathit{money}; \mathbf{0},\ \textsc{no} : \mathbf{0}\}; \mathbf{0}
\end{array} \tag{5.9}$$

Exercise 21. *Write the reduction chains of the choreography in eq.* (5.8) *and the network in eq.* (5.9)*. Do they mimic each other?*

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

$$\frac{f(v)\downarrow v' \qquad g(u,v')\downarrow u'}{\mathsf{p}\triangleright_v \mathsf{q}!f;B \;\mid\; \mathsf{q}\triangleright_u \mathsf{p}?g;B' \quad\rightarrow\quad \mathsf{p}\triangleright_v B \;\mid\; \mathsf{q}\triangleright_{u'} B'}\ \textsc{Com}$$

$$\frac{j\in I}{\mathsf{p}\triangleright_v \mathsf{q}\oplus l_j;B \;\mid\; \mathsf{q}\triangleright_u \mathsf{p}\,\&\{l_i : B_i\}_{i\in I};B' \quad\rightarrow\quad \mathsf{p}\triangleright_v B \;\mid\; \mathsf{q}\triangleright_u B_j;B'}\ \textsc{Sel}$$

$$\frac{i=1 \text{ if } f(v)\downarrow\texttt{true},\ i=2 \text{ otherwise}}{\mathsf{p}\triangleright_v (\mathsf{if}\ f\ \mathsf{then}\ B_1\ \mathsf{else}\ B_2)\,;B \rightarrow \mathsf{p}\triangleright_v B_i;B}\ \textsc{Cond}$$

$$\frac{N_1\rightarrow N_1'}{N_1\,|\,N_2 \;\rightarrow\; N_1'\,|\,N_2}\ \textsc{Par} \qquad \frac{N\preceq N_1 \quad N_1\rightarrow N_2 \quad N_2\preceq N'}{N\rightarrow N'}\ \textsc{Struct}$$

Figure 5.12: Processes with selections, semantics.

$$\frac{}{(N_1\,|\,N_2)\,|\,N_3 \;\equiv\; N_1\,|(N_2\,|\,N_3)}\ \text{PA} \qquad \frac{}{\mathbf{0};B\preceq B}\ \text{GCB}$$

$$\frac{}{N_1\,|\,N_2 \;\equiv\; N_2\,|\,N_1}\ \text{PC} \qquad \frac{}{N\,|\,\mathbf{0} \;\preceq\; N}\ \text{GCN} \qquad \frac{}{\mathsf{p}\triangleright_v \mathbf{0} \;\preceq\; \mathbf{0}}\ \text{GCP}$$

Figure 5.13: Processes with selections, structural precongruence.

$$
\begin{array}{rcl}
\mathsf{p}!f; B_1 \sqcup \mathsf{p}!f; B_2 & = & \mathsf{p}!f; (B_1 \sqcup B_2) \\
\mathsf{p}?f; B_1 \sqcup \mathsf{p}?f; B_2 & = & \mathsf{p}?f; (B_1 \sqcup B_2) \\
\mathsf{p} \oplus l; B_1 \sqcup \mathsf{p} \oplus l; B_2 & = & \mathsf{p} \oplus l; (B_1 \sqcup B_2) \\
\begin{array}{l} \mathsf{if}\, f\, \mathsf{then}\, B_1\, \mathsf{else}\, B_1'; B_1'' \\ \qquad\qquad \sqcup \\ \mathsf{if}\, f\, \mathsf{then}\, B_2\, \mathsf{else}\, B_2'; B_2'' \end{array} & = & \mathsf{if}\, f\, \mathsf{then}\ (B_1 \sqcup B_2)\ \mathsf{else}\ (B_1' \sqcup B_2')\, ; (B_1'' \sqcup B_2'') \\
\mathbf{0}; B_1 \sqcup \mathbf{0}; B_2 & = & \mathbf{0}; (B_1 \sqcup B_2) \\
\mathbf{0} \sqcup \mathbf{0} & = & \mathbf{0}
\end{array}
$$

$$
\mathsf{p}\,\&\{l_i : B_i\}_{i \in I}; B_1 \sqcup \mathsf{p}\,\&\{l_j : B_j'\}_{j \in J}; B_2 \ = \\
\mathsf{p}\,\&\,\Big(\ \{l_k : (B_k \sqcup B_k')\}_{k \in I \cap J} \cup \{l_i : B_i\}_{i \in I \setminus J} \cup \{l_j : B_j'\}_{j \in J \setminus I}\ \Big)\, ; (B_1 \sqcup B_2)
$$

Figure 5.14: Merging operator for processes with selections.

as follows.

$$
[\![\mathsf{p} \texttt{->} \mathsf{q}[l]; C]\!]_{\mathsf{r}} \ = \ \begin{cases} \mathsf{q} \oplus l; [\![C]\!]_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{p} \\ \mathsf{p}\,\&\{l : [\![C]\!]_{\mathsf{r}}\}; \mathbf{0} & \text{if } \mathsf{r} = \mathsf{q} \\ [\![C]\!]_{\mathsf{r}} & \text{otherwise} \end{cases}
$$

$$
[\![\mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2; C]\!]_{\mathsf{r}} \ = \ \begin{cases} (\mathsf{if}\ f\ \mathsf{then}\ [\![C_1]\!]_{\mathsf{r}}\ \mathsf{else}\ [\![C_2]\!]_{\mathsf{r}})\, ; [\![C]\!]_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{p} \\ ([\![C_1]\!]_{\mathsf{r}} \sqcup [\![C_2]\!]_{\mathsf{r}})\, ; [\![C]\!]_{\mathsf{r}} & \text{otherwise} \end{cases}
$$

The projection of a selection $\mathsf{p} \texttt{->} \mathsf{q}[l]$ is simple: the sender is projected to the sending of the label—$\mathsf{q} \oplus l$—whereas the receiver is projected to a branching with a single branch with label l—$\mathsf{p}\,\&\{l : [\![C]\!]_{\mathsf{r}}\}$. 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 *merge 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, $B_1 \sqcup B_2$ is defined inductively on the structure of B_1 and B_2. The rules defining $\sqcup$ are displayed in fig. 5.14. These are an adaptation to our model from the definition originally given by Carbone et al. [2007].

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

Exercise 22. *Write the behaviour projection for process* q *in the following choreography.*

$$\left(\begin{array}{ll} \mathsf{if}\ \mathsf{p}.f\ \mathsf{then} & \mathsf{p} \texttt{->} \mathsf{q}[\textsc{pay}]; \mathsf{q}.money \texttt{->} \mathsf{p}.m; \mathbf{0} \\ \qquad \mathsf{else} & \\ & \left(\begin{array}{ll} \mathsf{if}\ \mathsf{p}.g\ \mathsf{then} & \mathsf{p} \texttt{->} \mathsf{q}[\textsc{reimburse}]; \mathsf{p}.money \texttt{->} \mathsf{q}.m; \mathbf{0} \\ \qquad \mathsf{else} & \mathsf{p} \texttt{->} \mathsf{q}[\textsc{no}]; \mathbf{0} \end{array}\right); \mathbf{0} \end{array}\right); \mathbf{0}$$

5.4.4 Operational correspondence for EPP with selections

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

$$C_{\mathsf{cond}} \triangleq (\mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ \mathsf{p} \texttt{->} \mathsf{q}[\textsc{left}]; \mathbf{0}\ \mathsf{else}\ \mathsf{p} \texttt{->} \mathsf{q}[\textsc{right}]; \mathbf{0})\ ; \mathbf{0}$$

Its projection is the following, for any σ.

$$\begin{array}{rl} [\![\langle C_{\mathsf{cond}}, \sigma\rangle]\!] = & \mathsf{p} \triangleright_{\sigma(\mathsf{p})} (\mathsf{if}\ f\ \mathsf{then}\ \mathsf{q} \oplus \textsc{left}; \mathbf{0}\ \mathsf{else}\ \mathsf{q} \oplus \textsc{right}; \mathbf{0})\ ; \mathbf{0} \\ & \mid \\ & \mathsf{q} \triangleright_{\sigma(\mathsf{q})} \mathsf{p}\ \&\{\textsc{left} : \mathbf{0}, \textsc{right} : \mathbf{0}\}; \mathbf{0}; \mathbf{0} \end{array}$$

Let σ be such that $f(\sigma(\mathsf{p})) \downarrow \mathtt{true}$ (a similar reasoning to the one that follows holds for the case in which $f(\sigma(\mathsf{p}))$ does not evaluate to $\mathtt{true}$). Then, by rule COND for choreographies we can execute the conditional at p and obtain the following reduction:

$$\langle C_{\mathsf{cond}}, \sigma\rangle \to \langle \mathsf{p} \texttt{->} \mathsf{q}[\textsc{left}]; \mathbf{0}; \mathbf{0}, \sigma\rangle\ . \tag{5.10}$$

The corresponding reduction in $[\![\langle C_{\mathsf{cond}}, \sigma\rangle]\!]$, of course, should execute the same conditional at p. This yields the following reduction.

$$[\![\langle C_{\mathsf{cond}}, \sigma\rangle]\!] \to \begin{array}{l} \mathsf{p} \triangleright_{\sigma(\mathsf{p})} \mathsf{q} \oplus \textsc{left}; \mathbf{0}; \mathbf{0} \\ \mid \\ \mathsf{q} \triangleright_{\sigma(\mathsf{q})} \mathsf{p}\ \&\{\textsc{left} : \mathbf{0}, \textsc{right} : \mathbf{0}\}; \mathbf{0}; \mathbf{0} \end{array} \tag{5.11}$$

Let C_{left} be the choreography on the right-hand side in eq. (5.10) (the choreography after the reduction) and N_{left} be the network on the right-hand side in eq. (5.11) (the network after the reduction). Ideally, following our previous statements of operational correspondence, we would expect that

$N_{\mathsf{left}} \preceq [\![\langle C_{\mathsf{left}}, \sigma\rangle]\!]$. But this is not the case, since we can see that the EPP of C_{left} is different:

$$[\![\langle C_{\mathsf{left}}, \sigma\rangle]\!] = [\![\langle \mathsf{p} \,\text{->}\, \mathsf{q}[\text{LEFT}]; \mathbf{0}; \mathbf{0}, \sigma\rangle]\!] = \begin{array}{l} \mathsf{p} \triangleright_{\sigma(\mathsf{p})} \mathsf{q} \oplus \text{LEFT}; \mathbf{0} \\ \mid \\ \mathsf{q} \triangleright_{\sigma(\mathsf{q})} \mathsf{p} \,\&\{\text{LEFT} : \mathbf{0}\}; \mathbf{0} \end{array} .$$

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 $\mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$ in a choreography is reduced, we "cut off" either C_1 or C_2 in a single step and they may contain code that involves many processes, not just p. Instead, in the process calculus, 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 operational correspondence. First, we define formally what it means to have a network with "extra branches".

Definition 7. *We write $N \sqsupseteq N'$ when N has at least as many branches in branchings as N'. Formally, $\sqsupseteq$ is defined inductively as follows.*

- $\mathbf{0} \sqsupseteq \mathbf{0}$*;*
- $\mathsf{p} \triangleright_v B \sqsupseteq \mathsf{p} \triangleright_v B'$ *if* $B \sqcup B' = B$*;*
- $N_1 \mid N_2 \sqsupseteq N_1' \mid N_2'$ *if* $N_1 \sqsupseteq N_1'$ *and* $N_2 \sqsupseteq N_2'$*.*

Then, we can reformulate operational correspondence appropriately.

Theorem 4 (Operational Correspondence)**.** *Let $[\![\langle C, \sigma\rangle]\!] = N$. Then,*

Completeness *If $\langle C, \sigma\rangle \to \langle C', \sigma'\rangle$ for some C' and σ', then there exists N' such that $N \to N'$ and $N' \preceq \sqsupseteq [\![\langle C', \sigma'\rangle]\!]$.*

Soundness *If $N \to N'$ for some N', then there exists C' and σ' such that $\langle C, \sigma\rangle \to \langle C', \sigma'\rangle$ and $N' \preceq \sqsupseteq [\![\langle C', \sigma'\rangle]\!]$.*

Exercise 23 (!)**.** *Prove that relation $\sqsupseteq$ is a preorder. We recall what this means in the following items.*

- *Reflexivity: $N \sqsupseteq N$ for all N.*
- *Transitivity: $N \sqsupseteq N'$ and $N' \sqsupseteq N''$ imply $N \sqsupseteq N''$ for all N, N', and N''.*

Exercise 24 (!)**.** *Prove theorem 4.*

5.5 Amendment

Consider again the following unprojectable choreography.

$$C_{\mathsf{unproj}} \triangleq (\mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ \mathsf{p}.\mathtt{true} \rightarrow \mathsf{q}.x; \mathbf{0}\ \mathsf{else}\ \mathbf{0})\,; \mathbf{0} \tag{5.12}$$

Now that we have selections, we can easily fix it by adding more selections until we obtain a projectable choreography.

$$(\mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ \mathsf{p} \rightarrow \mathsf{q}[\mathrm{L}]; \mathsf{p}.\mathtt{true} \rightarrow \mathsf{q}.x; \mathbf{0}\ \mathsf{else}\ \mathsf{p} \rightarrow \mathsf{q}[\mathrm{R}]; \mathbf{0})\,; \mathbf{0} \tag{5.13}$$

This manual intervention suggests a general principle: if we have an unprojectable choreography, we can try to obtain a projectable one by adding selections inside of conditionals. More precisely, if we add selections with different labels from the process that makes a conditional to the other processes that need to behave differently in the two branches but do not know which branch they are in, then we should obtain a projectable choreography. This intuition is formalised in the following definition, which gives us a mechanical method to "amend" a choreography and obtain a projectable version.

Definition 8 (Amendment [Cruz-Filipe and Montesi, 2016])**.** *Let C be a choreography. The transformation $\mathsf{Amend}(C)$ repeatedly applies the following procedure until no longer possible, starting from the innermost subterms in C. For each conditional subterm $\mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$ in C, let $\{\mathsf{r}_1, \ldots, \mathsf{r}_n\} \subseteq (\mathsf{procs}(C_1) \cup \mathsf{procs}(C_2))$ be the largest set not containing p such that $[\![C_1]\!]_{\mathsf{r}_i} \sqcup [\![C_2]\!]_{\mathsf{r}_i}$ is undefined for all $i \in [1, n]$; then, the subterm $\mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$ is replaced with:*

$$\mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ \mathsf{p} \rightarrow \mathsf{r}_1[\mathrm{L}]; \cdots ; \mathsf{p} \rightarrow \mathsf{r}_n[\mathrm{L}]; C_1\ \mathsf{else}\ \mathsf{p} \rightarrow \mathsf{r}_1[\mathrm{R}]; \cdots ; \mathsf{p} \rightarrow \mathsf{r}_n[\mathrm{R}]; C_2.$$

As an example, the result of $\mathsf{Amend}(C_{\mathsf{unproj}})$ (where C_{unproj} is defined in eq. (5.12)) is the choreography in eq. (5.13).

Amendment is a complete procedure—it is defined for every choreography C. It also guarantees projectability, as we desired.

Proposition 3. *Let C be a choreography and $\mathsf{Amend}(C) = C'$. Then, for all σ, $[\![\langle C', \sigma\rangle]\!]$ is defined.*

Exercise 25. *Prove proposition 3.*

Exercise 26. *Write the result of $\mathsf{Amend}(C)$, where C is the following choreography.*

$$(\mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ \mathsf{p}.x \rightarrow \mathsf{q}.y; \mathbf{0}\ \mathsf{else}\ \mathsf{p}.x \rightarrow \mathsf{r}.z; \mathbf{0})\,; \mathbf{0}$$

Exercise 27. *Write the result of* $\mathsf{Amend}(C)$*, where* C *is the following choreography.*

$$(\mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ \mathsf{p}.x \ \texttt{->}\ \mathsf{q}.y; \mathbf{0}\ \mathsf{else}\ \mathsf{p}.(x+1)\ \texttt{->}\ \mathsf{r}.z; \mathsf{p}.(x+2)\ \texttt{->}\ \mathsf{q}.y; \mathbf{0})\,; \mathbf{0}$$

5.6 Smart Projection

An alternative to amending choreographies is to change the definition of EPP, such that the auxiliary selections are inserted automatically by the projection procedure. The modification is pretty simple: when we project a conditional, we project a selection towards all other processes involved in the conditional from the process that evaluates the guard, and we project a corresponding branching for all such other processes. Here is the modification:

$$\begin{array}{l}
[\![\mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2; C]\!]_{\mathsf{r}} = \\
\qquad \begin{cases}
(\mathsf{if}\ f\ \mathsf{then}\ B_1; [\![C_1]\!]_{\mathsf{r}}\ \mathsf{else}\ B_2; [\![C_2]\!]_{\mathsf{r}})\,; [\![C]\!]_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{p} \\
B; [\![C]\!]_{\mathsf{r}} \quad \text{if } \mathsf{r} \neq \mathsf{p} \text{ and } B = [\![C_1]\!]_{\mathsf{r}} \sqcup [\![C_2]\!]_{\mathsf{r}} & \\
\mathsf{p}\,\&\{\mathrm{L} : [\![C_1]\!]_{\mathsf{r}}, \mathrm{R} : [\![C_2]\!]_{\mathsf{r}}\}; [\![C]\!]_{\mathsf{r}} & \text{otherwise}
\end{cases} . \\
\text{where } B_1 = \mathsf{r}_1 \oplus \mathrm{L}; \cdots ; \mathsf{r}_n \oplus \mathrm{L} \text{ and } B_2 = \mathsf{r}_1 \oplus \mathrm{R}; \cdots ; \mathsf{r}_n \oplus \mathrm{R} \\
\qquad \text{such that } [\![C_1]\!]_{\mathsf{s}} \sqcup [\![C_2]\!]_{\mathsf{s}} \text{ undefined for all } s \in \{r_1, \ldots, r_n\}
\end{array}$$

Let $\star\,[\![C]\!]$ be EPP with the rule for projecting conditionals replaced by the rule above. (Notice that we adopt this rule only in this section, and do not use smart EPP in the other sections.)

Proposition 4 (Harmony of amendment and smart projection)**.** *Let* C *be a choreography and* σ *be a global memory state. Then,* $[\![\langle \mathsf{Amend}(C), \sigma\rangle]\!] = \star\,[\![\langle C, \sigma\rangle]\!]$.

Exercise 28 (!)**.** *Prove proposition 4.*

Chapter 6

Recursion

All the choreographies that we have seen so far have finite behaviour, in the sense that all their reduction chains have finite length. Even more, all choreographies in the models that we explored so far must terminate. Let us formalise this property.

Definition 9 (Termination)**.** *We say that* $\langle C, \sigma \rangle$ *must terminate if: for all* C' *and* σ' *such that* $\langle C, \sigma \rangle \to^* \langle C', \sigma' \rangle$*, there exists* σ'' *such that* $\langle C', \sigma' \rangle \to^* \langle \mathbf{0}, \sigma'' \rangle$*.*

We say that a choreography C *must terminate if* $\langle C, \sigma \rangle$ *must terminate for all* σ*.*

In other words, a choreography must terminate if its execution will necessarily end in $\mathbf{0}$.

We are now going to extend our choreography model to allow for recursive behaviour—"repetitions" of communication structures. In this new model, not all choreographies necessarily terminate, as we are going to see.

6.1 Models for recursive choreographies and processes

6.1.1 Choreographies

Syntax We extend the syntax our choreography model with two ingredients. The new syntax is displayed in fig. 6.1.

First, we add a set of procedure definitions, ranged over by $\mathscr{C}$. A set of procedure definitions is a (possibly empty) set of definitions like $X(\tilde{\mathsf{p}}) = C$, read "procedure X has parameters $\tilde{\mathsf{p}}$ and body C". We assume that all procedures defined in a set $\mathscr{C}$ have distinct names. This means that we

$$
\begin{aligned}
C &::= I; C \mid \mathbf{0} \\
I &::= \mathsf{p}.f \text{ -> } \mathsf{q}.g \mid \mathsf{p} \text{ -> } \mathsf{q}[l] \mid \mathsf{p}.f \mid \mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \mid X(\tilde{\mathsf{p}}) \mid \mathbf{0} \\
\mathscr{C} &::= X(\tilde{\mathsf{p}}) = C, \mathscr{C} \mid \emptyset
\end{aligned}
$$

Figure 6.1: Recursive choreographies, syntax.

can abstract from the order in which procedures are defined, and thus write $X(\tilde{\mathsf{p}}) = C \in \mathscr{C}$ to state that the definition $X(\tilde{\mathsf{p}}) = C$ is in $\mathscr{C}$. The notation $\tilde{\mathsf{p}}$ is a shortcut for a sequence $\mathsf{p}_1, \ldots, \mathsf{p}_n$ for some n (a list of process names). We assume that all process names in a list of parameters $\tilde{\mathsf{p}}$ are distinct (they are all different).

Second, we add a new primitive to choreographies for invoking procedures, namely $X(\tilde{\mathsf{p}})$, read "run procedure X with arguments $\tilde{\mathsf{p}}$". Also here, we assume that the list of arguments $\tilde{\mathsf{p}}$ contains distinct process names.

Example 12. *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*.*

```
$S(\mathsf{c},\mathsf{s}) =$    if s.$(n > 0)$ then
                                     s -> c[NEXT];
                                     s.$next$ -> c.$recvNext$;
                                     s.$dec(n)$;
                                     $S(\mathsf{c},\mathsf{s})$;
                                     $\mathbf{0}$
                                 else
                                     s -> c[END];
                                     $\mathbf{0}$
                  ;$\mathbf{0}$
```

Procedure S is recursive. The guard of the conditional checks that variable n at s *contains a value bigger than* 0 *(the number zero). If so,* s *informs the client* c *that it will receive a packet. Function $next$ returns the current packet to send, and function $recvNext$ stores it appropriately in some data structure (e.g., an array) at the client. Then, we decrement n by* 1 *at the server* s *and we recursively invoke S to send the remaining packets. When we have finished the packets to send ($n \leq 0$),* s *sends the label* END *to* c *to inform it that the procedure will terminate.*

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

$$\mathsf{c}.\mathit{filename} \text{ -> } \mathsf{s}.x;\ \mathsf{s}.\mathit{buildChunks};\ S(\mathsf{c},\mathsf{s});\ \mathbf{0}$$

Here, the client sends the $filename$ of the file it wishes to download to s*. Then* s *uses function buildChunks, which we assume sets up the right index n and the chunks for next used in procedure S. Finally, we invoke S to perform the streaming.*

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

$$\begin{aligned}
\mathsf{procs}\,(I;C) &= \mathsf{procs}(I) \cup \mathsf{procs}(C)\\
\mathsf{procs}(\mathbf{0}) &= \emptyset\\
\mathsf{procs}\,(\mathsf{p}.f \text{ -> } \mathsf{q}.g) &= \{\mathsf{p},\mathsf{q}\}\\
\mathsf{procs}\,(\mathsf{p} \text{ -> } \mathsf{q}[l]) &= \{\mathsf{p},\mathsf{q}\}\\
\mathsf{procs}\,(\mathsf{p}.f) &= \{\mathsf{p}\}\\
\mathsf{procs}\,(\mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2) &= \{\mathsf{p}\} \cup \mathsf{procs}(C_1) \cup \mathsf{procs}(C_2)\\
\mathsf{procs}(X(\tilde{\mathsf{p}})) &= \ \{\tilde{\mathsf{p}}\}
\end{aligned}$$

In the remainder, whenever we consider a procedure definition $X(\tilde{\mathsf{p}}) = C$, we assume that $\mathsf{procs}(C) = \{\tilde{\mathsf{p}}\}$, meaning that the process names used in the body of the procedure are exactly those declared as parameters.

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 $\mathscr{C}$ that we are considering. We thus denote reductions as $\langle C,\sigma\rangle \to_{\mathscr{C}} \langle C',\sigma'\rangle$, read "the configuration $\langle C,\sigma\rangle$ reduces to $\langle C',\sigma'\rangle$ assuming that procedures are defined as in $\mathscr{C}$".

Remark 1. *An alternative notation for reductions that consider procedure definitions could be $\langle C,\sigma,\mathscr{C}\rangle \to \langle C',\sigma',\mathscr{C}'\rangle$. However, this notation makes it look like $\mathscr{C}$ is part of the "state of the system", just like C (the program to run next) and σ (the memory of processes). The state of a system can typically change during execution. Thus the question become: is $\mathscr{C}$ ever going to change?*

In most programming systems, the definitions of procedures never do, and this will be the case also in our model. Therefore it makes more sense to put $\mathscr{C}$ out of the notation for the state of the system, and write $\langle C,\sigma\rangle \to_{\mathscr{C}} \langle C',\sigma'\rangle$ as we do.

$$\frac{f(\sigma(\mathsf{p}))\downarrow v \qquad g(\sigma(\mathsf{q}),v)\downarrow u}{\langle \mathsf{p}.f \;\texttt{->}\; \mathsf{q}.g; C, \sigma\rangle \to_{\mathscr{C}} \langle C, \sigma[\mathsf{q} \mapsto u]\rangle}\ \textsc{Com} \qquad \frac{}{\langle \mathsf{p} \;\texttt{->}\; \mathsf{q}[l]; C, \sigma\rangle \to_{\mathscr{C}} \langle C, \sigma\rangle}\ \textsc{Sel}$$

$$\frac{f(\sigma(\mathsf{p}))\downarrow v}{\langle \mathsf{p}.f; C, \sigma\rangle \to_{\mathscr{C}} \langle C, \sigma[\mathsf{p} \mapsto v]\rangle}\ \textsc{Local}$$

$$\frac{i = 1 \text{ if } f(\sigma(\mathsf{p}))\downarrow \texttt{true},\ i = 2 \text{ otherwise}}{\langle \mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2; C, \sigma\rangle \to_{\mathscr{C}} \langle C_i; C, \sigma\rangle}\ \textsc{Cond}$$

$$\frac{C \preceq_{\mathscr{C}} C_1 \qquad \langle C_1, \sigma\rangle \to_{\mathscr{C}} \langle C_2, \sigma'\rangle \qquad C_2 \preceq_{\mathscr{C}} C'}{\langle C, \sigma\rangle \to_{\mathscr{C}} \langle C', \sigma'\rangle}\ \textsc{Struct}$$

Figure 6.2: Recursive choreographies, semantics.

There are, however, examples of programming models where the code of procedures might change at runtime. In these settings, the alternative notation makes more sense, since $\mathscr{C}$ might change. A model where the definitions of choreographies can evolve at runtime was presented by Dalla Preda et al. [2017].

The new rules defining the semantics of choreographies are displayed in figs. 6.2 and 6.3. Structural precongruence is also annotated with $\mathscr{C}$ now, since it requires to know the definitions of procedures. The only new rule is UNFOLD, in fig. 6.3. It states that an invocation of procedure X can be replaced by the body of the procedure, replacing the parameters of the procedure with the arguments passed by the invocation site—$\tilde{\mathsf{p}}/\tilde{\mathsf{q}}$ is a shortcut for replacing each occurrence of q_i in C with the corresponding p_i (assuming that $\tilde{\mathsf{p}}$ and $\tilde{\mathsf{q}}$ have the same length).

Example 13. *Let $\mathscr{C}$ be the singleton set containing the definition of procedure S in example 12. Let C be the choreography in the same example:*

$$C \triangleq \mathsf{c}.\mathit{filename} \;\texttt{->}\; \mathsf{s}.x;\ \ \mathsf{s}.\mathit{buildChunks};\ \ S(\mathsf{c},\mathsf{s});\ \ \mathbf{0}$$

. By rule UNFOLD*, we can replace the invocation of S inside of C with the*

$$\frac{\mathsf{procs}(I) \mathrel{\#} \mathsf{procs}(I')}{I; I' \quad \equiv_{\mathscr{C}} \quad I'; I}\ \text{I-I}$$

$$\frac{\mathsf{p} \notin \mathsf{procs}(I)}{I; \mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \quad \equiv_{\mathscr{C}} \quad \mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ (I; C_1)\ \mathsf{else}\ (I; C_2)}\ \text{I-}\textsc{Cond}$$

$$\frac{\mathsf{p} \notin \mathsf{procs}(I) \qquad I \neq \mathbf{0}}{\mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2; I \quad \equiv_{\mathscr{C}} \quad \mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ (C_1; I)\ \mathsf{else}\ (C_2; I)}\ \textsc{Cond}\text{-I}$$

$$\frac{}{\mathbf{0}; C \preceq_{\mathscr{C}} C}\ \text{GC}\textsc{Nil}$$

$$\frac{X(\tilde{\mathsf{q}}) = C \ \in \ \mathscr{C}}{X(\tilde{\mathsf{p}}) \preceq_{\mathscr{C}} C[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}]}\ \textsc{Unfold}$$

Figure 6.3: Recursive choreographies, structural precongruence.

body of S, as follows.

$$\begin{array}{lllll}
C \preceq_D & \mathsf{c}.filename \text{ -> } \mathsf{s}.x; & & & \\
 & \quad \mathsf{s}.buildChunks; & & & \\
 & \quad \mathsf{if}\ \mathsf{s}.(n > 0)\ \mathsf{then} & & & \\
 & & & \mathsf{s} \text{ -> } \mathsf{c}[\textsc{next}]; & \\
 & & & \mathsf{s}.next \text{ -> } \mathsf{c}.recvNext; & \\
 & & & \mathsf{s}.dec(n); & \\
 & & & S(\mathsf{c}, \mathsf{s}); & \\
 & & & \mathbf{0} & \\
 & & \mathsf{else} & & \\
 & & & \mathsf{s} \text{ -> } \mathsf{c}[\textsc{end}]; & \\
 & & & \mathbf{0} & \\
 & ;\mathbf{0} & & &
\end{array}$$

Example 14. *We can now write choreographies that can always continue running, i.e., they never terminate.*

The following procedure implements a ping-pong communication structure between two processes p *and* q*, which take turns in pinging each other. Notice how, when we invoke the procedure, we invert the order of processes to make them take turns.*

$$PingPong(\mathsf{p}, \mathsf{q}) = \ \mathsf{p}.ping \text{ -> } \mathsf{q}.x;\ \mathsf{q}.pong \text{ -> } \mathsf{p}.y;\ PingPong(\mathsf{q}, \mathsf{p});\ \mathbf{0}$$

$$
\begin{aligned}
N &::= \mathsf{p} \triangleright_v B \mid N \,|\, N \mid \mathbf{0}\\
B &::= \mathsf{p}!f; B \mid \mathsf{p}?f; B \mid \mathsf{p} \oplus l; B \mid \mathsf{p}\,\&\{l_i : B_i\}_{i\in I}; B\\
&\quad\; \mid \mathsf{if}\ f\ \mathsf{then}\ B_1\ \mathsf{else}\ B_2; B \mid \mathbf{0}; B \mid X(\tilde{\mathsf{p}}) \mid \mathbf{0}\\
\mathscr{B} &::= X(\tilde{\mathsf{p}}) = B, \mathscr{B} \mid \emptyset
\end{aligned}
$$

Figure 6.4: Recursive processes, syntax.

The turns are evident by looking at the unfolding of PingPong:

$$
\begin{array}{l}
\mathsf{p}.\mathit{ping}\ \texttt{->}\ \mathsf{q}.x;\\
\mathsf{q}.\mathit{pong}\ \texttt{->}\ \mathsf{p}.y;\\
\mathit{PingPong}(\mathsf{q},\mathsf{p});\\
\mathbf{0}
\end{array}
\quad \preceq_D \quad
\begin{array}{l}
\mathsf{p}.\mathit{ping}\ \texttt{->}\ \mathsf{q}.x;\\
\mathsf{q}.\mathit{pong}\ \texttt{->}\ \mathsf{p}.y;\\
\mathsf{q}.\mathit{ping}\ \texttt{->}\ \mathsf{p}.x;\\
\mathsf{p}.\mathit{pong}\ \texttt{->}\ \mathsf{q}.y;\\
\mathit{PingPong}(\mathsf{p},\mathsf{q});\\
\mathbf{0};\\
\mathbf{0}
\end{array}
$$

.

Exercise 29. *Prove that the choreography* $\mathit{PingPong}(\mathsf{p},\mathsf{q}); \mathbf{0}$ *never terminates.*

6.1.2 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. 6.4 to 6.6. The additions are the new syntax term for invoking procedures and an unfolding rule for structural precongruence.

6.1.3 Projection

Consider again our procedure *PingPong* and a choreography C_{pp} that invokes it.

$$
\begin{aligned}
\mathscr{C}_{pp} &\triangleq \mathit{PingPong}(\mathsf{p},\mathsf{q}) = \mathsf{p}.\mathit{ping}\ \texttt{->}\ \mathsf{q}.x;\ \mathsf{q}.\mathit{pong}\ \texttt{->}\ \mathsf{p}.y;\ \mathit{PingPong}(\mathsf{q},\mathsf{p});\ \mathbf{0}\\
C_{pp} &\triangleq \mathit{PingPong}(\mathsf{p},\mathsf{q}); \mathbf{0}
\end{aligned}
$$

How should we project C_{pp}? The idea is that procedure *PingPong* should be translated to two procedures on the process level: one that describes the behaviour of p inside of its body—let us call this procedure $\mathit{PingPong}_{\mathsf{p}}$—and another that describes the behaviour of q—let us call this procedure

$$\frac{f(v)\downarrow v' \qquad g(u,v')\downarrow u'}{\mathsf{p}\triangleright_v \mathsf{q}!f;B \ \mid\ \mathsf{q}\triangleright_u \mathsf{p}?g;B' \quad \to_{\mathscr{B}} \quad \mathsf{p}\triangleright_v B \ \mid\ \mathsf{q}\triangleright_{u'} B'}\ \textsc{Com}$$

$$\frac{j\in I}{\mathsf{p}\triangleright_v \mathsf{q}\oplus l_j;B \ \mid\ \mathsf{q}\triangleright_u \mathsf{p}\,\&\{l_i : B_i\}_{i\in I};B' \quad \to_{\mathscr{B}} \quad \mathsf{p}\triangleright_v B \ \mid\ \mathsf{q}\triangleright_u B_j;B'}\ \textsc{Sel}$$

$$\frac{i=1 \text{ if } f(v)\downarrow \texttt{true},\ i=2 \text{ otherwise}}{\mathsf{p}\triangleright_v (\mathsf{if}\ f\ \mathsf{then}\ B_1\ \mathsf{else}\ B_2)\ ;B \to_{\mathscr{B}} \mathsf{p}\triangleright_v B_i;B}\ \textsc{Cond}$$

$$\frac{N_1 \to_{\mathscr{B}} N_1'}{N_1 \mid N_2 \ \to_{\mathscr{B}}\ N_1' \mid N_2}\ \textsc{Par} \qquad \frac{N \preceq_{\mathscr{B}} N_1 \quad N_1 \to_{\mathscr{B}} N_2 \quad N_2 \preceq_{\mathscr{B}} N'}{N \to_{\mathscr{B}} N'}\ \textsc{Struct}$$

Figure 6.5: Recursive processes, semantics.

$$\frac{}{(N_1 \mid N_2) \mid N_3 \ \equiv_{\mathscr{B}}\ N_1 \mid (N_2 \mid N_3)}\ \text{PA} \qquad \frac{}{\mathbf{0};B \preceq_{\mathscr{B}} B}\ \text{GCB}$$

$$\frac{}{N_1 \mid N_2 \ \equiv_{\mathscr{B}}\ N_2 \mid N_1}\ \text{PC} \qquad \frac{}{N \mid \mathbf{0} \ \preceq_{\mathscr{B}}\ N}\ \text{GCN} \qquad \frac{}{\mathsf{p}\triangleright_v \mathbf{0} \ \preceq_{\mathscr{B}}\ \mathbf{0}}\ \text{GCP}$$

$$\frac{X(\tilde{\mathsf{q}}) = B \ \in\ \mathscr{B}}{X(\tilde{\mathsf{p}}) \preceq_{\mathscr{B}} B[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}]}\ \textsc{Unfold}$$

Figure 6.6: Recursive processes, structural precongruence.

$PingPong_{\mathsf{q}}$. Thus we obtain the following set of procedure definitions for process behaviours.

$$\mathscr{B}_{pp} \triangleq \begin{array}{rl} PingPong_{\mathsf{p}}(\mathsf{q}) = & \mathsf{q}!ping; \\ & \mathsf{q}?y; \\ & PingPong_{\mathsf{q}}(\mathsf{q}) \\ & \mathbf{0}, \\ PingPong_{\mathsf{q}}(\mathsf{p}) = & \mathsf{p}?x; \\ & \mathsf{p}!pong; \\ & PingPong_{\mathsf{p}}(\mathsf{p}) \\ & \mathbf{0} \end{array}$$

We can then project C_{pp} as follows, for some σ.

$$[\![\langle C_{pp}, \sigma\rangle]\!] = \quad \mathsf{p} \triangleright_{\sigma(\mathsf{p})} PingPong_{\mathsf{p}}(\mathsf{q}); \mathbf{0} \ \mid \ \mathsf{p} \triangleright_{\sigma(\mathsf{p})} PingPong_{\mathsf{q}}(\mathsf{p}); \mathbf{0}$$

Exercise 30. *Write down the first four reductions of C_{pp} and its projection above. Do they correspond?*

Our example shows that now we need to be able to project both choreographies and procedure definitions.

For projecting choreographies, the definition is the same as before (but we will have to update behaviour projection, $[\![C]\!]_{\mathsf{p}}$).

Definition 10 (EndPoint Projection (EPP)). *The EPP of a configuration $\langle C, \sigma\rangle$, denoted $[\![\langle C, \sigma\rangle]\!]$, is defined as:*

$$[\![\langle C, \sigma\rangle]\!] = \prod_{\mathsf{p} \in \mathsf{procs}(C)} \mathsf{p} \triangleright_{\sigma(\mathsf{p})} [\![C]\!]_{\mathsf{p}}$$

.

To update behaviour projection to deal with procedure invocations, we first update merging. This is done by adding the following rule.

$$X(\tilde{\mathsf{p}}); C \sqcup X(\tilde{\mathsf{p}}); C' \ = \ X(\tilde{\mathsf{p}}); (C \sqcup C')$$

Then, we simply need to define the projection of a procedure invocation.

$$[\![X(\tilde{\mathsf{p}}); C]\!]_{\mathsf{r}} = \begin{cases} X_i(\tilde{\mathsf{p}} \setminus \mathsf{p}_i); [\![C]\!]_{\mathsf{r}} & \text{if } \tilde{\mathsf{p}} = \mathsf{p}_1, \ldots, \mathsf{p}_n \text{ and } \mathsf{r} = \mathsf{p}_i \text{ and } 1 \leq i \leq n \\ [\![C]\!]_{\mathsf{r}} & \text{otherwise} \end{cases}$$

The notation $\tilde{\mathsf{p}} \setminus \mathsf{r}$ means “the list obtained by removing r from $\tilde{\mathsf{p}}$”. Observe that the procedure invocation that we output for a process involved in the choreographic invocation is for X_{r}, which we assume implements the behaviour for process r in X.

Now that we know how to project a recursive choreography, we can write the definition of projection for procedure definitions.

Definition 11. *The EPP of a set of procedure definitions $\mathscr{C}$, denoted $[\![\mathscr{C}]\!]$, is defined as:*

$$[\![\mathscr{C}]\!] = \left\{ X_i(\tilde{\mathsf{p}} \setminus \mathsf{p}_i) = [\![C]\!]_{\mathsf{p}} \;\middle|\; \begin{array}{l} X(\tilde{\mathsf{p}}) = C \in \mathscr{C} \text{ and} \\ \tilde{\mathsf{p}} = \mathsf{p}_1, \ldots, \mathsf{p}_n \text{ and } 1 \leq i \leq n \end{array} \right\}$$

.

Exercise 31. *Write the EPP of the procedure and choreography in example 12.*

We can formulate an operational correspondence for recursive choreographies and their EPP as follows.

Theorem 5 (Operational Correspondence)**.** *Let $[\![\langle C, \sigma\rangle]\!] = N$ and $[\![\mathscr{C}]\!] = \mathscr{B}$. Then,*

Completeness *If $\langle C, \sigma\rangle \to_{\mathscr{C}} \langle C', \sigma'\rangle$ for some C' and σ', then there exists N' such that $N \to_{\mathscr{B}} N'$ and $N' \preceq_{\mathscr{B}} \sqsupseteq [\![\langle C', \sigma'\rangle]\!]$.*

Soundness *If $N \to_{\mathscr{B}} N'$ for some N', then there exists C' and σ' such that $\langle C, \sigma\rangle \to_{\mathscr{C}} \langle C', \sigma'\rangle$ and $N' \preceq_{\mathscr{B}} \sqsupseteq [\![\langle C', \sigma'\rangle]\!]$.*

Appendix A

Solution to selected exercises

We give the solutions to some selected exercises. Some solutions are given in full detail, to serve as examples of exposition. All solutions should still provide enough information for the reader to figure out the remaining parts.

Solution of exercise 1. 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:

- $\mathsf{conn}(A, B)$ provable in fig. 1.1 implies $\mathsf{conn}(A, B)$ provable in fig. 1.2;
- $\mathsf{path}(A, B)$ provable in fig. 1.1 implies $\mathsf{path}(A, B, n)$ for some n provable in fig. 1.2.

The proof is by induction on the structure of the derivation of $\mathsf{conn}(A, B)$ or $\mathsf{path}(A, B)$. We proceed by cases on the last applied rule of the derivation.

Base cases (axioms) If the last applied rule is an axiom, then the proof is valid also in the other system.

Case Sym The derivation has this shape:

$$\dfrac{\dfrac{\mathcal{P}}{\mathsf{conn}(B, A)}}{\mathsf{conn}(A, B)}\ \textsc{Sym}.$$

By induction hypothesis, we know that there exists $\mathcal{P}'$ in the other system such that:

$$\dfrac{\mathcal{P}'}{\mathsf{conn}(B, A)}.$$

The thesis follows by applying rule Sym.

$$\cfrac{\cfrac{\mathcal{P}'}{\mathsf{conn}(B,A)}}{\mathsf{conn}(A,B)}\ \textsc{Sym}.$$

Case Dir The derivation has this shape:

$$\cfrac{\cfrac{\mathcal{P}}{\mathsf{conn}(A,B)}}{\mathsf{path}(A,B)}\ \textsc{Dir}.$$

By induction hypothesis, we know that there exists $\mathcal{P}'$ in the other system such that:

$$\frac{\mathcal{P}'}{\mathsf{conn}(A,B)}.$$

The thesis follows by applying rule DirW.

$$\cfrac{\cfrac{\mathcal{P}'}{\mathsf{conn}(A,B)}}{\mathsf{path}(A,B,1)}\ \textsc{DirW}.$$

Case Trans The derivation has this shape:

$$\cfrac{\cfrac{\mathcal{P}}{\mathsf{path}(A,B)} \quad \cfrac{\mathcal{Q}}{\mathsf{path}(B,C)}}{\mathsf{path}(A,C)}\ \textsc{Trans}.$$

By induction hypothesis on $\mathcal{P}$ and by induction hypothesis on $\mathcal{Q}$, we know that there exist n and m, $\mathcal{P}'$ and $\mathcal{Q}'$ in the other system such that:

$$\frac{\mathcal{P}'}{\mathsf{path}(A,B,n)} \qquad \frac{\mathcal{Q}'}{\mathsf{path}(A,B,m)}.$$

The thesis follows by applying rule TransW.

$$\cfrac{\cfrac{\mathcal{P}'}{\mathsf{path}(A,B,n)} \quad \cfrac{\mathcal{Q}'}{\mathsf{path}(A,B,m)}}{\mathsf{path}(A,B,n+m)}\ \textsc{TransW}.$$

■

List of Figures

List of Notations

B A process behaviour. 29

C A choreography. 24

N A network. 29

$\mathscr{C}$ A context. 32

p A process name (or process identifier). 24

$\mathcal{P}$ A derivation in an inference system. 16

Index

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.

BPMN. Business Process Model and Notation. `http://www.omg.org/spec/BPMN/2.0/`, 2011.

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. A core model for choreographic programming. In Olga Kouchnarenko and Ramtin Khosravi, editors, *FACS*, volume 10231 of *LNCS*. Springer, 2016. doi: 10.1007/978-3-319-57666-4_3.

Luís Cruz-Filipe and Fabrizio Montesi. Procedural choreographic programming. In *FORTE*, LNCS. Springer, 2017.

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.

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.

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 Deniélou, 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.

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-c ontext-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 and David Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.