A choregraphy-based approach to distributed applications

May 3, 2019

Contents

List of Figures

Preface

These notes provide an introduction to the use of choreographies for the development of distributed application. Before diving into choreographies, the notes review some basic notions of software architectures in the context of distributed applications. The material is suitable for readers with a basic background in computer science and software engineering. Concepts are first introduced informally (using simple variants of UML sequence diagrams) and then more rigorously. This presentation style hopefully will allow readers not versed in mathematical jargon to grasp the main concepts.

For obvious reasons, these notes can only cover a portion of the spectrum of available approaches. For instance, alternative models based on behavioural types are not considered. Those approaches have their own merits and indeed offer solid basis for practical software engineering of communication-centric software. However, they cannot easily be taught in a single module as they require a substantial background on subjects like types, type checking, and behavioural relations (such as (bi-)simulations or trace equivalences). I hope that you will find this module interesting and I would like to encourage you to report any inaccuracy (for which I apologise in advance). I thank the who engaged with the module and provided useful insights with their questions, observations, and discussions that also contributed to polish the material presente here. In particular, I am grateful to Liu Zhao for having developed a nice web interface for **ChorGram** in his final project, and Shashidar Ette for his engagement and for his questions, suggestions, and the careful proof-reading of the notes.

Acronyms

API	Application Program Interface
BPMN	Business Process Modelling Notation
CFSM	Communicating finite-state machines
DbC	Design-by-contract
G-choreography	Global choreography
MSC	Message Sequence Chart
SOA	Service-Oriented Architecture
SOC	Service-Oriented Computing
UML	Unified Modelling Language

Part I

Overview

Chapter 1

Context and motivations

Distribution, interactions, and all that

Distribution has become a prominent requirement of software. The satisfaction of this requirement leads to a continuous increase in the complexity of software. After arguing that such complexity cannot be tamed with traditional approaches, this chapter surveys the relations between software development and some new methods recently introduced. Finally, we will make some remarks on distributed coordination in general and about orchestration and choreography more specifically.

1 Distributed applications

The combined effect of the availability of devices capable of communicating and computing and of the widespread connectivity[1] is determining a transition of applications. Big vendors, as well as small software companies, have to satisfy the appetite of users who want their data and applications "always handy". Applications and devices are, and will more and more be, interconnected in order to assist people. Software will increasingly interact and be ubiquitous in the "cyberspace" to help people in their day-to-day activities, to automatise

[1] which is characterising the urban areas of the most advanced countries

business-to-business interactions, to operate in and interact in complex cyber-physical environments, move through "smart cities", etc.

What is fostering this transition? Programming languages featuring high concurrency and message-passing seem to start becoming the favourite choice. For instance, Google's GoLang (golang.org) advocates channel-based communication among application components. Facebook is increasingly using Erlang and many companies are increasingly using Elixir (see e.g. https://blog.codeship.com/comparing-elixir-go), which is highly compatible with Erlang since it based on its virtual machine. All these languages aim to support high concurrency which is achieved with *asynchronous* communications relying on queues. Other directions are to equip existing languages with libraries providing communication-based features; eminently, the Akka (cf. https://akka.io/) initiative for Scala and Java.

Besides the linguistic angle (and the accompaining shift of paradim), software development methodologies and platforms (like https://www.ibm.com/cloud) are being developed based on "new" principles. In fact, recently concepts like *microservices*, *choreography*, *reactive systems* start to be commonplace in industrial contexts [5]. Modern platforms such as those featuring *cloud computing* and the *internet of things* can be seen as catalysing factors of this disruptive transformation. The magnitude of this change is marked by what is envisaged as the rise of a new enabler: the *API economy* [3, 13], which promises deep transformations of the software industry. The API (after application programming interfaces) space represents the latest evolution in interoperability for the current industrial context and emphasises the realisation of clear descriptions as the utmost form of documentation. The idea behind the API economy is to build new services by composing other services available online and documented through their API. This composition of services yields new services once their API is published. This is currently supported by platforms like Mashery (https://www.mashery.com) or Kong (https://getkong.org) to help developers to combine and manage APIs (theirs and/or third partys). The API economy appears as a transformative wave in software development, deployment, and consumption.

This rich technological environment supports fast development of applications. Basically, software is becoming ubiquitous and distributed applications are becoming predominant in almost all sectors; besides being used in commercial applications, distributed software is progressively adopted to assist people in any other aspect of their life. For instance, e-health or e-government applications are more and more crucial in modern societies. Practically, every application could be conceived as operating in an ecosystem of distributed applications that can be composed together to form new services. For instance, a personal trainer app could interact with a restaurant booking system to find a diner satisfying the users dietary constraints; another example could be a service for patients travelling with a monitoring device whose software alerts a nearby hospital in case of an emergency. This makes *distribution* one of the key requirements of modern software.

2 New challenges

It is widely accepted that distributed systems and applications are not easy to design, implement, verify, deploy, and maintain. Not only there are intrinsic issues due to the underlying computational model (concurrency, physical distribution, fragility of the communication networks and their low-level protocols, etc.), but also applications have to be engineered within a strange schism. In fact, applications are typically made of computational components that, on the one hand, have to collaborate with each other (in order to fulfill some requirements) while, on the other hand, may have to compete for resources and/or conflicting goals. Non-functional requirements make the problem even more intricate. For instance, different administrative domains may impose different access policies to their services or resources. Or they may provide different (levels of) services to their users. Of course, the picture gets worse when factoring in malicious components/users that may try to spoil or take advantage of non robust applications. For such (and other) reasons, developers are required to carefully design their applications so that unintended behaviours do not happen at runtime.

A key factor to consider when developing distributed applications is scalability. In this respect, traditional technology for service composition manifests its limitations. In fact, a widespread approach to compose services is the combination of (variants of) remote procedure call and *orchestration*; such architectures do not scale smoothly to very large applications. More recently, different approaches have been considered. The most successful ones adopt technology based on *message-passing* as coordination primitive: distributed components coordinate with each other by exchanging messages. Paradigmatic languages embedding this type of primitives are Erlang, Elixir, and GoLang, while others such as Java and Scala feature libraries supporting the *actor model* [1].

The linguistic shift *per se* would not be enough to attain scalability; message exchanges must be disciplined and efficient. Orchestration does not seem to be the best coordination paradigm for message-passing programming since it typically requires a larger volume of messages than *choreography*[2]. We advocate choreographies as a proper way to specify and document the composition of distributed APIs. In fact, the composition of distributed services through their APIs is promising, but it suffers of the weakness best described by the following quotation [31]

[2] Choreography will be extensively discussed in Part II; an intuitive description of choreographies is given in Section 5

APIs aren't difficult to create, but they can be difficult to learn, says analyst Larry Perlstein at Stamford, Conn.-based Gartner Group Inc. Application developers and vendors must constantly be thinking about whether their APIs will be understandable to future developers. "An API is useless unless you document it".

The above problem has been observed in paradigms like SOC or cloud computing and seems exacerbated in *microservice architectures* [25] that take to an extreme the idea of loosely coupled software developments interacting by messages passing. Microservices are revolutionising distributed software moving away from traditional *monolithic* applications, built around databases that need to be shared by the different components. Although microservices enable scalability and reduced time-to-market, developers highlight some drawbacks. For instance, the fragmentation of software in microservices imposes the use of well-documented interfaces to avoid catastrophic effects when changing some of the components in an application.

3 "I have this terrible feeling of deja vu"

The issues briefly discussed above call for design and implementation methodologies based on rigorous grounds to precisely specify (and verify) applications according to

- the assumptions relied upon and
- the guarantees an application should provide to its partners.

Let us look at a very simple example to illustrate those issues.

3.1 From message-passing programs...

Consider the following simple Erlang program[3]:

```
start() ->
  Pong_PID = spawn(example, pong, []),                          % the server starts
  spawn(example, ping, [3, Pong_PID]).                          % the client starts
```

which implements a ping-pong protocol where a “ping” client and a “pong” server interact according to the following two functions:

```
     ping(0, Pong_PID) ->                                       % first clause of the client
       Pong_PID ! finished,
 3     io:format("ping finished~n", []);

     ping(N, Pong_PID) ->                                       % second clause of the client
       Pong_PID ! {ping, self()},
 6     receive
         pong ->
           io:format("Ping received pong~n", [])
 9     end,
       ping(N - 1, Pong_PID).

     pong() ->                                                  % clause of the server
12     receive
         finished ->
           io:format("Pong finished~n", []);
15       {ping, Ping_PID} ->
           io:format("Pong received ping~n", []),
           Ping_PID ! pong,
18         pong()
       end.
```

Function `ping` consists of two clauses (starting at lines 1 and 3 respectively); the first clause is executed when `ping` is invoked with the first parameter set to 0 otherwise the second clause is executed. In the first case, a process running `ping` sends the `finished` message to another process running `pong` and identified by the second parameter of `ping` and then it terminates after printing a string on the screen (line 3). In the other case, the `ping` process sends a `ping` message to the `pong` process, waits for a `pong` message from the other process and invokes itself after decrementing the first parameter and printing a string on the screen (line 8). The process running `pong` waits for either of the two messages (`finished` and `ping`) and reacts as expected by the partner process.

[3] No prior knowledge of Erlang is assumed.

We recall that Erlang adopts the actor model [1] processes communicate using "mailboxes" (in Erlang jargon), that is each process has a queue of incoming where incoming messages from other processes are kept to be then consumed and processed by the receiver.

A natural question to ask about the above program is: what is the "communication pattern" of the program? Our models will address such question. And, more crucially, we will advocate explicit mechanisms ruling the "correct" use of components such as the functions `ping` and `pong` above. For instance, is the behaviour of a program spawning a `ping` process where the first parameter is lower than 0 admissible? Or, does the following program

```
start() ->
  Pong_PID = spawn(example, pong, []),
  spawn(example, ping, [3, Pong_PID]),
  spawn(example, ping, [2, Pong_PID]).
```

"correctly" use the functions? And what does "correct" mean here?

3.2 ...to APIs & contracts

As said, sofware has been shifting from 'stand-alone' applications to mobile and dynamically composable ones. This has in turn also increased the emphasis on communication and interaction. In this context, it is crucial to define and use languages, methodologies, and tools to specify, analyse, and verify application-level protocols underlying distributed software. The typical properties of application-level protocols that are of interest extend classical protocols like deadlock- or livelock-freedom, with *progress* or *absence of message orphanage* that we will define later.

We advocate the use of *choreographies* to foster a rigorous approach to software development. In particular, we promote the idea that distributed applications should be built out of precise specification of their APIs, which use mechanisms based on message-passing to coordinate with each other. More precisely, we envisage APIs as *contracts* that precisely state the expectation they have on their execution context, and what they guarantee to other partners. In fact, application-level protocols can be thought of as contracts that stipulates the expected communication pattern or distributed components.

The notion of "contract" dates back[4] to the seminal work of Floyd [14], Dijkstra [12] and Hoare [18] who pioneered the idea of decorating software with statements that should hold true when the control reaches them. Probably, the most successful fallout with important practical repercussions of these research lines is Meyer's *design-by-contract* (DbC); in fact, DbC is nowadays an effective software development technique in object-oriented programming [26]. The idea of DbC is that the method of an object realises a precise contract between invokers and the object expressed in terms of *pre-* and *post-* conditions. For instance, in[5]

```
int keepGoing(int p)
  // pre:  p >= 0
  // post: p < keepGoing(p)
```

```
int keepGoing(int p)
// pre:  p < 0
// post: keepGoing(p) < p
```

the assertions on the left establish that if the method `keepGoing` is invoked with a positive integer `p`, it will return an integer strictly greater than `p`; instead, the assertions on the right stipulate that `keepGoing` returns a value strictly lower than `p` when the latter is (strictly) negative.

The benefits of the DbC approach in software development is undisputed. As a matter of fact, DbC allows programmers to avoid errors in their software and greatly reduces *defensive programming*, that is the need of cluttering programs with code to check if the state of the computation meets the conditions

[4] ...thence the title of this section borrowed from the famous Monty Python's "Flying Circus".

[5] We use a fictional syntax.

expected by a construct before its execution. Moreover, DbC also enables the automatic synthesis of monitors that can check the conditions stipulated in the contracts at run-time.

Remark 1 *The so called* behavioural types *(that constrain the reciprocal interactions of distributed participants) are an example of such approaches. Notably, behavioural types are at the ground of a number of projects, notably the network* BehAPI: behavioural APIs *(EU MSCA-RISE-2017 num. 778233) and* Behavioural Types for Reliable Large-Scale Software Systems *(BETTY, oc-2011-2-10054-EU COST Action). BehAPI is an international network of academic institutions and small-medium enterprises (SMEs) across Argentina, Europe, and the US; this project started in March 2018 to foster models, languages, and tools to support API-based software. BETTY has developed new foundations, programming languages, and software development methods for communication-intensive distributed systems. Our department has played and plays an important role in both projects.*

In classical DbC approaches contracts are specified by directly annotating software with assertions. Instead, application-level protocols specify high-level contracts at design level. Such difference should probably be more deeply explored as it introduces interesting questions. For instance, contracts of high-level specifications do not guarantee the correctness of actual implementations. On the other hand, code correctness does not guarantee that e.g. the protocol of an application enjoys good properties. It is therefore necessary to establish clear links between the two levels and study their complementary interplay. Although intriguing, these types of questions are not in our scope here.

Remark 2 *A possible drawback of using sophisticated contracts in distributed applications is that design and implementation become more complex. Indeed, this is what happens in the DbC approach in object-oriented programming men-*

tioned above.[6] *However, this seems to be a necessary price to pay; since the problem is complex and simplistic approaches do not provide satisfactory solutions.*

4 Distributed coordination

Among the approaches to the design of distributed coordination, *orchestration* and *choreography* are probably the most popular. They both aim to describe the distributed *workflow* of components, namely they specify how control and data exchanges coordinated in distributed applications or systems. Intuitively, orchestration yields the description of a distributed workflow from "one party's perspective" [33], whereas choreography describes the behaviour of involved parties from a "global viewpoint" [21]. In an orchestrated model, the distributed computational components coordinate with each other by interacting with a special component, the *orchestrator*, which at run time dictates how the computation evolves. In a choreographed model, the distributed components autonomously execute and interact with each other on the basis of a local control flow expected to comply with their role as specified in the "global viewpoint".

The dichotomy orchestration-choreography has been discussed in several papers (see e.g., [33, 7]) although, to the best of our knowledge, precise definitions of those concepts are still missing and only intuitive descriptions have been given so far. It is therefore worth clarifying further the intuitive descriptions of orchestration and choreography given above.

There is common consensus that the distinguishing element of a choreographic model is the specification of a so-called *global viewpoint* detailing the interactions among distributed participants and offering a "contract" about their

[6] In this respect it is interesting the reaction of students in software engineering from different universities reported in [29].

expected communication behaviour in terms of message exchanges. This intuition is best described in W3C words [21]:

Using the Web Services Choreography specification, a contract containing a global definition of the common ordering conditions and constraints under which messages are exchanged, is produced that describes, from a global viewpoint [...] observable behaviour [...]. Each party can then use the global definition to build and test solutions that conform to it. The global specification is in turn realised by combination of the resulting local systems [...]

Noteworthy, the excerpt above points out that local behaviour should be realised by conforming to the global viewpoint in a "top-down" fashion. Hence, the relations among the global and local specifications are paramount. These aspects are addressed in [22] through an analysis of the relations between the *interaction-oriented* choreographies (i.e., global specifications expressed as interactions) and the *process-oriented* ones (i.e., the local specifications expressed as process algebra terms). A different "bottom-up" approach has been recently introduced in [24] that synthesise choreographies from local specifications. This makes choreography models more flexible (for instance, choreographies have been exploited in [23] as a contract model for service composition). Note that – adapting the terminology of [22] – we use communicating machines as *automata-oriented* choreography, as in [24].

The concept of orchestration is more controversial. We adopt a widely accepted notion of orchestration [11, 2, 32] nicely described by Ross-Talbot's as [35]:

In the case of orchestration we use a service to broker the interactions between the services including the human agents and in the case of choreog-

raphy we define the expected observable interactions between the services as peers as opposed to mandating any form of brokering.

This description envisages the distributed coordination of services as mediated by a distinguished participant that – besides acting as provider of some functionalities – regulates the control flow by exchanging messages with partner services according to their exposed communication interface.

In Peltz's words [33]:

Orchestration refers to an executable business process that can interact with both internal and external Web services. The interactions occur at the message level. They include business logic and task execution order, and they can span applications and organisations to define a long-lived, transactional, multi-step process model. [...] Orchestration always represents control from one party's perspective.

The "executable process" mentioned by Peltz is called *orchestrator* and specifies the workflow from the "one party's perspective" describing the interactions with other available services, so to yield a new composed service. This description accounts for a composition model enabling developers to combine existing and independently developed services. The orchestrator then "glues" them together in order to realise a new service, as done for instance in Orc [28]. This is a remarkable aspect since the services combined by an orchestrator are not supposed to have been specifically designed for the service provided by the orchestrator and can in fact be (re)used by other orchestrators for realising different services. Notice that this approach differs from the "bottom-up" one of [24], because synthesised choreographies do not correspond to executable orchestrators.

Other authors consider orchestration as the description of message exchanges among participants from the single participants' viewpoint *without assuming the presence of an orchestrator*. For instance, in [34, 36] the local specifications of a choreography are considered the orchestration model of the choreography itself. This acceptation could be considered too lax because any distributed application consists of parties that exchange information (no matter if realised with channel communication, remote method invocation, etc.). Considering each local specification of a choreography as an orchestration may obscure the matter; rather local specifications are tailored to (and dependent of) the corresponding party of the choreography instead of begin independently designed.

Part II

From Global Specifications...

Chapter 2

What is a choreography?

Think global, ...

Before venturing in the technical and formal details of our artefacts, this chapter provides an intuitive description of choreography. We will see that the ideas underpinning our theories are pretty practical: choreographic approaches were indeed advocated by software industry.

This chapter adopts a semi-formal approach to introduce the most important concepts of choreographies we will focus on and discusses the quest for more precision.

5 An intuitive account

Let us consider again the excerpt from [21] we already saw on page 23 (bold text is mine)

"Using the Web Services Choreography specification, a contract containing a **global definition** *of the common ordering conditions and constraints under which messages are exchanged, is produced that describes, from a* **global viewpoint** *[...] observable behaviour of all the parties involved. Each party can then use the global definition to* **build and test** *solu-*

tions that conform to it. The global specification is in turn realised by combination of the resulting **local systems** *[...]"*

The first part of the quotation above envisages a choreography as a global specification regulating the exchange of messages. The last part of the quotation above yields two distinctive elements of choreographies. The first element is the fact that the global definition can be used by each party to build its component. The second element is that conformance checks can also be done locally using the global contract.

Remark 3 *The first element is referred to as the* projectability *of the choreography, that is the possibility to determine the behaviour of all the participants of the choreography from the global specification. In fact the description of W3C conceptualises two* views*, a* global *and a* local *one, which enable the relations represented by the following diagram:*

$$\begin{tikzcd} \text{Global view} \arrow[r, "\text{projection}"] & \text{Local view} & \text{System Local} \arrow[l, "\text{comply}"'] \end{tikzcd} \tag{1}$$

where 'projection' is an operation producing the local view from the global one and 'comply' verifies that the behaviour of each components adhere with the one of the corresponding local view. The second element is referred to as realisability *of the choreography, namely the property of a choreography to be correctly implemented by* distributed components*. We will see that this is not always the case.*

Before continuing, let us deviate on a terminological digression. We will use the term 'artefact' when referring to actual specifications embodying the global/local views. Such embodiments may assume various forms: types [19], programs [9], graphs and automata [24, 10], executable models [21, 4], etc. Typically, the literature uses the (overloaded) word 'model' to refer to this flora of embodiments. We prefer the word ‘artefact’ because it allows us to refer to different contexts and different abstraction levels without attaching yet another meaning to ‘model’.

Let us now go back to diagram (1); this diagram depicts a beautiful idea as it unveils the interplay between global and local artefacts and this allows us to apply some of the best principles of computer science:

Separation of concerns The *intrinsic logic* of the distributed coordination is expressed in and analysed on global artefacts, while the local artefacts refine such logic at lower levels of abstraction.

Modular software development life-cycle The W3C description above yields a distinctive element of choreographies which makes them appealing (also to practitioners). Choreographies enables independent development: components can harmoniously interact if they are proven to comply with the local view. Global and local views yield the “blueprints” of systems as a whole and of each component, respectively.
Such methodology suits industry as it allows the combination of parties developed independently (e.g., services) while hiding implementation details that typically companies do not want to reveal. Moreover, the developers of a local component can check it against the global view and have the guarantee that, if the local component is compliant with the global view, the whole application will conform to the global choreography.

Principled design A choreographic framework orbits around the following implication:

$$\textbf{if } \textit{cond}(\text{global artefact}) \textbf{ then } \textit{behave}(\textit{projection}(\text{global artefact}))$$

that is, proving that a correctness condition *cond* holds on an abstraction (the global artefacts) guarantees that the system is well behaved, provided that the local artefacts are “compiled” from the global ones via a *projection* operation that preserves behaviour.

It is important to remark that a choregraphy should guarantee that the distributed interactions happen *harmoniously*. For instance, a choreography where some of the participants terminate while others remain waiting for some messages to arrive, may not be considered a good choreography.

Remark 4 *Harmonious execution has to be considered under the light of distributed execution based on the assumptions made on the communication model and on the fact that each participant executes distributively.*

More precisely, we require that *well-behaved* choreographies have the following properties:

Graceful termination: all the participants involved in a choreography eventually terminate, or never get stuck.

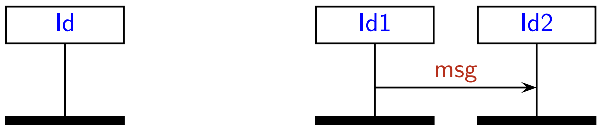
No orphan-message: all sent messages are eventually received.

No unspecified-reception: each participant should receive only expected messages.

Exercise 1 *Does the Erlang program obtained by replacing 3 with -1 in the first program of Section 3 (on page 17) gracefully terminate? Briefly justify your answer.*

Exercise 2 *Consider the* `ping` *and* `pong` *functions of Section 3 (on page 17). Is the second Erlang program of Section 3 (on page 17) well-behaved? Briefly justify your answer.*

We will first consider a (semi-)formal language to represent global specifications. Such language similar to UML's sequence diagrams or, more precisely, to *message sequence charts* (MSCs) `http://www.itu.int/rec/T-REC-Z.120`, a standard defined by the International Telecommunication Unit used to define communication protocol and equipped with visual and textual presention as well a formal semantics [17].



(a) Lifeline (b) Asynchronous call

Fig. 1: Graphical elements of sequence diagrams

6 Global views as interaction diagrams

The global view of a choreography can be represented using *sequence diagrams* [27]. A sequence diagram (see also `http://www.uml-diagrams.org/sequence-diagrams.html`) is an interaction diagram describing how some components collaborate together in order to realise some functionality. Graphically, components are represented as *headed lifelines* as depicted in Fig. 1a.

Remark 5 *It is worth emphasising that we use sequence diagrams with a different flavour than the usual one, that is as diagrams of interactions in object-oriented modelling. In fact, we will use a simplified version of sequence diagrams. More precisely, the lifelines in our sequence diagrams do not represent objects, rather* participants[7]*, namely sequential components that execute distributively. Also, interactions do not represent method calls; rather they describe message communications.*

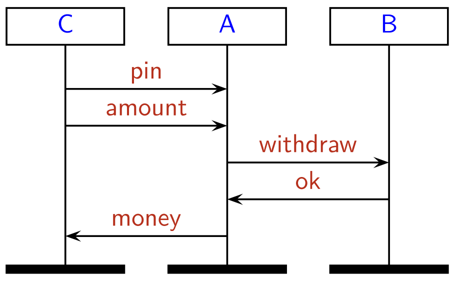
For us, a participant could be either a human being interacting with the system[8] or one of the software components of the system; we do not need to specify which of the two cases apply. In fact, by Remark 5 (and as illustrated in Fig. 1a), we may omit the class name of participants from the lifeline head of our sequence diagrams. Our basic assumption is that interaction is possible

[7] Note that a participant may be actually implemented with objects in an object-oriented language; the crucial point would be that participants do not interact with each other through method calls.

[8] Hereafter, we use the terms ‘component’ and ‘participant’ interchangeably.

through *message exchange*: this is dictated by the distributed nature of the systems that we consider. Also, each participant is supposed to be not multi-threaded and communication is asynchronous so our sequence diagrams will not have the so-called 'synchronous calls'. Moreover, we will assume that messages are exchanged on *channels* (or *ports*) even when channels are not explicitly mentioned in communication actions.

Example 6.1. A choregraphy for a simplified ATM-application can be depicted as follows:



This is a very basic choreography and we will come back to it later. ◇

There are two possible ways of interpreting the diagram in Example 6.1:

Data-flow interpretation envisages the message on the arrows of a sequence diagram as a description of what (and when) data are exchanged by participants. In this interpretation it is often immaterial how the values are exchanged (since the focus is on the flow of data). For instance, in the first interaction between C and A of the diagram in Example 6.1, pin has to be understood as C providing its personal identification number (represented by the variable pin) to A. To make it explicit the communication channel, say *c*, over which such exchange happen, one could have written pin **on** *c*. We save this notation for *labels*, some special messages that will be introduced in the following.

Communication-flow interpretation envisages the message on the arrows of a sequence diagram as identifiers of communication channels used by participants when interacting. Often the values exchanged in the interaction are not mentioned (although optionally they could be explicitly represented). For instance, in the first interaction between C and A of the choreography in Example 6.1, pin has to be understood as C sending a value on a channel called pin and A waiting on that channel to receive such value. To make it explicit the personal identification number, one could have written $\mathsf{pin}\langle pin\_number\rangle$.

Unless otherwise stated, we adopt the data-flow interpretation as we are interested in the coordination of communicating distributed participants where we assume that the identities of the participants involved in a communication identify the channel used to exchange the message. In fact, we will let $\mathsf{A\,B!m}$ denote that participant A sends participant B the message m and let $\mathsf{A\,B?m}$ denote that participant B receives message m from participant A. In other words, when channels are not explicit in sequence diagrams we use the identities of the sender and receiver to identify the channels on which a communication action happens.

Remark 6 *As we will see more precisely later, channels (implicit or not) behave as an unbound FIFO queue (that is, they preserve the order).*

7 An execution model, informally

We give now an informal account of the communication behaviour of components that we consider (a precise definition will be given later). Basically, we consider communication mechanisms abstracting standard communication primitives such as those in the TCP/IP stack or those offered by modern message-oriented middlewares.

Each lifeline has to be thought of as a component made of an *autonomous* (single) thread of execution. Lifelines show only the (observable) communication pattern of participants in terms of their send and receive operations. Between two consecutive interactions, a component may execute internal behaviour, i.e. local computation not represented in the diagram. What do we mean by “local computation”? In a distributed system/application, a computation is *local* to one of the participants, say A, if the computation does not require any interaction with other participants and it is performed (and modifies) only the local state of A. An important role among local computation is played by local *decisions* participants make and that can affect the interactions to be carried out with other participants. As we will see, these decisions must be carefully designed in order to avoid problems in the communications.

We assume that send operations are *non-blocking* while receive operations are *blocking*. In other words, the sender continues its execution even if the receiver is not ready to consume the message, while a receiver cannot continue if (one of) the expected message(s) is not available.

Example 7.1. It is easy to verify that, for Example 6.1,

1. *C sends her pin number to A*
2. *C sends A the amount of money to withdraw*
3. *A tells B that C intends to widthraw some money*
4. *B tells A that the operation is allowed*
5. *A gives C the money*

is a possible order of execution. ◇

Exercise 3 *Give a sequence of communication actions corresponding to the execution in Example 7.1.*

Remark 7 *In an actual implementation, B would perform some local computation between steps 3 and 4 (for instance, B might access a local database). Since we are interested in distributed coordination, we abstract away from such local computations and do not represent them in our models.*

Exercise 4 *Give all the sequences of communication actions which respect the order in Example 7.1.*

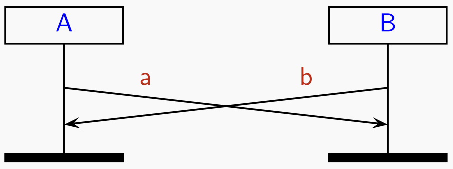
Remark 8 *You are invited to figure out why the trace*

$$\mathsf{C\,A!pin};\ \mathsf{C\,A!amount};\ \mathsf{C\,A?amount};\ \mathsf{C\,A?pin};\ \mathsf{A\,B!widthraw};$$
$$\mathsf{A\,B?widthraw};\ \mathsf{B\,A!ok};\ \mathsf{B\,A?ok};\ \mathsf{A\,C!money};\ \mathsf{A\,C?money}$$

is not a correct answer to Exercise 4.

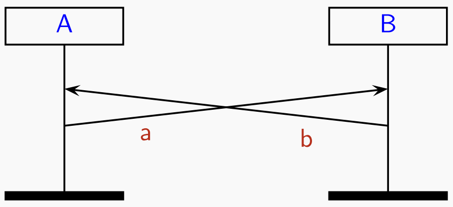
Although appealing for their intuitive clarity, sequence diagrams may look odd. The following exercise is meant to highlight this.[9]

Exercise 5 *Is the diagram below executable with asynchronous communications?*



Justify your answer.

And what about this one?



Reflect on "how time flows" and how participants execute locally.

[9] Exercise 5 is tricky because our execution model is not fully described yet; the exercise should be doable with some ingenuity, but do not be worry if you cannot solve it, you will manage after having studied the whole chapter.

7.1 From sequence diagrams to "programs"

The execution model we are drafting here naturally yields a semantics of sequence diagrams based on *execution traces* of the communication actions of each participant. Hence, the first step to take in order to find the execution traces of a sequence diagram is to determine the communication actions of participants. This is fairly simple: we just have to follow the lifeline each participant, say A, and note down the sequence of input actions and output actions performed by A. At the end we obtain a very simple program made of the sequential composition of such actions in the order determined by the lifeline of A.

Example 7.2. For the participants of the first diagram in Exercise 5

$$\mathsf{A\,B!a};\ \mathsf{B\,A?b} \qquad \textit{and} \qquad \mathsf{B\,A!b};\ \mathsf{A\,B?a}$$

are the programs of participants A and B respectively.[10] ◇

Once we have identified the programs of participants, we have to determine how these programs execute *together*! In fact, each participant A runs independently of the others (since participants are distributed), but at the same time A interacts with other participants (since participants are communicating). This will be clarified in the next section; before we have to answer another question: When is an action of (the program of) a participant A *enabled*? Basically, enabled action are the communication actions that are ready to be executed. Our assumption that participants are single-threaded requires that an action is enabled if all the actions preceding it have been executed. Hence, initially only the first action of the program of A could be enabled, once it has been executed, then the second action of the program of A could be enabled, and so on. Moreover, a moment's thought suggests that

[10] We use _; _ for sequential composition, as usual in most programming languages.

- in order to have non-blocking output actions, if the next action of the program of A is an output action then it is enabled
- likewise, in order to have blocking input actions, if the next action of the program of A is an input action, then it cannot be enabled unless the channel contains an expected message.

We are now ready to describe the execution of sequence diagrams.

7.2 Running sequence diagrams

The third and last step finally answers the question: How do we execute sequence diagram? Or, putting it more precisely, how do we describe the execution*s* of sequence diagram? As said before, we use traces for this. Intuitively, a trace describes a possible run[11] of a sequence diagram where the actions of participants alternate, in a *meaningful* order. We will now to spell out what this means to us.

Firstly, we recall the characteristics of our framework:

- channels behave as FIFO queues
- participants are single-threaded
- sending actions are non-blocking while receiving actions are blocking.

This suggests that the state of execution of a set of participants is fully determined by

1. the next communication action of each participant
2. the state of each channel (what messages are stored and in which order).

For (1) we use a sort of "program counters" that, for each principal, tell which is the next communication action to be executed. So, let us write $A_{(i)}$ when the

[11] At this point it should not come as a surprise that concurrent and distributed programs may have more than one possible execution, even on the same inputs.

"program counter" of a participant A "points" to its i-th action, that is when A has executed its first $i-1$ actions and is ready to execute the i-th one.

Example 7.3. For participant A *in Example 7.2,*

- $A_{(1)} = B\,A?b$ *(the first action has been executed and now the second one is ready to be fired)*
- $A_{(2)}$ *represents that* A *has terminated (all actions have been already executed).*

It is a simple observation that $A_{(0)}$ *is the whole program of* A*, namely the program where nothing has been executed yet, and* A *is ready to execute its first action.* ◇

For (2), the state of a channel can be simply described by the sequence of messages, say $m_1 \cdots m_k$ it contains (with m_1 the top of the queue and m_k the last message of the queue.

Then, a *configuration* maps each participant to its program and each channel to its content and we can describe the execution of a sequence diagram in terms of "evolution" of configurations starting from the *initial configuration*, namely the configuation where every participant A is mapped to the program $A_{(i)}$ and every channel is mapped to the empty sequence of messages. It is convenient to adopt a compact notation to represent configurations; we will write the sequence of programs followed by the sequence of channels (both ordered alphabetically) in angled brackets.

Example 7.4. The initial configuration of the sequence diagram in Example 7.2 and the one after A *executed its output are respectively written as*

$$\langle A_{(0)}, B_{(0)}, \overset{AB}{\boxed{\vdots}}, \overset{BA}{\boxed{\vdots}} \rangle \qquad \textit{and} \qquad \langle A_{(1)}, B_{(0)}, \overset{AB}{\boxed{\begin{array}{c} a \\ \vdots \end{array}}}, \overset{BA}{\boxed{\vdots}} \rangle$$

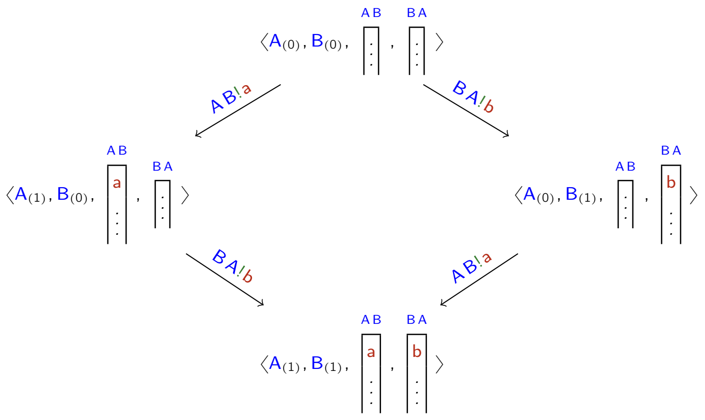
Note that we can univocally determine at which point A and B arrived in their execution and which messages are still in the channels. ⋄

The possible executions of a sequence diagram are then systematically obtained from the initial configuration as follows. We build a graph where nodes are configurations and arcs are labelled by communication actions; such graph is obtained applying the following procedure:

1. choose a configuration, say $\mathfrak{C}$, that has not been explored yet (initially, $\mathfrak{C}$ can only be the initial configuration)
2. for all the enabled action α in a program $\mathsf{A}_{(i)}$ in $\mathfrak{C}$, create a configuration $\mathfrak{C}'$ obtained from $\mathfrak{C}$ by
   - replacing $\mathsf{A}_{(i)}$ with $\mathsf{A}_{(i+1)}$
   - updating the content of the channel in α, namely:
     - if α is an output action adding the message in α at the end of the sequence of messages previously in the channel
     - if α is an input action, remove the message in α from the top of the sequence of messages previously in the channel
3. add an arc from $\mathfrak{C}$ to $\mathfrak{C}'$ labelled with α
4. mark $\mathfrak{C}$ as explored and $\mathfrak{C}'$ as not explored and repeat until all configurations are marked as explored.

In other words, starting from the initial configuration, we systematically generate configurations by firing enabled actions; in the new configurations, the participant who fired the enabled action continues to its next communication action and the channel is updated; once all the enabled action of the initial configuration have been considered, one continues the same process with another node taking care of not adding a node for a configuration that is already present in the graph. This process terminates when there are no more configurations to consider.

Example 7.5. Building the execution graph of the first diagram of Exercise 4 we get



Note that this is not the whole graph! ◇

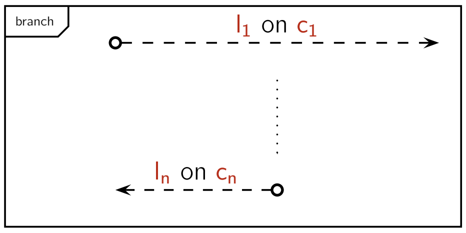
Exercise 6 *Complete the execution graph of Example 7.5.*

8 More expressiveness

As said, the choreography described in Example 6.1 (cf. page 34) is very basic. Usually, we need to add constructs to express more interesting (and precise) choreographies. For instance, one would like to be able to express what happens when C enters the wrong pin or tries to withdraw more money than the amount available on her account. We extend our language of sequence diagrams to overcome this lack of expressiveness.

8.1 Branching

Standard sequence diagrams use the so-called *alternate* construct to express conditional behaviour. For our purposes it is better to use a slightly more general construct, called *branching*, that allows us to express more than two alternative behaviours. We adopt the following notation for the branching construct:

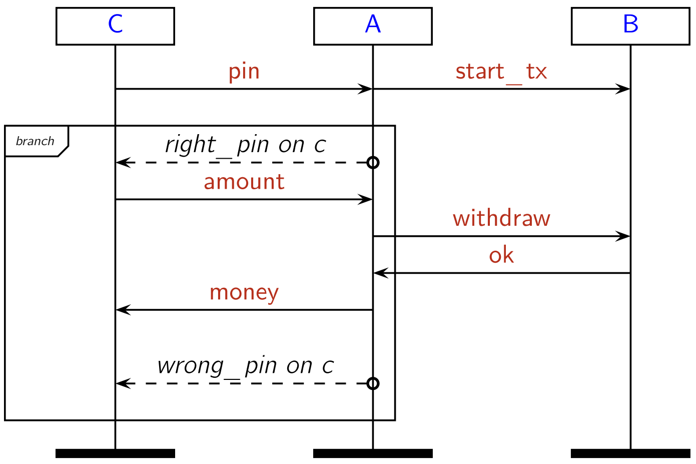


In the branching block above, *labels* $l_1, \ldots, l_n$ are sent by a participant – the one whose lifeline intersects the circle (not represented in the picture above) – on channel c to another participant – the one whose lifeline is pointed by the empty-headed arrow – to communicate how the choregraphy should proceed.

Remark 9 *The 'on c' part specifies on which channel the selection/branching is made; we may just write the labels when the channel is understood.*

We will now see that branching brings in expressiveness, but also problems. In fact, branching-blocks will be subject to some restrictions that we will spell out later. Let us first refine the choreography in Example 6.1 and show how branching caters for more expressiveness.

Example 8.1. A refined choregraphy for the simplified ATM-application can be depicted as follows:



Note that the interactions between A and B happen only if C enters a right pin number. Also, in the case that C enters a wrong pin, the choreography stops after A sends the label 'wrong' to C. ◇

Remark 10 *An important observation emerging from Example 8.1 is that the check on the pin number performed by A is an "local" computation not represented in the diagram; we will come back on this.*

8.2 A note on choice and non-determinism

Two important concepts of choreographies (and more generally in parallel and distributed computing) are the notions of *choice* and *non-determinism*. Unlike in sequential computations, the control flow in concurrent and distributed computations is not unique; in fact, by definition there may be many parts of a system that may concurrently execute. This makes non-determinism unavoidable (besides being sometimes desirable) in distributed applications. It is worth discussing in more detail the interplay between distributed choice and non-determinism may lead to problems in distributed choreographies.

8.2.1 Distributed choices

A participant A is executing an *internal computation* when no interactions with other participants are required for A to progress. At design level, very often such internal computations are abstracted away so to maintain the design simple. In a choreography this becomes evident when considering branches; as observed earlier (cf. Remark 10), the sequence diagram of a choreography simply describes the communication pattern of participants and, for instance, does not detail how the selector of a branch decides which label to send. In particular, it is crucial to establish when the choice of a participant is *external* or *internal*. For the moment internal choice an external choice can be only described informally (later, when the automata model will be introduced, we will give a precise definition): a participant of a choreography, say A, makes

an *internal choice* when its execution can proceed along two or more different directions (that is A can execute different patterns of interactions) and the decision about what direction to take is made without interacting with other participants

an *external choice* when A has several possible alternatives to continue its execution, but cannot progress autonomously and has to wait for interactions with other participants (in our framework, this means that A has to wait for inputs sent by other participants).

Example 8.2. In the branch of the choreography of Example 8.1, participant A *makes an internal choice when she decides to send either label* right_pin *or label* wrong_pin*; instead,* C *makes an external choice in that branch as he cannot progress until either of the labels is received. In other words, how* C *continues his execution depends on what interactions* A *does (hence* C *does an external choice accordingly).* ◇

8.2.2 Non-determinism

In general, non-determinism occurs in computations where at least one step of execution is not a function of the state determined in the previous step and/or of the input. Non-determinism is an abstraction to represent computations that are not entirely under the control of a program. For instance, when designing software the behaviour of users, or third-party or off-the-shelf components can be modelled with non-deterministic constructs. A close approximation of non-deterministic sequential computation can be obtained in usual programming via the generation of (pseudo)random values. For instance, one could write an 'if' statement whose guard is a randomly generated boolean value so to simulate the non-deterministic choice between the 'then' and 'else' branches.

In distributed computations non-determinism is not only an important abstraction mechanism, but also an intrinsic part of the behaviour. In fact it is practically unfeasible to make a distributed system completely deterministic. For example, it is not possible to predict the order in which requests from independent clients arrive to a server in an asynchronous setting (unless severe limitations that would degrade performances and efficiency are acceptable).

An intuitive way to think of non-determinism is by admitting that in some points of the computation several alternative computations are possible without explicitly specifying how the actual choice is made (for instance, by some internal computation or by a scheduler that establishes which alternative to execute next "arbitrarily", namely regardless the state of the computation or of the design/program).

Remark 11 *It is worth to remark that non-determinism and distributed choices are orthogonal concepts. For instance, internal choice can be deterministic or non-deterministic and similarly for external choice. As an example, the ATM in Example 8.1 would very likely make a deterministic choice (one could imagine that which branch to take is established by a simple if-statement that checks* *the validity of the pin number). Instead, the behaviour of B in the model answer of Exercise 7 could be a non-deterministic internal choice where B generates a random number representing the time to wait for a label from A; if the label does not arrive with such time, B sends a message on b″.*

8.3 Problems with branching

As anticipated, more expressiveness yields problems; to present the problems that branching-blocks bring in, we consider a few examples.

Unique selector. Consider the diagram in Fig. 2. After the user U starts

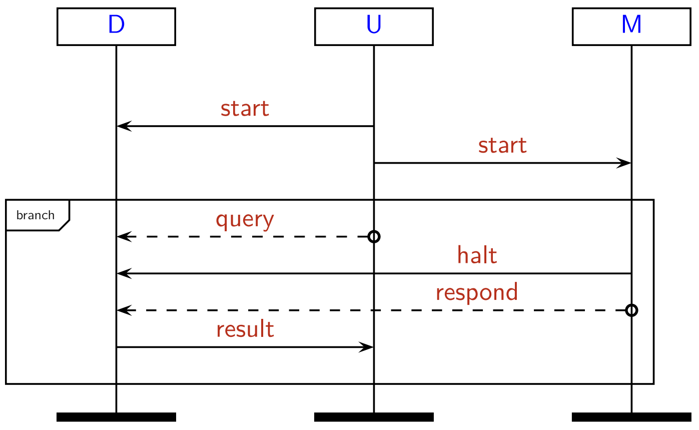


Fig. 2: The “unique selector” problem

by triggering a database D and a database manager M with messages start respectively all the participants engage in a choice so that

- if U sends label query to D then M sends message halt to D
- if instead M sends label respond to D then D sends a result result to U.

In a distributed asynchronous execution, this may lead to a deadlock. In fact, in the branch U may (locally) decide to either communicate label query to D and then terminate, or wait for D to send message result. Similarly, after receiving start, M (locally) decides whether to send halt to D and terminate, or to send message respond. If both U and M choose to send the label to D, they will both terminate hence, whatever branch D takes, the communications of halt or result will not be completed.

Exercise 7 *Give a choreography such that there is a deadlock due to the fact that in a branch labels are sent to different participants by the same participant.*

Example 8.3. Deadlock configurations can be found by building the execution graph from the sequence diagram. Consider the choreography in Fig. 3 under the communication-flow interpretation (cf. page 34). Intuitively, a deadlock

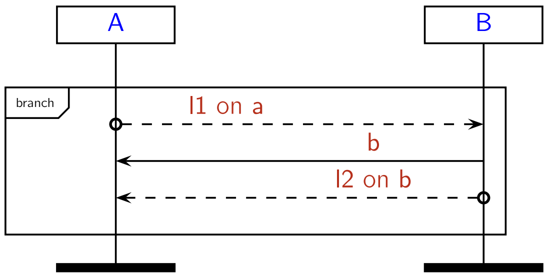
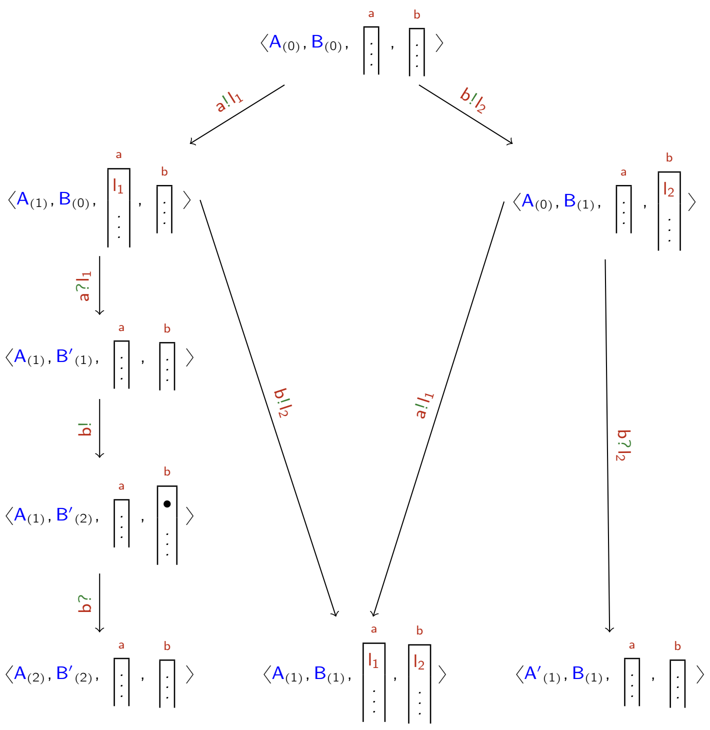


Fig. 3: Deadlock issue

occurs when both A and B decide to send their labels; in such case in fact, A will be stuck on the input from channel b which will never arrive, since B would have terminated once it has sent label l2. The execution graph of the above choreography is:



and the middle state at the bottom of the graph is a deadlock state because A_1 *is stuck on the input on* b *which is empty and the only partner which could communicate on* b *is terminated.* ◇

Determinacy. The participants involved in a branching have to be able to unambiguously determine what they have to execute after being notified a label. Take the choreography in Fig. 4. When A is notified the label l1, she can decide either to send on channel a or on channel b. However, depending on which of the branches B executes, B may be waiting on the other channel. Namely, it may happen that B chooses the first branch (and so waits on a) while A decides to send on channel B after the notification of the label. In such case, a deadlock will occur.

Un-notified participants. If a participant, say C, is not notified about which branch has been chosen, then C has to have the same behaviour in any branch. Let us reconsider the choreography in Example 8.1 (cf. page 43); there B was

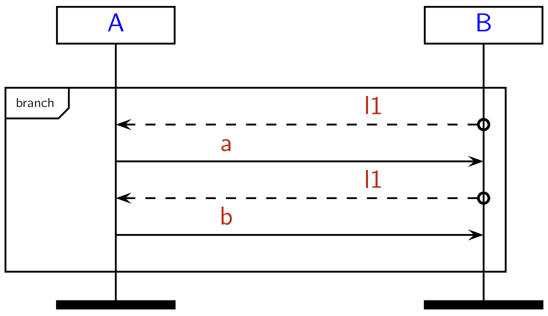
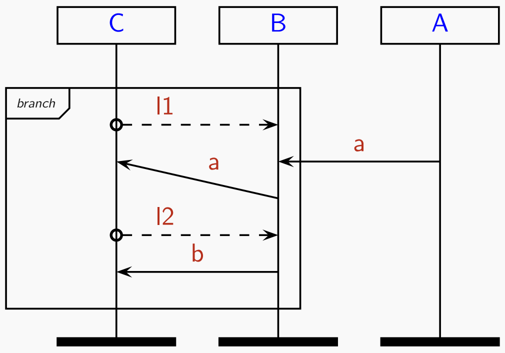


Fig. 4: Determinacy issue

not involved in the choice and still was required to be prompt to interact in case a customer C enters a right pin. This type of choreographies are not considered "good" because one of the participants (B in our example) may not terminate so causing deadlocks.

Exercise 8 *Modify the choreography in Example 8.1 (on page 43) so that the un-notified problem is solved.*

Exercise 9 *Which of the conditions of well-behaviour the second choreography of Exercise 5 (on page 37) and the choreography below violate.*



Chapter 3

A more precise framework

I have a variety of different languages at my command, different styles, different ways of talking, which do involve different parameter settings. Noam Chomsky

We start now to adopt more rigorous artefacts with precise syntax and semantics. There are many of such artefacts; we decided to consider only one of them based on a *workflow* language (for global views) and on an automata model (for local views). An advantage is that our approach is at the same time quite expressive and close to other design languages such as BPMN or UML's state machines.

9 Global views as workflow

We will now introduce a language of workflows to express global views. This language has a precise syntax and semantics. We start by looking at its syntax.

Let $\mathcal{P}$ be a set of *participants* (ranged over by A, B, etc.), $\mathcal{M}$ a set of *messages* (ranged over by m, x, etc.), and $\mathbb{Z}_0$ the set of integers. We take $\mathcal{P}$, $\mathcal{M}$, and $\mathbb{Z}_0$ pairwise disjoint. The participants of a choreography exchange messages to coordinate with each other.

Definition 9.1 (Global choreography). The set $\mathcal{G}$ of *global choreographies* (g-choreography for short) consists of the terms G derived by the grammar

$$\begin{aligned} \mathsf{G} \;\; ::= \;& \mathsf{A} \rightarrow \mathsf{B} \colon \mathsf{m} && (2) \\ \mid \;& \mathsf{G} ; \mathsf{G}' && (3) \\ \mid \;& \mathsf{G} \mid \mathsf{G}' && (4) \\ \mid \;& \big(\mathsf{G} + \mathsf{G}'\big) && (5) \\ \mid \;& {*}\mathsf{G} @ \mathsf{A} && (6) \end{aligned}$$

that satisfy the following conditions:

- in interactions $\mathsf{A} \rightarrow \mathsf{B} \colon \mathsf{m}$, we require that $\mathsf{A} \neq \mathsf{B}$
- , called *control point*, is a strictly positive integer
- any two control points occurring in different positions of a g-choreography are different (e.g., $\mathsf{A} \xrightarrow{2} \mathsf{B} \colon \mathsf{m} \mid \mathsf{C} \xrightarrow{1} \mathsf{D} \colon \mathsf{y}$ is not an element of $\mathcal{G}$)
- in iteration ${*}\mathsf{G} @ \mathsf{A}$, A is a participant of G (e.g., ${*}(\mathsf{A} \rightarrow \mathsf{B} \colon \mathsf{m}) @ \mathsf{C}$ is not an element of $\mathcal{G}$).

For $\mathsf{G} \in \mathcal{G}$, let $\mathsf{cp}\big(\mathsf{G}\big)$ denote the set of control points in G.

A g-choreography can be a simple interaction (2), the sequential (3) or parallel (4) composition of g-choreographies the choice between two g-choreographies (5), or the iteration of a choreography (6). In the global view, this is modelled with *interactions* $\mathsf{A} \rightarrow \mathsf{B} \colon \mathsf{m}$, which represent the fact that participant A sends message m to participant B, which is expected to receive m.

Control points tag interactions, non-deterministic choices and iterations, and parallel composition of g-choreographies. As we shall discuss later, sequential composition does not require control points and iterations entail a distributed choice. In the following, we may omit control points when immaterial, e.g., writing $\mathsf{G} + \mathsf{G}'$ instead of $\big(\mathsf{G} + \mathsf{G}'\big)$. Also, the values of control points are immaterial and therefore we consider equivalent g-choreographies that differ only on the values of control points; for instance:

$$\mathsf{A} \xrightarrow{2} \mathsf{B} \colon \mathsf{m} \mid \mathsf{C} \xrightarrow{3} \mathsf{D} \colon \mathsf{y} \qquad \text{and} \qquad \mathsf{A} \xrightarrow{3} \mathsf{B} \colon \mathsf{m} \mid \mathsf{C} \xrightarrow{2} \mathsf{D} \colon \mathsf{y}$$

are equivalent. Moreover, we also consider equivalent choreographies that differ just for the order of the components in non-deterministic or parallel composition; for instance,

$$(\mathsf{G} + \mathsf{G}') \qquad \text{and} \qquad (\mathsf{G}' + \mathsf{G})$$

are equivalent. Formally, $\_ + \_$ and $\_ \mid \_$ are *commutative* operations.

Exercise 10 *Give a g-choreography corresponding to the choreography in Exercise 8 (on page 50).*

Exercise 11 *Give a g-choreography equivalent, but different from the one in Exercise 10 (on page 53).*

The syntax in Definition 9.1 captures the structure of a visual language of directed graphs so that each g-choreographies G can be represented as a rooted graph with a single "enter" control point and a single "exit" one, called *source* and *sink* respectively. Fig. 5 illustrates our graphical notation and, before

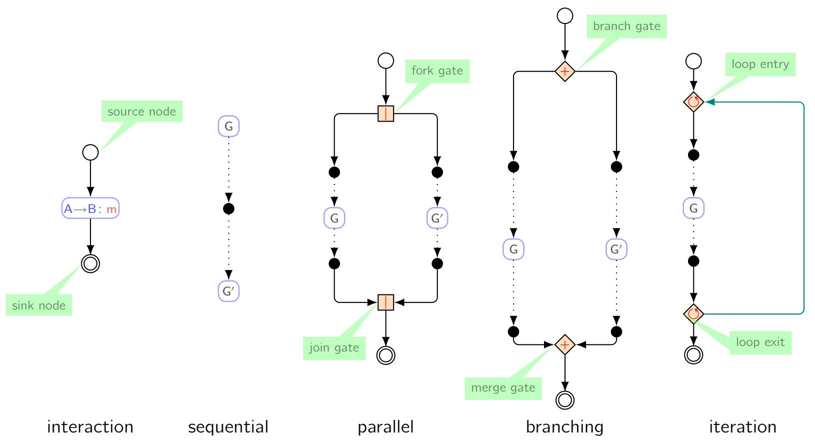
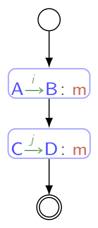
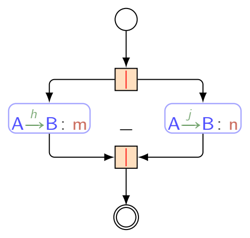


Fig. 5: A graphical notation for g-choreographies



(a) $G_{(6a)}$, a sequential composition



(b) $G_{(6b)}$, a parallel composition

Fig. 6: Examples of workflows

commenting it, a remark is worthwhile: each fork or branch gate with control point in our pictures will have a corresponding join and merge gate with control point $-$; also, negative control points will appear only in the visual notation of a global graph and not in its textual representation.

In the visual notation of Fig. 5, ○ and ◎ respectively denote the source node the sink node; other nodes are drawn as • and a dotted edge from/to a •-control points singles out the source/sink control point the edge connects to. More precisely, a dotted edge from • to a boxed G means that • is the source of G; similarly, a dotted edge from a boxed G to • means that • is the sink of G. For instance, in the graph for the sequential composition, the top-most edge identifies the sink node of G and the other edge identifies the source node of G′; intuitively, • is the control point of the sequential composition of G and G′ obtained by "coalescing" the sink control point of G with the source control point of G′. In a graph $G \in \mathcal{G}$, each control point marks either a branch or a fork gate, respectively graphically depicted as ◇ and ◫; to each branch/fork control point also corresponds a "closing" control point $-$ marking the merge/join point of the execution flow. Labels will not be depicted when immaterial.

Figs. 6a and 6b give an example of this construction for sequential and parallel composition respectively (in the latter figure we omitted the control points of interactions for readability). Fig. 6a shows why there is no need to assign a control point to sequential composition: there is no interesting event at the coalescing $\bullet$ node. Indeed, the pattern $\_ \to \bullet \to \_$ in a graph does not yield any important event to trace and in the following we will often replace it with a simple edge $\_ \to \_$).

Akin to BPMN [30] diagrams, g-choreographies yield a visual description of the distributed coordination of communicating components. In this respect, control points mark the nodes of the graph where communication and distributed work flow activities may happen. In fact, similarly to BPMN, communication activities and gate have a special standing in our visual notation.

Our diagrams resemble the ones in [10, 24] the only differences being that

- by construction, forking and branching control points have a corresponding join and merge control point $-$;
- there is a unique sink control point with a unique incoming edge (as in [10, 24], there is also a unique source control point with a unique outgoing edge).

Example 9.1. The g-choreography[12] $\mathsf{G}_{(6b)}$ *in Fig. 6b (where the control points of interactions are omitted for readability) represents a choreography where* A *sends* B *messages* m *and* n *in any order.* ◇

Exercise 12 *Give a graph corresponding to the diagram in Fig. 3 (on page 48); ignore control points.*

Exercise 13 *Add control points to the expression that you gave as a solution of Exercise 12 so that the resulting term is in* $\mathcal{G}$*.*

Exercise 14 *Give a g-choreography for the graph in Fig. 6b (on page 54).*

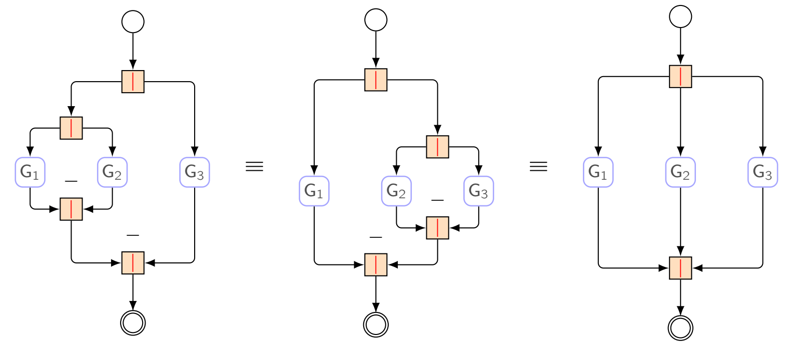
Besides being commutative, non-deterministic and parallel composition of choreographies are *associative* up to control points. Namely,

[12] We indexed our examples with the numbering of the figure they are in; therefore, we will hereafter avoid cross-referencing the figures.

$$(\mathsf{G}_1 + (\mathsf{G}_2 + \mathsf{G}_3)) \equiv ((\mathsf{G}_1 + \mathsf{G}_2) + \mathsf{G}_3)$$

$$\mathsf{G}_1 \mid \mathsf{G}_2 \mid \mathsf{G}_3 \equiv \mathsf{G}_1 \mid \mathsf{G}_2 \mid \mathsf{G}_3$$

are valid laws for all , $\in \mathbb{Z}_0$ and all $\mathsf{G}_1, \mathsf{G}_2, \mathsf{G}_3 \in \mathcal{G}$ such that each side of the equations yields a choreography. This allows us to write choices with more than two branches and, likewise, parallel compositions with more than two threads as depicted below:



where in the last graph we ignore the control point of the parallel gate.

10 Towards a precise semantics

The semantics of a choice-free g-choreography $\mathsf{G} \in \mathcal{G}$ (i.e. a g-choreography that does not contain $\_ + \_$ terms) is a partial order, which represents the causal dependencies of the communication actions specified by G.

Choices are a bit more tricky. Intuitively, the semantics of $(\mathsf{G} + \mathsf{G}')$ consists of two partial orders, one representing the causal dependencies of the communication actions of G and the other of those of G'. In the following, we will use hypergraphs as a compact representations of sets of partial orders.

10.1 Basic definitions

For us, communications take place on channels; hence, communication actions *operate* on channels. In our models, a channel is identified by (the names of the) participants involved in the communications on such channel. Formally, a channel is an element of the set

$$\mathcal{C} \quad := \quad \{(\mathsf{A}, \mathsf{B}) \in \mathcal{P} \times \mathcal{P} \mid \quad \text{and} \quad \mathsf{A} \neq \mathsf{B}\} \tag{7}$$

and we let $\mathsf{A\,B}$ denote the channel (A, B) to avoid cumbersome parenthesis. The set of *events* $\mathcal{E}$ (ranged over by e, e', ...) is defined by

$$\mathcal{E} \quad := \quad \mathcal{E}^{!} \ \cup \ \mathcal{E}^{?} \ \cup \ \mathbb{Z}_0 \tag{8}$$

where $\qquad \mathcal{E}^{!} := \mathcal{C} \times \{!\} \times \mathbb{Z}_0 \times \mathcal{M} \qquad$ and $\qquad \mathcal{E}^{?} := \mathcal{C} \times \{?\} \times \mathbb{Z}_0 \times \mathcal{M}.$

Sets $\mathcal{E}^{!}$ and $\mathcal{E}^{?}$, the output events and input events, respectively represent the "observable" effect of *sending actions* and *receiving actions*. We shorten $(\mathsf{A\,B}, !, , \mathsf{m})$ as $\mathsf{A\,B!m}$ and $(\mathsf{A\,B}, ?, , \mathsf{m})$ as $\mathsf{A\,B?m}$. As it will be clear later, events in $\mathbb{Z}_0$ correspond to control points and represent "non-observable" events, like (the execution of) a choice or a merge. In other words, events in $\mathbb{Z}_0$ correspond to local computations that do not involve interactions among components. Before continuing, we need to introduce some auxiliary notions.

We give two functions to extract the elements of communication events. The functions $\mathsf{sbj} : \mathcal{E} \to \mathcal{P}$ and $\mathsf{act} : \mathcal{E} \to \mathcal{C} \times \{?, !\} \times \mathcal{M}$, respectively called the *subject* and the *communication action* of an event are defined by

$$\begin{array}{rcl} \mathsf{sbj}\big(\mathsf{A\,B!m}\big) = \mathsf{A} & \quad \text{and} \quad & \mathsf{sbj}\big(\mathsf{A\,B?m}\big) = \mathsf{B} \\ \mathsf{act}\big(\mathsf{A\,B!m}\big) = \mathsf{A\,B!m} & \quad \text{and} \quad & \mathsf{act}\big(\mathsf{A\,B?m}\big) = \mathsf{A\,B?m} \end{array}$$

In words: the subject of an output is the sender and the subject of an input is the receiver while the communication action of an event is the underline output or input once the control point of the event is dropped. We take both sbj and act to be undefined on $\mathbb{Z}_0$ and extend cp to events, namely $\mathsf{cp}(e)$ returns the control point of event e.

Another auxiliary notion is the *dual of an event*; for an input event $\mathsf{A}\,\mathsf{B}?\mathsf{m} \in \mathcal{E}^?$, the dual event is the output event $\mathsf{A}\,\mathsf{B}!\mathsf{m} \in \mathcal{E}^?$ and, similarly, the dual of an output event $\mathsf{A}\,\mathsf{B}!\mathsf{m} \in \mathcal{E}^?$ is the input event $\mathsf{A}\,\mathsf{B}?\mathsf{m} \in \mathcal{E}^?$.

Remark 12 *Obviously, duality is a simmetric relation, namely e is dual of e' if, and only if, e' is dual of e.*

For a communication event $e \in \mathcal{E}\backslash\mathbb{Z}_0$, we write $e \in \mathsf{G}$ when there is an interaction $\mathsf{A}{\to}\mathsf{B}\colon \mathsf{m}$ in G such that $e \in \{\mathsf{A}\,\mathsf{B}!\mathsf{m}, \mathsf{A}\,\mathsf{B}?\mathsf{m}\}$, and accordingly $E \subseteq \mathsf{G}$ means that $e \in \mathsf{G}$ for all $e \in E$.

10.2 Hypergraphs and their order

Fixed a set $\mathcal{V}$ of vertexes, a (directed) *hypergraph* on $\mathcal{V}$ is a relation $H \subseteq 2^{\mathcal{V}} \times 2^{\mathcal{V}}$, namely an element in H, called *hyperarc* (or *hyperedge*), is a pair $(\!| \tilde{v}, \tilde{v}' |\!)$ that relates two sets of vertexes, the *source* $\tilde{v}$ and the *target* $\tilde{v}'$. The vertexes of our hypergraphs H are drawn from the set of events $\mathcal{E}$ and hyperedges impose an order on such events: $(\!| E, E' |\!) \in H$ represents the fact that each event in E' causally depends on all events in E. Let $\mathsf{cs}, \mathsf{ef} : 2^{\mathcal{E}} \times 2^{\mathcal{E}} \to 2^{\mathcal{E}}$ be the maps respectively returning first and second component of two related sets of vertexes, that is: if $h = (\!| E, E' |\!)$ then $\mathsf{cs}(h) = E$ are the *causes* of h and $\mathsf{ef}(h) = E'$ are its *effects*. Given $H, H' \subseteq 2^{\mathcal{E}} \times 2^{\mathcal{E}}$, define the hypergraph[13]

$$H \circ H' = \{(\!| \mathsf{cs}(h), \mathsf{ef}(h') |\!) \mid h \in H, h' \in H', \mathsf{ef}(h) \cap \mathsf{cs}(h') \neq \varnothing\}$$

[13] You are invited to check that $H \circ H'$ indeed respects our definition of hypergraph.

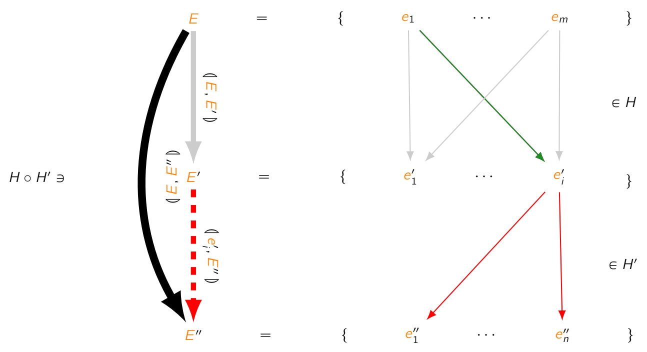


Fig. 7: A happens-before relation

That is, $H \circ H'$ is a generalised relational composition of H and H' that establishes a causal order between sets of events E and E' when there are two other non-disjoint sets of events E_1 and E_2 such that events in E happens before those in E_1 according to H and events in E_2 happens before those in E' according to H'. In other words, $H \circ H'$ is the "concatenation" of H and H'. Intuitively, the events in E cause all the events in E'' due to the dependency of the event e'_i from the events in E and the fact that e_i causes all events in E''.

Example 10.1. Fig. 7 aims to give a visual description of how operation $\_ \circ \_$ *composes hyperedges in a simple example. The composition of* $(\!| E, E' |\!) \in H$ *and* $(\!| e', E'' |\!) \in H'$ *(with* $e'_i \in E'$ *and* $(\!| e', E'' |\!) \in H'$*) yields that* $(\!| E, E'' |\!) \in E \circ E'$ *(thick arrow on the left) induced by the underlying causal relations (thin arrows on the right).* ◇

A sequence of hyperedges in a hypergraph H

$$(\!| E_0, E_1 |\!), (\!| E'_1, E_2 |\!), \ldots, (\!| E'_k, E'_{k+1} |\!) \tag{9}$$

is a *chain* if $E_i \cap E'_i \neq \emptyset$ for each $1 \leqslant i \leqslant k$ and we say that E_0 *causes* E_{k+1} or, equivalently, that E_{k+1} causally depends on E_0. An alternative definition of this relation can be also given as the reflexo-transitive closure $H^\star$ of H with respect to the composition relation $\_ \circ \_$, namely

$$H^\star \quad := \quad \bigcup_n \underbrace{H \circ \cdots \circ H}_{n\text{-times}}$$

Remark 13 *Chains of hyperedges (like the one in (9)) can be characterised in terms of the* $\_^\star$ *operation. In fact, it is easy to see that* E *causes* E' *in* H *if, and only if,* $(\!| E, E' |\!) \in H^\star$.

This construction is fundamental in the next definition.

Definition 10.1. A *happens-before relation* is a hypergraph H on $\mathcal{E}$ such that

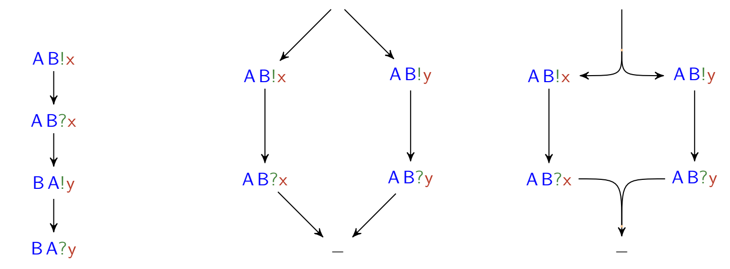
1. for all $h \in H$, $\mathsf{cs}(h) \cap \mathsf{ef}(h) = \varnothing$
2. for all $e \neq e' \in H$ we have that e is dual of e' when $\mathsf{cp}(e) = \mathsf{cp}(e')$,
3. if E causes E' in H then there are no two sets E_1 and E_1' such that $E \cap E_1 \neq \varnothing$, $E' \cap E_1' \neq \varnothing$, and E_1' causes E_1.

Basically, an hyperedge $(\!| E, E' |\!)$ in a happens-before relation relates two sets of (pairwise distinct) communication events, the *source* E and the *target* E', and represents the fact that *each event in the source happens before each of event in the target*. Therefore,

- an event cannot happen before itself (condition 1 of Definition 10.1),
- two events have the same control point only if they are dual of each other (condition 2 of Definition 10.1), and
- there cannot be circular dependencies among events (condition 3 of Definition 10.1).

A happens-before relation H induces a relation $\widehat{H} \subseteq \mathcal{E} \times \mathcal{E}$ on the vertexes of H as follows

$$\widehat{H} \quad := \quad \bigcup_{(\!| E, E' |\!) \in H} E \times E'$$



(a) $H_{(8a)}$ (b) $H_{(8b)}$ (c) $H_{(8c)}$

Fig. 8: Some happens-before relations

Basically, $\widehat{H}$ are the causal dependencies among the vertices in H and $\langle e, e' \rangle \in \widehat{H}$ when e precedes e' in H. In fact,

$$\sqsubseteq_H \quad := \quad \widehat{(H^\star)}$$

yields a partial order on the vertices of H.

In the following, we will tacitly assume that sets of events $E \subseteq \mathcal{E}$ *respect* control-points, namely that for all $e, e' \in E$, $\mathsf{act}(e)$ is the dual of $\mathsf{act}(e')$ when $\mathsf{cp}(e) = \mathsf{cp}(e')$. Also, to avoid cumbersome parenthesis, singleton sets[14] in hyperarcs are shortened by their element, e.g., we write $(\!| e, E |\!)$ instead of $(\!| \{e\}, E |\!)$.

Remark 1. Note that $H \circ H'$ may not be a happens-before relation.

Exercise 15 *Give two happens-before relations H and H' such that $H \circ H'$ is not a happens-before relation.*

Example 10.2. Some happens-before relations are depicted in Fig. 8; relation $H_{(8a)}$ and $H_{(8b)}$ contain only simple arcs, while the relation $H_{(8c)}$ contains two hyperedges:

[14] A singleton set is a set containing just one element.

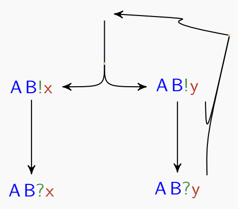
$$(\!|\ ,\ \{\mathsf{A\,B!x}, \mathsf{A\,B!y}\}\ |\!) \qquad \textit{and} \qquad (\!|\ \{\mathsf{A\,B?x}, \mathsf{A\,B?y}\}, -\ |\!)$$

Intuitively,

- $H_{(8a)}$ *establishes a total causal order from the top-most to the bottom-most event; this is the simplest possible scenario: each event happens in order from the top-most one to the bottom-most one.*
- $H_{(8b)}$ *represents a choice at control point between the left and the right branch; basically after happens either one (but not both) of its successors happens.*
- $H_{(8c)}$ *represents the parallel execution of two threads at the control point ; note that the hyperedge* $(\!|\ ,\ \{\mathsf{A\,B!x}, \mathsf{A\,B!y}\}\ |\!)$ *of* $H_{(8c)}$ *relates the event to both* $\mathsf{A\,B!x}$ *and* $\mathsf{A\,B!y}$ *"at the same time", which formalises the idea that has to happen before both its successors and* both *successors happen.*

Note the correspondence among the control points in $H_{(8b)}$ *and in* $H_{(8c)}$. ◇

Exercise 16 *Consider the following hypergraph*



and give all the reasons why the hypergraph is not a happens-before relation. Justify your answer.

10.3 Composing happens-before relations sequentially

We will see that the semantics of a choreographyG is a happens-before relation. The way such relation is built is by (inductively) "composing" the semantics of the sub-choreographies of G. This section considers the simple, yet non fully trivial, case of sequential composition of happens-before relations. In later sections we will see more complex cases.

Given a happens-before relation H and a hyperedge $h \in H$, we write $e \in h$ to shorten $e \in \mathsf{cs}(h) \cup \mathsf{ef}(h)$; also, $e \in H$ shortens $\exists h \in H : \ e \in h$. We define the *maximal* and *minimal* events of a hypergraph H respectively as

$$\max H = \{e \in H \mid \forall h \in H : \ e \notin \mathsf{cs}(h)\}$$
$$\min H = \{e \in H \mid \forall h \in H : \ e \notin \mathsf{ef}(h)\}$$

Example 10.3. With reference to Fig. 8, we have $\min H_{(8b)} = \min H_{(8c)} = \{\}$ *and* $\max H_{(8b)} = \max H_{(8c)} = \{-\}$*, while the minimal and maximal elements of* $H_{(8a)}$ *are* $\mathsf{A\,B!x}$ *and* $\mathsf{B\,A?y}$ *respectively.* ◇

Exercise 17 *Give the sets of maximal and minimal events for the graph in Exercise 16 (on page 62).*

As we will see, the semantics of a g-choreography G (when defined) will always be made of hyperedges where either the source or the target is a singleton and hyperedge of the form $(\!|\mathsf{A\,B!m}, \mathsf{A\,B?m}|\!)$ for each interaction $\mathsf{A}{\to}\mathsf{B} \colon \mathsf{m}$ in G. Also, hyperedges made only of communication events will be of importance in order to establish the existence of the semantics. In particular, we define

$$\begin{aligned}
\mathsf{fst}\, H &= \{(\!|E, E'|\!) \in H \mid (E \cup E') \cap \mathbb{Z}_0 = \varnothing \ \wedge \\
&\qquad\qquad \forall h \in H^\star : \ \mathsf{ef}(h) = E \implies \mathsf{cs}(h) \subseteq \mathbb{Z}_0\} \\
\mathsf{lst}\, H &= \{(\!|E, E'|\!) \in H \mid (E \cup E') \cap \mathbb{Z}_0 = \varnothing \ \wedge \\
&\qquad\qquad \forall h \in H^\star : \ \mathsf{cs}(h) = E' \implies \mathsf{ef}(h) \subseteq \mathbb{Z}_0\}
\end{aligned}$$

namely, $\mathsf{fst}\, H$ and $\mathsf{lst}\, H$ are the sets of the hyperedges involving the "last" and the "first" communication events of a happens-before relation H.

Example 10.4. With reference to Fig. 8, we have

$$\mathsf{fst}\, H_{(8a)} = \{(\!|\, \mathsf{A}\,\mathsf{B}!\mathsf{x}, \mathsf{A}\,\mathsf{B}?\mathsf{x} \,|\!)\} \qquad \textit{and} \qquad \mathsf{lst}\, H_{(8a)} = \{(\!|\, \mathsf{B}\,\mathsf{A}!\mathsf{y}, \mathsf{B}\,\mathsf{A}?\mathsf{y} \,|\!)\}$$

while $H_{(8b)}$ and $H_{(8c)}$ have the same set of "first" and the "last" communication actions ($\mathsf{fst}\, H_{(8b)} = \mathsf{fst}\, H_{(8c)} = \{(\!|\, \mathsf{A}\,\mathsf{B}!\mathsf{x}, \mathsf{A}\,\mathsf{B}?\mathsf{x} \,|\!), (\!|\, \mathsf{A}\,\mathsf{B}!\mathsf{y}, \mathsf{A}\,\mathsf{B}?\mathsf{y} \,|\!)\} = \mathsf{lst}\, H_{(8b)} = \mathsf{lst}\, H_{(8c)}$). ◇

We can now define $\mathsf{seq}(H, H')$, the *sequential* composition of relations H and H' on $\mathcal{E}$ as follows:

$$\begin{aligned} \mathsf{seq}(H, H') \quad := \quad & H \cup H' \cup \\ & \{(\!|\, e, e' \,|\!) \mid \exists h \in \mathsf{lst}\, H, h' \in \mathsf{fst}\, H' : \\ & e \in h \ \wedge \ e' \in h' \ \wedge \ \mathsf{sbj}(e) = \mathsf{sbj}(e')\} \end{aligned}$$

The sequential composition of two happens-before relations H and H' preserves the causal dependencies of its constituents (namely those in $H \cup H'$) and establishes dependencies between every event in $\mathsf{lst}\, H$ and every event in $\mathsf{fst}\, H'$ with the same subject.

Example 10.5. Fig. 9 depicts the sequential composition $\mathsf{seq}(H, H')$ where $H = \mathsf{A}\,\mathsf{B}!\mathsf{x} \to \mathsf{A}\,\mathsf{B}?\mathsf{x}$ and H' ranges over the happens-before relations

$$\begin{array}{ccccc} \mathsf{A}\,\mathsf{C}!\mathsf{y} & \mathsf{B}\,\mathsf{C}!\mathsf{y} & \mathsf{C}\,\mathsf{B}!\mathsf{y} & \mathsf{A}\,\mathsf{B}!\mathsf{y} & \mathsf{C}\,\mathsf{D}!\mathsf{y} \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ \mathsf{A}\,\mathsf{C}?\mathsf{y} & \mathsf{B}\,\mathsf{C}?\mathsf{y} & \mathsf{C}\,\mathsf{B}?\mathsf{y} & \mathsf{A}\,\mathsf{B}?\mathsf{y} & \mathsf{C}\,\mathsf{D}?\mathsf{y} \end{array} \tag{10}$$

Normal arrows represent the dependencies induced by the subjects and dotted arrows represent dependencies induced by the sequential composition (the meaning of stroken arrows will be explained in Section 11). Basically a causal

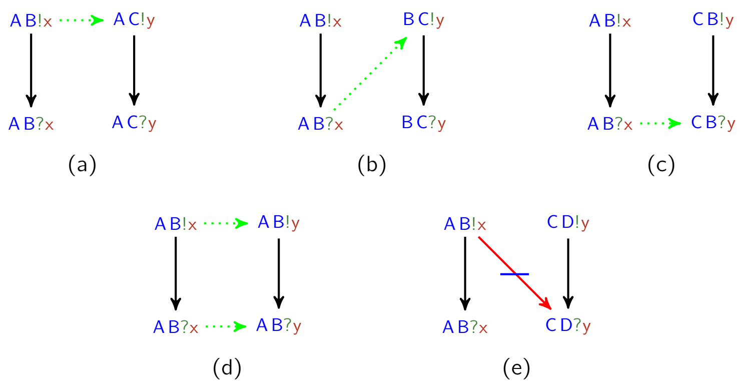


Fig. 9: Examples of sequential composition

relation is induced whenever a participant performing a (last) communication of H also starts a communication in H'. ◇

11 The semantics of g-choreographies

The semantics of g-choreography is a the partial map $[\![\_]\!]$ defined below that given a g-choreography G returns a happens-before relation capturing legal executions of the communications in G. We have $[\![G]\!]$ defined only if G is a "good" g-choreography. Following [16], function $[\![\_]\!]$ is defined as per the clauses from Eqs. (11) to (16) below.[15]

The semantics of an interaction $A \to B : m$ is straightforward:

$$[\![A \to B : m]\!] = \{(\!|\, A\,B!m, A\,B?m \,|\!)\} \tag{11}$$

namely, the send part $A\,B!m$ of the interaction must precede its receive $A\,B?m$ part. This reflects the intuition that the receiver cannot "consume" a message before the sender has made it available. Notice that the control point is neces-

[15] Here we first give a simplified version of the semantics which is less general than the one defined in [16].

sary to distinguish among events that carry the same communication actions, but happen in different parts of a choreography.

The semantics of iteration is also straightforward:

$$[\![* \mathsf{G} @ \mathsf{A}]\!] = [\![\mathsf{G}]\!] \tag{12}$$

namely, the causal order on the events of a loop is simply given by the order on the events of its body.

The semantics of sequential composition is more tricky than the previous cases as g-choreographies may not always be sequentially composed together. We define

$$[\![\mathsf{G}; \mathsf{G}']\!] = \begin{cases} \mathsf{seq}([\![\mathsf{G}]\!], [\![\mathsf{G}']\!]) & \text{if } ws(\mathsf{G}, \mathsf{G}') \\ \text{undef} & \text{otherwise} \end{cases} \tag{13}$$

where, setting $R := \mathsf{cs}\big(\mathsf{lst}\, [\![\mathsf{G}]\!]\big) \times \mathsf{ef}\big(\mathsf{fst}\, [\![\mathsf{G}']\!]\big)$,

$$ws(\mathsf{G}, \mathsf{G}') \iff \big(\widehat{\mathsf{seq}([\![\mathsf{G}]\!], [\![\mathsf{G}']\!])}\big)^{\star} \supseteq \widehat{R}$$

establishes the *well-sequencedness* condition. The semantics of sequential composition $\mathsf{G}; \mathsf{G}'$ is determined by the happens-before relations as computed by $\mathsf{seq}([\![\mathsf{G}]\!], [\![\mathsf{G}']\!])$ provided that they cover the dependencies between the last communication actions of G with the first actions of G′. This condition ensures the soundness of the composition; when such condition does not hold, then there is a participant A in G′ that cannot ascertain if all the events of G did happen before A could start.

Example 11.1. The compositions in Fig. 9 correspond to the composition of the semantics of $\mathsf{A} \rightarrow \mathsf{B} : \mathsf{x}$ *with the semantics of*

$$\mathsf{A} \xrightarrow{j} \mathsf{C} : \mathsf{y} \qquad \mathsf{B} \xrightarrow{j} \mathsf{C} : \mathsf{y} \qquad \mathsf{C} \xrightarrow{j} \mathsf{B} : \mathsf{y} \qquad \mathsf{A} \xrightarrow{j} \mathsf{B} : \mathsf{y} \qquad \mathsf{C} \xrightarrow{j} \mathsf{D} : \mathsf{y}$$

respectively. All those compositions are sound, barred the one in Fig. 9e which violates well-sequencedness; in fact, the stroken edge depicts the missing dependency required by well-sequencedness. ◇

For the parallel composition[16] $\mathsf{G} \mid \mathsf{G}'$ we have

$$[\![\mathsf{G} \mid \mathsf{G}']\!] = \begin{cases} [\![\mathsf{G}]\!] \cup [\![\mathsf{G}']\!] \cup H & \text{if } wf(\mathsf{G}, \mathsf{G}') \\ \text{undef} & \text{otherwise} \end{cases} \tag{14}$$

where, setting $H := \{(\!|\,, \mathsf{min}\,[\![\mathsf{G}]\!] \cup \mathsf{min}\,[\![\mathsf{G}']\!]\,|\!), (\!|\,\mathsf{max}\,[\![\mathsf{G}]\!] \cup \mathsf{max}\,[\![\mathsf{G}']\!], -\,|\!)\}$, the side condition

$$wf(\mathsf{G}, \mathsf{G}') \iff \mathsf{act}\big([\![\mathsf{G}]\!]\big) \cap \mathsf{act}\big([\![\mathsf{G}']\!]\big) \cap \mathcal{L}^{?} = \varnothing$$

established the *well-forkedness* condition. In words, we take the union of the dependencies of G and G′ provided that G and G′ satisfy well-forkedness, namely that the input events of G and G′ are disjoint so to avoid that the actions corresponding to the events in a thread are confused with those in other threads.

Exercise 18 *Draw the happens-before relation for the semantics of the g-choreography*

$$\mathsf{G} = \mathsf{A} \xrightarrow{1} \mathsf{B} : \mathsf{x} \mid \mathsf{A} \xrightarrow{2} \mathsf{B} : \mathsf{y} \tag{15}$$

Justify your answer.

The semantics of non-deterministic choice involves the most complex side condition:

[16] In this and the following clauses, the semantics of a composed g-choreography is defined if $[\![\mathsf{G}]\!]$ and $[\![\mathsf{G}']\!]$ are defined.

$$[\![(\mathsf{G} + \mathsf{G}')]\!] = \begin{cases} [\![\mathsf{G}]\!] \cup [\![\mathsf{G}']\!] \cup H & \text{if } wb(\mathsf{G}, \mathsf{G}') \\ \text{undef} & \text{otherwise} \end{cases} \tag{16}$$

where $H = \{ (\!|\ , \mathsf{min}\, [\![\mathsf{G}]\!]\ |\!), (\!|\ , \mathsf{min}\, [\![\mathsf{G}']\!]\ |\!), (\!| \mathsf{max}\, [\![\mathsf{G}]\!] , - |\!), (\!| \mathsf{max}\, [\![\mathsf{G}']\!] , - |\!) \}$ and the definition of the notion of *well-branchedness* ($wb(\mathsf{G}, \mathsf{G}')$) is given by:

Definition 11.1. A choice $\mathsf{G} + \mathsf{G}'$ is *well-branched*, in symbols $wb(\mathsf{G}, \mathsf{G}')$, if both the following conditions hold:

1. there is at most one *active* participant in G and G' and
2. all the other participants in G and G' are *passive*.

In Section 11.1 we define active and passive participants (cf. Definitions 11.2 and 11.3 below). Intuitively, a participant is active when it selects which branch to take next from a choice while a participant is passive when it is either not involved in the choice of it is "told" about which branch to follow. Besides the dependencies induced by G and G', $[\![(\mathsf{G} + \mathsf{G}')]\!]$ contains those making (the control point of the branch) precede all minimal events of G and G'; similarly, the maximal events of G and G' have to precede the conclusion of the choice (marked by the control point $-$). Notice that no additional dependency is required. In fact, during one instance of the g-choreography either the actions of the first branch or the actions of the second one will be performed.

11.1 Active & passive roles...easily

The semantics of a choice is defined provided that the *well-branchedness* condition holds. Such condition formalises the intuitive notion discussed in Part II which is based on the notions of active and passive participant, that respectively single out participants that do not make an internal choice, namely participants that do not select whether to execute G or G' and those participants instead that (internally) select which branch to execute.

We first give some auxiliary definitions. Given a participant $\mathsf{A} \in \mathcal{P}$ and a happens-before relation H, the A-*part of* H is the happens-before relation

$$H^{@\mathsf{A}} = \bigcup_{\llparenthesis E_1, E_2 \rrparenthesis \in H} \{\llparenthesis {E_1}^{@\mathsf{A}}, {E_2}^{@\mathsf{A}} \rrparenthesis\}$$

where, writing $\mathsf{A} \in e$ when A occurs in $\mathsf{act}(e)$,

$$\begin{aligned} E^{@\mathsf{A}} = & \{e \in E \mid \mathsf{A} \in e \ \vee \ e \in \mathbb{Z}_0\} \\ & \cup \{\mathsf{cp}(e) \mid e \in E \cap \mathcal{E}^{!} \ \wedge \ \mathsf{A} \notin e\} \\ & \cup \{-\mathsf{cp}(e) \mid e \in E \cap \mathcal{E}^{?} \ \wedge \ \mathsf{A} \notin e\} \end{aligned}$$

Intuitively, the A-part of H "focuses" on the dependencies of the events executed by A in H.

Remark 14 *We use* $\mathsf{cp}(e)$ *and* $-\mathsf{cp}(e)$ *for outputs and inputs respectively, so that different events not belonging to* A *remain distinguished.*

Definition 11.2. Let $\mathsf{G}, \mathsf{G}' \in \mathcal{G}$ such that $[\![\mathsf{G}]\!]$ and $[\![\mathsf{G}']\!]$ are defined; fix a participant $\mathsf{A} \in \mathcal{P}$ and let $E = \bigcup_{h \in \mathsf{fst}\,[\![\mathsf{G}]\!]^{@\mathsf{A}}} \{e \in h \mid \mathsf{sbj}(e) = \mathsf{A}\}$ and $E = \bigcup_{h \in \mathsf{fst}\,[\![\mathsf{G}']\!]^{@\mathsf{A}}} \{e \in h \mid \mathsf{sbj}(e) = \mathsf{A}\}$. Participant $\mathsf{A} \in \mathcal{P}$ is *passive* in $\mathsf{G} + \mathsf{G}'$ if

1. $E = \varnothing \iff E' = \varnothing$, $E \subseteq \mathcal{E}^{?}$, and $E' \subseteq \mathcal{E}^{?}$
2. $\mathsf{act}(E) \cap \mathsf{act}(E') = \varnothing$

where $\mathsf{act}(E) = \{\mathsf{act}(e) \mid e \in E\}$ and $\mathsf{act}(E') = \{\mathsf{act}(e) \mid e \in E'\}$.

Thus, the behaviour of a passive participant A in $\mathsf{G} + \mathsf{G}'$ is determined by the sets E and E' in Definition 11.2 (those are the sets of first communications of A in G and G' respectively). Either A does not perform any communication in G and in G' or any first communication that A performs in G must be an input and it must not be confused with any of the communications that A performs in G', and viceversa.

Exercise 19 *Identify the passive participant of the graph in Fig. 8b (on page 61)*

Definition 11.3. Let $\mathsf{G}, \mathsf{G}' \in \mathcal{G}$ such that $[\![\mathsf{G}]\!]$ and $[\![\mathsf{G}']\!]$ are defined; fix a participant $\mathsf{A} \in \mathcal{P}$ and let $E = \bigcup_{h \in \mathsf{fst}\, [\![\mathsf{G}]\!]^{@\mathsf{A}}} \{e \in h \mid \mathsf{sbj}(e) = \mathsf{A}\}$ and $E = \bigcup_{h \in \mathsf{fst}\, [\![\mathsf{G}']\!]^{@\mathsf{A}}} \{e \in h \mid \mathsf{sbj}(e) = \mathsf{A}\}$. Participant $\mathsf{A} \in \mathcal{P}$ is *active* in $\mathsf{G} + \mathsf{G}'$ if

1. $E \neq \varnothing$, $E' \neq \varnothing$, $E \subseteq \mathcal{E}^{!}$, and $E' \subseteq \mathcal{E}^{!}$
2. $\mathsf{act}(E) \cap \mathsf{act}(E') = \varnothing$

Thus, the first actions of participant A in $\mathsf{G} + \mathsf{G}'$ must be an output actions (at least one in each branch) and each first output in G must be different from each first output in G'.

Exercise 20 *Identify the active participant of the graph in Fig. 8c (on page 61). Justify your answer.*

Example 11.2. The following g-choreography:

$$\mathsf{G}_{(8b)} = \left(\mathsf{A} \xrightarrow{1} \mathsf{B} \colon \mathsf{x} + \mathsf{A} \xrightarrow{2} \mathsf{B} \colon \mathsf{y}\right)$$

has a defined semantics. In fact, the choice in $\mathsf{G}_{(8b)}$ is well-branched:

- *participant B is passive (receiving either $\mathsf{A}\,\mathsf{B}?\mathsf{x}$ or $\mathsf{A}\,\mathsf{B}?\mathsf{y}$ in the point of branching) and*
- *participant A is active (sending either $\mathsf{A}\,\mathsf{B}!\mathsf{x}$ or $\mathsf{A}\,\mathsf{B}!\mathsf{y}$ in the point of branching).*

The hypergraph in Fig. 8b is the happens-before relation yielding the semantics of $\mathsf{G}_{(8b)}$. ◇

Let us consider a slightly more involved example with three participants.

Example 11.3. Consider the following g-choreography:

$$\mathsf{G}_{(11.3)} = \left(\left(\mathsf{A} \xrightarrow{1} \mathsf{B} \colon \mathsf{x}; \mathsf{B} \xrightarrow{2} \mathsf{C} \colon \mathsf{y}\right) + \left(\mathsf{A} \xrightarrow{3} \mathsf{C} \colon \mathsf{z}; \mathsf{C} \xrightarrow{4} \mathsf{B} \colon \mathsf{w}\right)\right) \tag{17}$$

We now verify that the semantics of $\mathsf{G}_{(11.3)}$ *is*

$$\begin{tikzcd}
& & {} \arrow[dl] \arrow[dr] & & \\
\mathsf{A}\,\mathsf{B}?\mathsf{x} \arrow[d] & \mathsf{A}\,\mathsf{B}!\mathsf{x} \arrow[l] & & \mathsf{A}\,\mathsf{C}!\mathsf{z} \arrow[r] & \mathsf{A}\,\mathsf{C}?\mathsf{z} \arrow[d] \\
\mathsf{B}\,\mathsf{C}!\mathsf{y} \arrow[r] & \mathsf{B}\,\mathsf{C}?\mathsf{y} \arrow[dr] & & \mathsf{C}\,\mathsf{B}?\mathsf{w} \arrow[dl] & \mathsf{C}\,\mathsf{B}!\mathsf{w} \arrow[l] \\
& & \_ & &
\end{tikzcd}$$

We need to check that the choice is well-branched. Consider A *first; we have*

$$[\![\mathsf{G}_{(11.3)}]\!]^{@\mathsf{A}} = \begin{tikzcd}
& & {} \arrow[dl] \arrow[dr] & & \\
\mathsf{A}\,\mathsf{B}?\mathsf{x} \arrow[d] & \mathsf{A}\,\mathsf{B}!\mathsf{x} \arrow[l] & & \mathsf{A}\,\mathsf{C}!\mathsf{z} \arrow[r] & \mathsf{A}\,\mathsf{C}?\mathsf{z} \arrow[d] \\
{} \arrow[r] & \_ \arrow[dr] & & \_ \arrow[dl] & {} \arrow[l] \\
& & \_ & &
\end{tikzcd}$$

note that the actions carried by the first communication events of A are A B!x (in the “left” branch) and A C!z (in the “right” branch), which are different outputs

therefore A *is active while for* B *we have*

$$[\![\mathsf{G}_{(11.3)}]\!]^{@\mathsf{B}} = \begin{tikzcd}
& & {} \arrow[dl] \arrow[dr] & & \\
\mathsf{A}\,\mathsf{B}?\mathsf{x} \arrow[d] & \mathsf{A}\,\mathsf{B}!\mathsf{x} \arrow[l] & & {} \arrow[r] & \_ \arrow[d] \\
\mathsf{B}\,\mathsf{C}!\mathsf{y} \arrow[r] & \mathsf{B}\,\mathsf{C}?\mathsf{y} \arrow[dr] & & \mathsf{C}\,\mathsf{B}?\mathsf{w} \arrow[dl] & \mathsf{C}\,\mathsf{B}!\mathsf{w} \arrow[l] \\
& & \_ & &
\end{tikzcd}$$

note that the actions carried by the first communication events of B are A B?x (in the “left” branch) and C B?w (in the “right” branch), which are different outputs

and therefore B *is passive. Similarly we can verify that* C *is also passive since it receives* B C?y *in* G *and* A C?z *in* G′*).* ◇

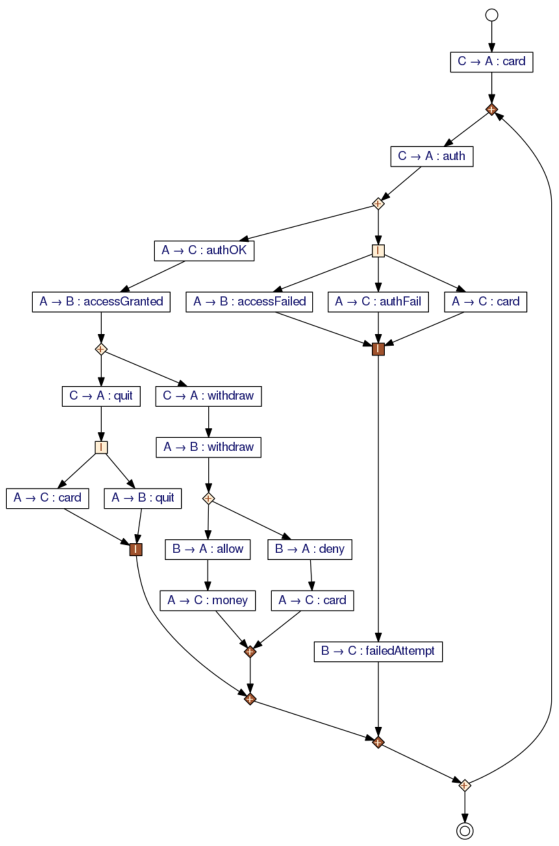
Exercise 21 *Draw the happens-before relation for the semantics of the g-choreography*

$$\mathsf{G} = (\mathsf{A} \xrightarrow{1} \mathsf{B} : \mathsf{x} + \mathsf{A} \xrightarrow{2} \mathsf{C} : \mathsf{y}) \tag{18}$$

Justify your answer.

11.2 Applying semantic equations

Let us see how to apply the equations of the semantics of g-choreographies to get happens-before relations (if any). For this we consider the following global graph:



which is a more sophisticated version[17] of the atm example considered in Example 8.1 exploiting the greater expressiveness of g-choreographies.

Remark 15 *Note that in the global graph above, several interactions run in parallel. Also, the diagram above clearly identifies the scope of branches, fork-join points, and loops.*

Since our semantics is defined according to the syntax of g-choreographies, it helps to consider the textual presentation of the diagram above.

[17] Control points are omitted for simplicity.

$$
\begin{aligned}
\mathsf{G}_{\mathsf{atm}} &= \mathsf{C}{\rightarrow}\mathsf{A}\colon \mathsf{card}\ ;\ *\mathsf{G}_{\mathsf{auth}}@\mathsf{C}\\
\mathsf{G}_{\mathsf{auth}} &= \mathsf{C}{\rightarrow}\mathsf{A}\colon \mathsf{auth}\ ;\ \left(\mathsf{G}_{\mathsf{ok}} + \mathsf{G}_{\mathsf{ko}}\right)\\
\mathsf{G}_{\mathsf{ok}} &= \mathsf{A}{\rightarrow}\mathsf{C}\colon \mathsf{authOk}; \mathsf{A}{\rightarrow}\mathsf{B}\colon \mathsf{accessGranted}; \left(\mathsf{G}_{\mathsf{wdw}} + \mathsf{G}_{\mathsf{quit}}\right)\\
\mathsf{G}_{\mathsf{wdw}} &= \mathsf{C}{\rightarrow}\mathsf{A}\colon \mathsf{withdraw}; \mathsf{A}{\rightarrow}\mathsf{B}\colon \mathsf{withdraw}; \left(\mathsf{G}_{\mathsf{alw}} + \mathsf{G}_{\mathsf{dny}}\right)\\
\mathsf{G}_{\mathsf{alw}} &= \mathsf{B}{\rightarrow}\mathsf{A}\colon \mathsf{allow}; \mathsf{A}{\rightarrow}\mathsf{C}\colon \mathsf{money}\\
\mathsf{G}_{\mathsf{dny}} &= \mathsf{B}{\rightarrow}\mathsf{A}\colon \mathsf{deny}; \mathsf{A}{\rightarrow}\mathsf{C}\colon \mathsf{card}\\
\mathsf{G}_{\mathsf{quit}} &= \mathsf{C}{\rightarrow}\mathsf{A}\colon \mathsf{quit}; \mathsf{A}{\rightarrow}\mathsf{C}\colon \mathsf{card} \mid \mathsf{A}{\rightarrow}\mathsf{B}\colon \mathsf{quit}\\
\mathsf{G}_{\mathsf{ko}} &= \mathsf{G}_{\mathsf{par}}; \mathsf{B}{\rightarrow}\mathsf{A}\colon \mathsf{failedAttempt}\\
\mathsf{G}_{\mathsf{par}} &= \mathsf{A}{\rightarrow}\mathsf{B}\colon \mathsf{accessFailed} \mid \mathsf{A}{\rightarrow}\mathsf{C}\colon \mathsf{authFail} \mid \mathsf{A}{\rightarrow}\mathsf{C}\colon \mathsf{card}
\end{aligned}
$$

Note that the syntax above ignores all the control points that are immaterial to the construction of the semantics. In particular, no control point decorates interactions since each of them has a unique occurrence in the g-choreography; observe that this immediately implies that all the fork-join g-choreographies in $\mathsf{G}_{\mathsf{atm}}$ are well-forked.

We have

$$
\begin{aligned}
[\![\mathsf{G}_{\mathsf{atm}}]\!] &= \mathsf{seq}([\![\mathsf{C}{\rightarrow}\mathsf{A}\colon \mathsf{card}]\!], [\![*\mathsf{G}_{\mathsf{auth}}@C]\!])\\
&= \mathsf{seq}\Big(\begin{array}{c}{\scriptstyle \mathsf{C\,A!card}}\\ \downarrow\\ {\scriptstyle \mathsf{C\,A?card}}\end{array}, [\![\mathsf{G}_{\mathsf{auth}}]\!]\Big) && \text{if } ws(\mathsf{C}{\rightarrow}\mathsf{A}\colon \mathsf{card}, \mathsf{G}_{\mathsf{auth}})
\end{aligned}
$$

To determine the semantics of $\mathsf{G}_{\mathsf{atm}}$ is therefore necessary to first find out what is the semantics of $\mathsf{G}_{\mathsf{auth}}$. To do this, it is convenient to start ascertaining the semantics of the last four sub-g-choreographies (since they do not depend on any other sub-g-choreography); in fact, if any of these semantics were undefined, it would be pointless to compute the rest of the semantics.

Let us start with $\mathsf{G}_{\mathsf{alw}}$:

$$[\![\mathsf{G}_{\mathsf{alw}}]\!] = \mathsf{seq}([\![\mathsf{B}{\to}\mathsf{A}\colon \mathsf{allow}]\!], [\![\mathsf{A}{\to}\mathsf{C}\colon \mathsf{money}]\!])$$

$$= \mathsf{seq}\Big(\begin{tikzcd} \mathsf{B\,A!allow} \arrow[d] \\ \mathsf{B\,A?allow} \end{tikzcd} , \begin{tikzcd} \mathsf{A\,C!money} \arrow[d] \\ \mathsf{A\,C?money} \end{tikzcd} \Big) \qquad \text{if } ws(\mathsf{B}{\to}\mathsf{A}\colon \mathsf{allow}, \mathsf{A}{\to}\mathsf{C}\colon \mathsf{money})$$

$$= \begin{tikzcd} \mathsf{B\,A!allow} \arrow[d] & \mathsf{A\,C!money} \arrow[d] \\ \mathsf{B\,A?allow} \arrow[ur] & \mathsf{A\,C?money} \end{tikzcd}$$

where in the last equation the diagonal arrow is the one added by $\mathsf{seq}(,)$, which makes the side condition $ws(\mathsf{B}{\to}\mathsf{A}\colon \mathsf{allow}, \mathsf{A}{\to}\mathsf{C}\colon \mathsf{money})$ hold. Basically, We have to verify that the order induced by the happens-before relation computed above contains

$$\mathsf{cs}\Big(\mathsf{lst}\ \begin{tikzcd} \mathsf{B\,A!allow} \arrow[d] \\ \mathsf{B\,A?allow} \end{tikzcd}\Big) \times \mathsf{ef}\Big(\mathsf{fst}\ \begin{tikzcd} \mathsf{A\,C!money} \arrow[d] \\ \mathsf{A\,C?money} \end{tikzcd}\Big) = \{(\mathsf{B\,A!allow}, \mathsf{A\,C?money})\}$$

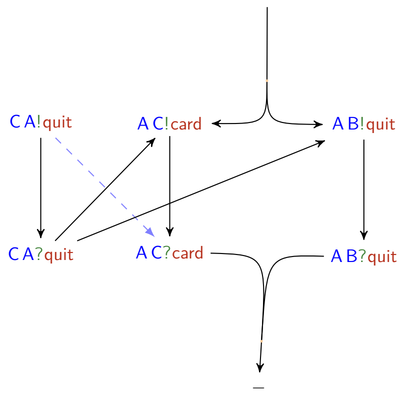
which is indeed the case. And, with a similar reasoning, we have that the semantics of $\mathsf{G}_{\mathsf{dny}}$ is

$$[\![\mathsf{G}_{\mathsf{dny}}]\!] = \begin{tikzcd} \mathsf{B\,A!deny} \arrow[d] & \mathsf{A\,C!card} \arrow[d] \\ \mathsf{B\,A?deny} \arrow[ur] & \mathsf{A\,C?card} \end{tikzcd}$$

We now turn our attention to $\mathsf{G}_{\mathsf{quit}}$:

$$[\![\mathsf{G}_{\mathsf{quit}}]\!] = \mathsf{seq}\left(\begin{smallmatrix} \mathsf{C\,A!quit} \\ \downarrow \\ \mathsf{C\,A?quit} \end{smallmatrix}, [\![\mathsf{A}\to\mathsf{C}\colon \mathsf{card} \mid \mathsf{A}\to\mathsf{B}\colon \mathsf{quit}]\!]\right)$$

$$=$$

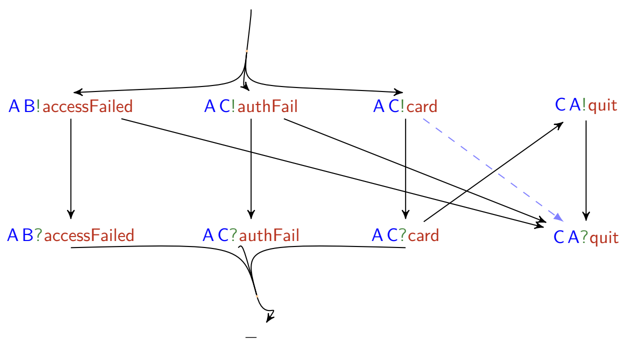


where it is easy to see that also in the case the well-sequecedness condition holds; note that the dashed arrow can be omitted because it can be obtained by transitive closure.

Provided that $ws(\mathsf{G}_{\mathsf{par}}, \mathsf{C}\to\mathsf{A}\colon \mathsf{quit})$ holds, for G_{ko} we have:

$$[\![\mathsf{G}_{\mathsf{ko}}]\!] = \mathsf{seq}\left([\![\mathsf{A}\to\mathsf{B}\colon \mathsf{accessFailed} \mid \mathsf{A}\to\mathsf{C}\colon \mathsf{authFail} \mid \mathsf{A}\to\mathsf{C}\colon \mathsf{card}]\!], \begin{smallmatrix} \mathsf{C\,A!quit} \\ \downarrow \\ \mathsf{C\,A?quit} \end{smallmatrix}\right)$$

$$=$$



where it is to verify that the well-sequecedness condition holds (since all the causes of each "last" communication edge of the parallel composition precede the effect of the input C A?quit); as before, the dashed arrow can be omitted because it can be obtained by transitive closure.

To compute the semantics of $\mathsf{G}_{\mathsf{wdw}}$ we have two options, depending on how we associate the sequential compositions (recall that the sequential composition is associative); if we associate "on the left", we have the equation below provided that the well-sequecedness condition holds:

$$\llbracket \mathsf{G}_{\mathsf{wdw}} \rrbracket = \mathsf{seq}(\llbracket \mathsf{C} \to \mathsf{A} : \mathsf{withdraw}; \mathsf{A} \to \mathsf{B} : \mathsf{withdraw} \rrbracket, \llbracket (\mathsf{G}_{\mathsf{alw}} + \mathsf{G}_{\mathsf{dny}}) \rrbracket) \tag{19}$$

Note that the first part is very similar to $\mathsf{G}_{\mathsf{dny}}$, hence:

$$\llbracket \mathsf{C} \to \mathsf{A} : \mathsf{withdraw}; \mathsf{A} \to \mathsf{B} : \mathsf{withdraw} \rrbracket = \begin{tikzcd} \mathsf{C}\,\mathsf{A}!\mathsf{withdraw} \arrow[d] & \mathsf{A}\,\mathsf{B}!\mathsf{withdraw} \arrow[d] \\ \mathsf{C}\,\mathsf{A}?\mathsf{withdraw} \arrow[ur] & \mathsf{A}\,\mathsf{B}?\mathsf{withdraw} \end{tikzcd} \tag{20}$$

For the second part, after verifying that $wb(\mathsf{G}_{\mathsf{alw}}, \mathsf{G}_{\mathsf{dny}})$ holds, we just have to add the "gluing" edges connecting minima and maxima of the branches to the branch and merge control points. Since

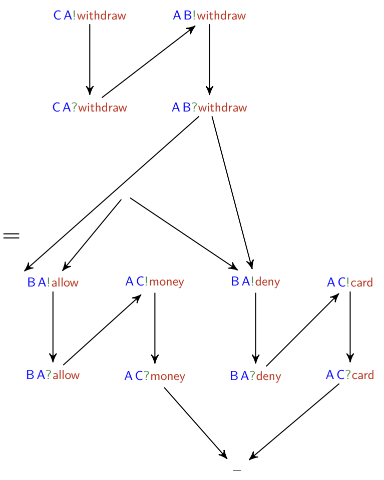
- B is active; in fact, its the first actions in both branches are different outputs
- both A and C are passive since their first actions in the two branches are different inputs

we have that the choice is well-branched, hence

$$\llbracket (\mathsf{G}_{\mathsf{alw}} + \mathsf{G}_{\mathsf{dny}}) \rrbracket = \begin{tikzcd} & {} \arrow[dl] \arrow[dr] & & \\ \mathsf{B}\,\mathsf{A}!\mathsf{allow} \arrow[d] & \mathsf{A}\,\mathsf{C}!\mathsf{money} \arrow[d] & \mathsf{B}\,\mathsf{A}!\mathsf{deny} \arrow[d] & \mathsf{A}\,\mathsf{C}!\mathsf{card} \arrow[d] \\ \mathsf{B}\,\mathsf{A}?\mathsf{allow} \arrow[ur] & \mathsf{A}\,\mathsf{C}?\mathsf{money} \arrow[d] & \mathsf{B}\,\mathsf{A}?\mathsf{deny} \arrow[ur] & \mathsf{A}\,\mathsf{C}?\mathsf{card} \arrow[dll] \\ & \_ & & \end{tikzcd} \tag{21}$$

Hence, from (19), (20), and (21) we can check that the well-sequecedness condition hold and we derive:

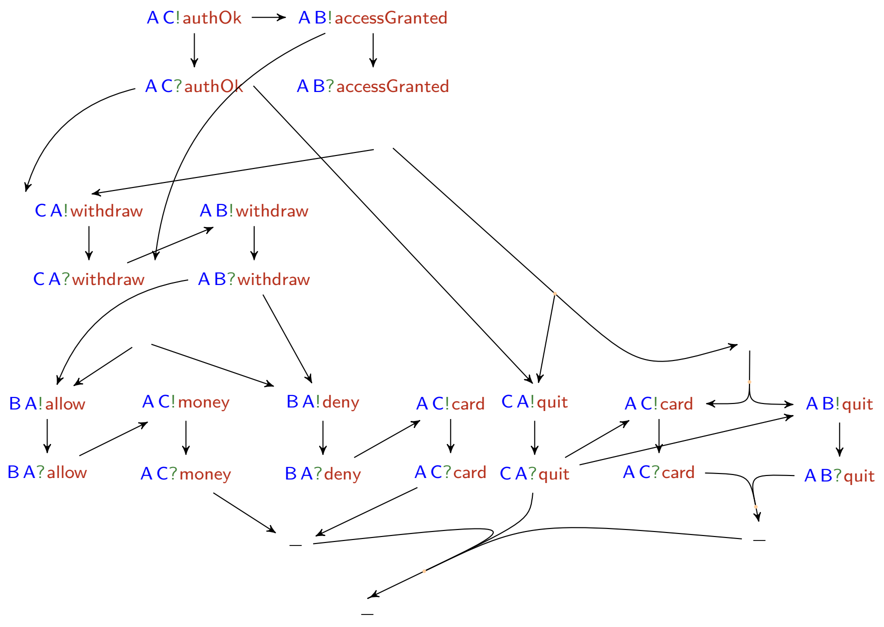
$[\![\mathsf{G}_{\mathsf{wdw}}]\!] =$



where the edges obtainable by transitive closure have been omitted.

Proceeding as done for $\mathsf{G}_{\mathsf{wdw}}$ we can compute the semantics of G_{ok} as

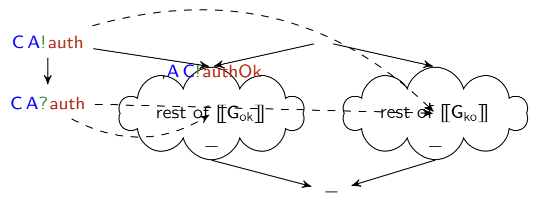
$[\![\mathsf{G}_{\mathsf{ok}}]\!] =$



∎

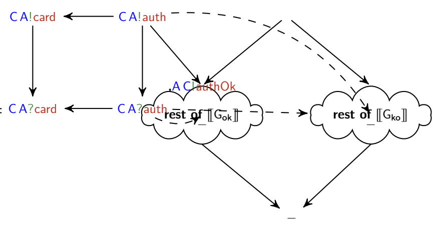
We can now reconsider

$$\begin{aligned}
[\![\mathsf{G}_{\mathsf{auth}}]\!] &= \mathsf{seq}([\![\mathsf{C}{\rightarrow}\mathsf{A}\colon \mathsf{auth}]\!], [\![(\mathsf{G}_{\mathsf{ok}} + \mathsf{G}_{\mathsf{ko}})]\!]) \\
&= \mathsf{seq}(\begin{array}{c}\scriptstyle \mathsf{C\,A!auth} \\ \downarrow \\ \scriptstyle \mathsf{C\,A?auth}\end{array}, [\![(\mathsf{G}_{\mathsf{ok}} + \mathsf{G}_{\mathsf{ko}})]\!])
\end{aligned}$$

$=$ 

where the second equation is valid if $ws(\mathsf{C}{\rightarrow}\mathsf{A}\colon \mathsf{auth}, (\mathsf{G}_{\mathsf{ok}} + \mathsf{G}_{\mathsf{ko}}))$ and the dashed edges represent the hyperedges connecting events with the same subject of their source in the semantics of G_{ok} and G_{ko}; note that again the well-sequecedness conditions is satisfied once such edges are added.

Finally, the semantics of $\mathsf{G}_{\mathsf{atm}}$ is obtained by applying once more the definition of $\mathsf{seq}(\mathsf{G}, \mathsf{G}')$ and verifying that the well-sequecedness condition holds between $\begin{array}{c}\scriptstyle \mathsf{C\,A!card} \\ \downarrow \\ \scriptstyle \mathsf{C\,A?card}\end{array}$ and $[\![\mathsf{G}_{\mathsf{auth}}]\!]$; so, we obtain

$[\![\mathsf{G}_{\mathsf{atm}}]\!] =$ 

11.3 A generalisation of well-branchedness

The notion of well-branchedness given in Section 11.1 can be made more general. In fact, such notion rules out some g-choreography that are "reasonable". We show this in the following example.

Example 11.4. Take the following variant of the g-choreography $\mathsf{G}_{(11.3)}$ *in Example 11.3:*

$$\mathsf{G}_{(11.4)} = \left(\left(\mathsf{A} \xrightarrow{1} \mathsf{B}\colon x; \mathsf{B} \xrightarrow{2} \mathsf{C}\colon y; \mathsf{B} \xrightarrow{6} \mathsf{D}\colon z\right) + \left(\mathsf{A} \xrightarrow{3} \mathsf{C}\colon z; \mathsf{C} \xrightarrow{4} \mathsf{B}\colon w; \mathsf{B} \xrightarrow{7} \mathsf{D}\colon z\right)\right)$$

We have that $[\![\mathsf{G}_{(11.4)}]\!]^{@\mathsf{A}}$ *and* $[\![\mathsf{G}_{(11.4)}]\!]^{@\mathsf{C}}$ *are very similar to those in Example 11.3. For* $[\![\mathsf{G}_{(11.4)}]\!]^{@\mathsf{B}}$ *we have:*

$$\begin{tikzcd}
 & & {} \arrow[dl] \arrow[dr] & & \\
\mathsf{A\,B?x} \arrow[d] & \mathsf{A\,B!x} \arrow[l] & & {} \arrow[r] & - \arrow[d] \\
\mathsf{B\,C!y} \arrow[r] & \mathsf{B\,C?y} \arrow[dl] & & \mathsf{C\,B?w} \arrow[dr] & \mathsf{C\,B!w} \arrow[l] \\
\mathsf{B\,D!z} \arrow[r] & \mathsf{B\,D?z} \arrow[r] & - & \mathsf{B\,D?z} \arrow[l] & \mathsf{B\,D!z} \arrow[l]
\end{tikzcd}$$

note that the actions carried by the first communication events of B are A B?y (in the "left" branch) and C B?w (in the "right" branch), which are different outputs

which still ensures that B is passive. However, for $[\![\mathsf{G}_{(11.4)}]\!]^{@\mathsf{D}}$ *we have:*

$$\begin{tikzcd}
 & & {} \arrow[dl] \arrow[dr] & & \\
- \arrow[d] & {} \arrow[l] & & {} \arrow[r] & - \arrow[d] \\
{} \arrow[r] & - \arrow[dl] & & - \arrow[dr] & {} \arrow[l] \\
\mathsf{B\,D!z} \arrow[r] & \mathsf{B\,D?z} \arrow[r] & - & \mathsf{B\,D?z} \arrow[l] & \mathsf{B\,D!z} \arrow[l]
\end{tikzcd}$$

The first communication events of D carry the same action in both branches; this violates both the conditions of active and passive participants and make the g-choreography $\mathsf{G}_{(11.4)}$ *according to Definition 11.1.* ◇

Intuitively, the g-choreography $\mathsf{G}_{(11.4)}$ in Example 11.4 is "reasonable", despite our definition rules it out as non well-branched. In fact, participants D has the same behaviour independently of the branch chosen in the choice; in other words, D is oblivious of the choice and, with its communications, D cannot mislead the participants actually involved in the choice (namely, A and B in this case).

We will now introduce a more general notion which admits more well-branched g-choreography, including $\mathsf{G}_{(11.4)}$ in Example 11.4. The generalised notion of well-branchedness relies on the concept of "common" part, with respect to a participant A, of the two happens-before relations H and H' corresponding to the branches of a choice.

For a happens-before relation H we define

$$\widehat{H} = \{\langle e, e' \rangle \in \mathcal{E} \times \mathcal{E} \mid \exists (\!| E, E' |\!) \in H : \ e \in E \text{ and } e' \in E'\} \subseteq \mathcal{E} \times \mathcal{E}$$

Also, given a graph G, observe that the happens-before relation (if any) $[\![\mathsf{G}]\!]$ yields an order $\leqslant_{\mathsf{G}} \subseteq \mathcal{E} \times \mathcal{E}$ defined as

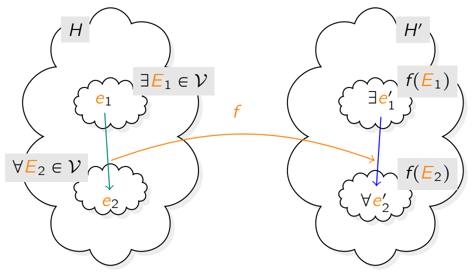
$$e \leqslant_{\mathsf{G}} e' \iff [\![\mathsf{G}]\!] \text{ is defined } \ \wedge \ \langle e, e' \rangle \in \widehat{[\![\mathsf{G}]\!]}^{\star}$$

Remark 16 *The order $\leqslant_{\mathsf{G}}$ is* partial, *namely there might be e and e', events of* G, *for which neither $e \leqslant_{\mathsf{G}} e'$ nor $e' \leqslant_{\mathsf{G}} e$.*

Before introducing the notion of *reflectivity*, we set some terminology. Two vertexes $e_1, e_2 \in H$ are *independent in a hypergraph* H if there are $h \in H$ and $e'_1, e'_2 \in \mathsf{ef}(h)$ such that, for each $i, j \in \{1, 2\}$, $\langle e'_i, e_j \rangle \in \widehat{(H^{\star})} \iff i = j$; also, for a participant $\mathsf{A} \in \mathcal{P}$, a set of vertices $E \subseteq H$ is A*-uniform in* H if $E \cap \mathbb{Z}_0 = \varnothing$, $\mathsf{sbj}(E) = \{\mathsf{A}\}$, $\mathsf{act}(E)$ is a singleton, and each $e \neq e' \in E$ are not independent and are such that $\{\langle e, e' \rangle, \langle e', e \rangle\} \cap \widehat{(H^{\star})} = \varnothing$.

Definition 11.4. Given a participant $\mathsf{A} \in \mathcal{P}$, a partition $\mathcal{V}$ of a subset of vertices of H A*-reflects* a partition $\mathcal{V}'$ of subsets of vertices of H' if there is a bijection $f : \mathcal{V} \to \mathcal{V}'$ such that the following conditions hold:

- for each $E \in \mathcal{V}$ both E and $f(E)$ are A-uniform and $\mathsf{act}(E) = \mathsf{act}(f(E))$
- $\forall E_2 \in \mathcal{V}, e_2 \in E_2, \ \langle e_1, e_2 \rangle \in \widehat{H} : \ \mathsf{sbj}(e_1) = \mathsf{A} \implies (\exists E_1 \in \mathcal{V} : \ e_1 \in E_1 \ \wedge$ $\forall e \in E_2, e' \in e_1 : \ \langle e, e' \rangle \notin \widehat{H} \ \wedge \ \forall e'_2 \in f(E_2) \exists e'_1 \in f(E_1) : \ \langle e'_1, e'_2 \rangle \in \widehat{H'})$



The relations H and H' have to be thought of as specifying the causal relations of two branches of a distributed choice. All the vertexes of $E \in \mathcal{V}$ and of $f(E)$ have the same subject.
The bijection f preserves both actions and causality relation in all the sets $E \in \mathcal{V}$. So, any predecessor e_1 in H of a vertex $e_2 \in E_2$ with the same subject of e_2 must be in a set E_1 of $\mathcal{V}$. Moreover, the f-images of E_1 and E_2 must reflect such order in H'. Such condition must also hold for the inverse of f.

Fig. 10: Reflectivity

- $\forall E'_2 \in \mathcal{V}', e'_2 \in E'_2, \langle e'_1, e'_2 \rangle \in \widehat{H'} : \mathsf{sbj}(e'_1) = \mathsf{A} \implies (\exists E'_1 \in \mathcal{V}' : e'_1 \in E'_1 \wedge \forall e \in E'_2, e' \in e'_1 : \langle e, e' \rangle \notin \widehat{H'} \wedge \forall e_2 \in f^{-1}(E'_2) \exists e_1 \in f^{-1}(E'_1) : \langle e_1, e_2 \rangle \in \widehat{H})$.

An intuitive explanation of this notion is given in Fig. 10. Reflectivity allows us to generalise the notions of active and passive participants, and therefore it makes well-branchedness more general.

For a participant $\mathsf{A} \in \mathcal{P}$, two g-choreography $\mathsf{G}, \mathsf{G}' \in \mathcal{G}$, a partition $\mathcal{V}$ of a subset of vertices of $[\![\mathsf{G}]\!]$, and a partition $\mathcal{V}'$ of a subset of vertices of $[\![\mathsf{G}']\!]$ we say that (E, E') is the A-*branching pair of* $\mathsf{G} + \mathsf{G}'$ *with respect to* $\mathcal{V}$ *and* $\mathcal{V}'$ if

$$\mathcal{V}' \ \mathsf{A}\text{-reflects } \mathcal{V} \quad \text{and} \quad \begin{cases} E = \bigcup \mathsf{cs}(\mathsf{fst}\,([\![\mathsf{G}]\!]^{@\mathsf{A}})) \setminus \bigcup \mathcal{V} \\ \text{and} \\ E' = \bigcup \mathsf{cs}(\mathsf{fst}\,([\![\mathsf{G}']\!]^{@\mathsf{A}})) \setminus \bigcup \mathcal{V}' \end{cases}$$

we write $(E_1, E_2) = \mathsf{div}_{\mathsf{A}}^{\mathcal{V},\mathcal{V}'}(\mathsf{G}, \mathsf{G}')$ when E A-reflects E' with respect to $\mathcal{V}$ and $\mathcal{V}'$ (otherwise $\mathsf{div}_{\mathsf{A}}^{\mathcal{V},\mathcal{V}'}(\mathsf{G}, \mathsf{G}')$ is undefined). Intuitively, the behaviour of A in the two branches G and G' can be the same up to the point of branching $\mathsf{div}_{\mathsf{A}}^{\mathcal{V},\mathcal{V}'}(\mathsf{G}, \mathsf{G}')$. The notion of A-reflectivity is used to identify such common behaviour (i.e., all events in E and E') and to ignore it when checking the behaviour of A in the branches. In fact, by taking the A-only parts of these hypergraphs and selecting their first interactions (that is the A-branching pair

E, E') we identify when the behaviour of A in G starts to be different with respect to behaviour in G'.

11.3.1 Active and passive roles

The intersection of sets of events $E \sqcap E'$ disregards control points: $E \sqcap E' = \{\mathsf{act}(e) : e \in E\} \cap \{\mathsf{act}(e') : e' \in E'\}$. A participant $\mathsf{A} \in \mathcal{P}$ is *passive* in $\mathsf{G} + \mathsf{G}'$ with respect to $\mathcal{V}$ and $\mathcal{V}'$ if, assuming $(E, E') = \mathsf{div}_{\mathsf{A}}^{\mathcal{V},\mathcal{V}'}(\mathsf{G}, \mathsf{G}')$, the following hold

$$E \sqcap \{e \in \mathsf{G}' \mid \nexists e' \in E' : e \leqslant_{\mathsf{G}'} e'\} = \varnothing \qquad E \cup E' \subseteq \mathcal{E}^{?}$$
$$E' \sqcap \{e \in \mathsf{G} \mid \nexists e' \in E : e \leqslant_{\mathsf{G}} e'\} = \varnothing \qquad E = \varnothing \iff E' = \varnothing$$

Thus, the behaviour of A in G and G' must be the same up to a point where she receives either of two different messages, each one identifying which branch had been selected. Clearly, A cannot perform outputs at the points of communicating system. We say that a participant A is *passive* in $\mathsf{G} + \mathsf{G}'$ if such E and E' exist.

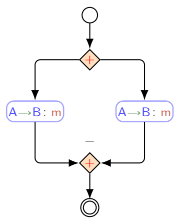
A participant $\mathsf{A} \in \mathcal{P}$ is *active* in $\mathsf{G} + \mathsf{G}'$ with respect to $\mathcal{V}$ and $\mathcal{V}'$ if, assuming $(E, E') = \mathsf{div}_{\mathsf{A}}^{\mathcal{V},\mathcal{V}'}(\mathsf{G}, \mathsf{G}')$,

$$E \cup E' \subseteq \mathcal{E}^{!} \qquad E \sqcap E' = \varnothing \qquad E \neq \varnothing \qquad E' \neq \varnothing$$

Thus, the behaviour of A in G and G' must be the same up to the point where she informs the other participants, by sending different messages, which branch she choses. We say that a participant A is *active* in $\mathsf{G} + \mathsf{G}'$ if such E and E' exist. Interestingly, if one takes the empty reflection in the determination of active and passive roles, the definition above yield exactly the same notion of Definition 11.1.

11.3.2 Some examples

When it exists, the active participant is the selector of the choice. Unlike the one in Definition 11.1, the more general notion well-branchedness does not require the selector to exist. For instance, the choreography



$$= \qquad (\mathsf{A}{\to}\mathsf{B}\colon \mathsf{m} + \mathsf{A}{\to}\mathsf{B}\colon \mathsf{m})$$

is well-branched even if it has no active participant. The simple notion of Definition 11.1 does not hold also for the following example:

$$\mathsf{A}{\to}\mathsf{B}\colon \mathsf{m}; \mathsf{B}{\to}\mathsf{C}\colon \mathsf{x} + \mathsf{A}\xrightarrow{j}\mathsf{B}\colon \mathsf{m}; \mathsf{B}{\to}\mathsf{C}\colon \mathsf{y}$$

here the problem is that the two branches have the same first interactions, so the communication action of say A in one branch occurs also in the other branch. Instead, using reflection on the $(\!|\mathsf{A}\,\mathsf{B}!\mathsf{m}, \mathsf{A}\,\mathsf{B}?\mathsf{m}|\!)$ and $(\!|\mathsf{A}\,\mathsf{B}!\mathsf{m}, \mathsf{A}\,\mathsf{B}?\mathsf{m}|\!)$, our framework establishes that B is active, and both A and C are passive, making the choicewell-branched.

Part III

...To Local Specifications

Chapter 4

Local views as automata

...act local!

This chapter reviews the automata model that we adopt to specify local views. We illustrate the precise relation between global specifications and local ones using the notion of projection (already informally discussed in Part II). This model is very close to some programming languages such as Erlang and the actor model of Scala.

12 An intuition of projections

As discussed in § 1, an appealing aspect of choreography is the possibility of obtaining local viewpoints from global ones. This is often achieved by *projecting* the global viewpoint.

An intuitive description of projections can be given using sequence diagrams. For this we consider the diagram in Fig. 11 (the actual choreography described in such diagram is immaterial for our purposes). The diagram in Fig. 11 can be "projected onto each one of the three participants" WebSite, WareHouse, and Bank so to obtain a specification of each of them. We consider projection as the operation of *focusing on a single role* of the choreography at the time. This is achieved by "blurring" any other participant. For instance, the projection of

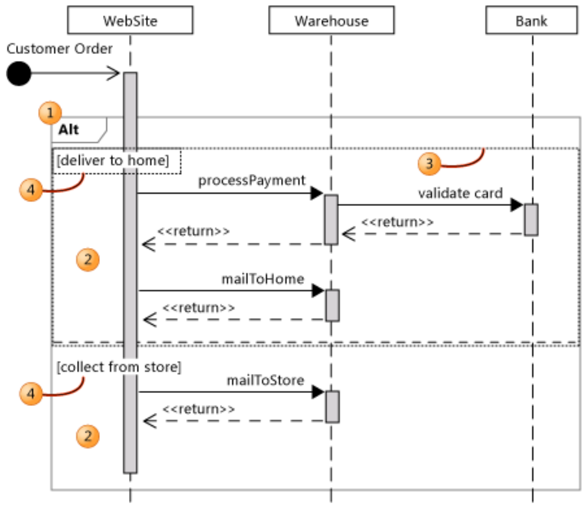
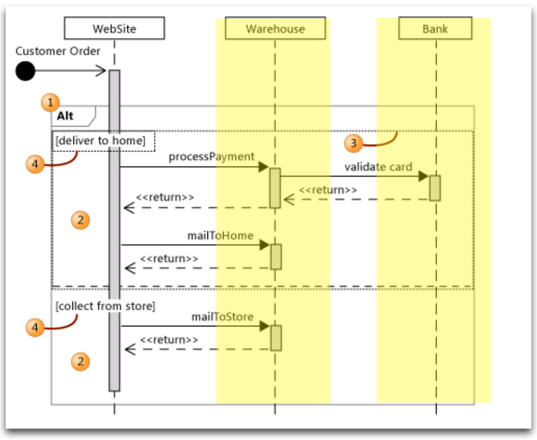
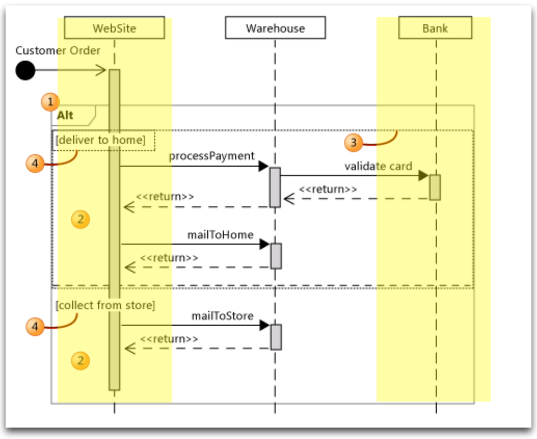


Fig. 11: A sequence diagram borrowed from http://msdn.microsoft.com/en-us/library/dd465153.aspx

the diagram with respect to WebSite could be thought of as being represented by the following figure



while the one of the WareHouse by the following one



The above idea is rather informal and the projections do not correspond to any meaningful model of specification. Instead, a projection operation can be precisely defined on global graphs as we explain in the following.

12.1 Communicating Machines

The model of *communicating finite-state machines* (CFSMs) was proposed in [6] as a convenient setting to analyse communication protocols. Informally, this model uses finite state automata (aka machines) to represent the behaviour of distributed participants that interact exchanging messages. When a machine sends a message to another machine, the message is "stored" into a queue accessible to the receiver. Dually, a machine willing to input a message visits its queues and consumes the message (if any).

In order to adopt CFMSs to represent local viewpoints of choreographies we fix some sets, notations, and terminology.

The set of *labelss* (ranged over by ℓ) is

$$\mathcal{L} \quad := \quad \{\varepsilon\} \quad \cup \quad \mathcal{C} \times \{!\} \times \mathcal{M} \quad \cup \quad \mathcal{C} \times \{?\} \times \mathcal{M} \qquad (22)$$

Intuitively, an element $\ell \in \mathcal{L}$ denotes either an internal computation (ε) or a communication action; more precisely, $\ell = (\mathsf{A}\,\mathsf{B}, !, \mathsf{m})$ denotes an output on channel $\mathsf{A}\,\mathsf{B}$ of the symbol $\mathsf{m} \in \mathcal{M}$ and $\ell = (\mathsf{A}\,\mathsf{B}, ?, \mathsf{m})$ denotes an input from channel $\mathsf{A}\,\mathsf{B}$ of the symbol $\mathsf{m} \in \mathcal{M}$. Hereafter, we adopt the (more evocative) notation $\mathsf{A}\,\mathsf{B}!\mathsf{m}$ instead of $(\mathsf{A}\,\mathsf{B}, !, \mathsf{m})$ and likewise we use $\mathsf{A}\,\mathsf{B}?\mathsf{m}$ instead of $(\mathsf{A}\,\mathsf{B}, ?, \mathsf{m})$. Elements in $\mathcal{C} \times \{!\} \times \mathcal{M}$ are called *sending actions* and those in $\mathcal{C} \times \{?\} \times \mathcal{M}$ are called *receiving actions*.

Remark 17 *Elements of $\mathcal{L}$ are basically either the actions of communication events in $\mathcal{E}^{?} \cup \mathcal{E}^{!}$ or internal computations ε.*

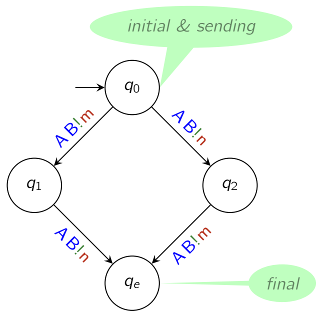
Given a set X, one can form *words on* X, namely sequences $x_1 \cdots x_n$ of elements of X; the set of words on X is customarily denoted as $X^{\star}$. We consider two types of words, the words on $\mathcal{M}$ and those on $\mathcal{L}$; for this, let $\mathcal{M}^{*}$, ranged over by w, (resp. $\mathcal{L}^{*}$, ranged over by φ) denote the set of finite words on $\mathcal{M}$ (resp. $\mathcal{L}$) with ε a distinguished symbol (not in $\mathcal{M}$ or in $\mathcal{L}$) representing the empty word. Write $|\varphi|$ for the length of a word φ, and $\varphi \cdot \varphi'$ or $\varphi\varphi'$ for the concatenation of words φ and φ' (we overload these notations for words over $\mathcal{M}$).

A communicating finite-state machine is a finite-state automaton whose transitions are labelled by actions.

Definition 12.1 (CFSM). An A-*communicating finite-state machine* is a finite transition system given by a 4-tuple $M = (Q, q_0, \mathcal{M}, \to)$ where

- Q is a finite set of *states*,
- $q_0 \in Q$ is the initial state, and
- $\to \subseteq Q \times \mathcal{L} \times Q$ is a set of *transitions* such that $\mathsf{A} = \mathsf{sbj}(\ell)$ for each $(q, \ell, q') \in \to$; we write $q \xrightarrow{\ell} q'$ for $(q, \ell, q') \in \to$ (and $q \to q'$ when ℓ is immaterial).

Example 12.1. The following machine



specifies the behaviour of the first participant of the choreography in Example 9.1 (cf. page 55). ◇

Exercise 22 *Give the machine of participant B of the choreography in Example 9.1 (on page 55).*

Given a CFSM $M = (Q, q_0, \mathcal{M}, \rightarrow)$, it is also convenient to establish some terminology: a state $q \in Q$ is a

- *final state* if it has no outgoing transition
- *sending state* if all its outgoing transitions are labelled with sending actions
- *receiving state* if all its outgoing transitions are receiving actions
- *mixed state* otherwise.

Exercise 23 *List all the final, sending, receiving, and mixed stated of the machine in Example 12.1 (on page 90).*

We will need to consider the following special class of CFSMs.

Definition 12.2 (Deterministic CFSMs). A communicating finite-state machine $M = (Q, q_0, \mathcal{M}, \rightarrow)$ is *deterministic* iff it has no transitions labelled with ε for all states $q \in Q$ and all actions $\ell \in \mathcal{L}$

$$\text{if} \quad q \xrightarrow{\ell} q' \text{ and } q \xrightarrow{\ell} q'' \quad \text{then} \quad q' = q''$$

Example 12.2. It is a simple observation that the CFSM in Example 12.1 is deterministic according to Definition 12.2. ◇

Remark 18 *Sometimes, a CFSM is considered deterministic when* $q \xrightarrow{\mathsf{S\,R!m}} q'$ *and* $q \xrightarrow{\mathsf{S\,R!m'}} q''$ *then* $\mathsf{m} = \mathsf{m'}$ *and* $q' = q''$*. Here, we follow a different definition in order to represent branching constructs.*

By putting together CFSMs we obtain communicating systems:

Definition 12.3 (Systems). Given an A-CFSM $M_{\mathsf{A}} = (Q_{\mathsf{A}}, q_{0\mathsf{A}}, \mathcal{M}, \rightarrow_{\mathsf{A}})$ for each $\mathsf{A} \in \mathcal{P}$, the tuple $\mathbf{S} := (M_{\mathsf{A}})_{\mathsf{A}\in\mathcal{P}}$ is a *communicating system*. A *configuration* of $\mathbf{S}$ is a pair $s = \langle \mathbf{q} \; ; \; \mathbf{w} \rangle$ where $\mathbf{q} = (q_{\mathsf{A}})_{\mathsf{A}\in\mathcal{P}}$ with $q_{\mathsf{A}} \in Q_{\mathsf{A}}$ and where $\mathbf{w} = (w_{\mathsf{A\,B}})_{\mathsf{A\,B}\in\mathcal{C}}$ with $w_{\mathsf{A\,B}} \in \mathcal{M}^*$; component $\mathbf{q}$ is the *control state* and $q_{\mathsf{A}} \in Q_{\mathsf{A}}$ is the *local state* of machine M_{A}. The *initial configuration* of $\mathbf{S}$ is $s_0 = \langle \mathbf{q}_0 \; ; \; \mathbf{f} \rangle$ with $\mathbf{q}_0 = (q_{0\mathsf{A}})_{\mathsf{A}\in\mathcal{P}}$ and $\mathbf{f}$ a map assigning to each channel the empty word ε (namely, $\mathbf{f}(\mathsf{A\,B}) = \varepsilon$ for all $\mathsf{A\,B} \in \mathcal{C}$).

Given a tuple $\langle t \rangle = (t_i)_{i\in I}$ over an index set I, $\mathbf{t}[j] = t_j$ for each $j \in I$; for instance, for a system $\mathbf{S} = (M_{\mathsf{A}})_{\mathsf{A}\in\mathcal{P}}$ and $\mathsf{B} \in \mathcal{P}$, $\mathbf{S}[\mathsf{B}] = M_{\mathsf{B}}$. Hereafter, for each participant $\mathsf{A} \in \mathcal{P}$, we fix a machine $M_{\mathsf{A}} = (Q_{\mathsf{A}}, q_{0\mathsf{A}}, \mathcal{M}, \rightarrow_{\mathsf{A}})$ and let $\mathbf{S} = (M_{\mathsf{A}})_{\mathsf{A}\in\mathcal{P}}$ be the corresponding communicating system.

Definition 12.4 (Reachable states and configurations). Given a system $\mathbf{S}$, a configuration $s' = \langle \mathbf{q}' \; ; \; \mathbf{w}' \rangle$ of $\mathbf{S}$ is *reachable* from another configuration $s = \langle \mathbf{q} \; ; \; \mathbf{w} \rangle$ of $\mathbf{S}$ if, and only if, either of the following conditions holds:

1. $\mathbf{q}[\mathsf{S}] \xrightarrow{\mathsf{S\,R!m}}_{\mathsf{S}} \mathbf{q}'[\mathsf{S}]$ is a transition in $\mathbf{S}[\mathsf{S}]$

   a. $\mathbf{q}'[\mathsf{A}] = \mathbf{q}[\mathsf{A}]$ for all $\mathsf{A} \neq \mathsf{S}$, and

   b. $\mathbf{w}'[\mathsf{S\,R}] = \mathbf{w}[\mathsf{S\,R}] \cdot \mathsf{m}$ and $\mathbf{w}'[\mathsf{A\,B}] = \mathbf{w}[\mathsf{A\,B}]$ for all $\mathsf{A\,B} \neq \mathsf{S\,R}$; or

2. $\mathbf{q}[\mathsf{R}] \xrightarrow{\mathsf{S\,R?m}}_{\mathsf{R}} \mathbf{q}'[\mathsf{R}]$ is a transition in $\mathbf{S}[\mathsf{R}]$

   a. $\mathbf{q}'[\mathsf{A}] = \mathbf{q}[\mathsf{A}]$ for all $\mathsf{A} \neq \mathsf{R}$, and

   b. $\mathbf{w}[\mathsf{S\,R}] = \mathsf{m} \cdot \mathbf{w}'[\mathsf{S\,R}]$ and $\mathbf{w}'[\mathsf{A\,B}] = \mathbf{w}[\mathsf{A\,B}]$ for all $\mathsf{A\,B} \neq \mathsf{S\,R}$.

3. $\mathbf{q}[\mathsf{B}] \xrightarrow{\varepsilon}_{\mathsf{B}} \mathbf{q}'[\mathsf{B}]$ is a transition in $\mathbf{S}[\mathsf{B}]$

   a. $\mathbf{q}'[\mathsf{A}] = \mathbf{q}[\mathsf{A}]$ for all $\mathsf{A} \neq \mathsf{B}$, and

b. $\mathbf{w} = \mathbf{w}'$.

Write $s \overset{\ell}{\Rightarrow} s'$ when s reaches s' with an action ℓ.

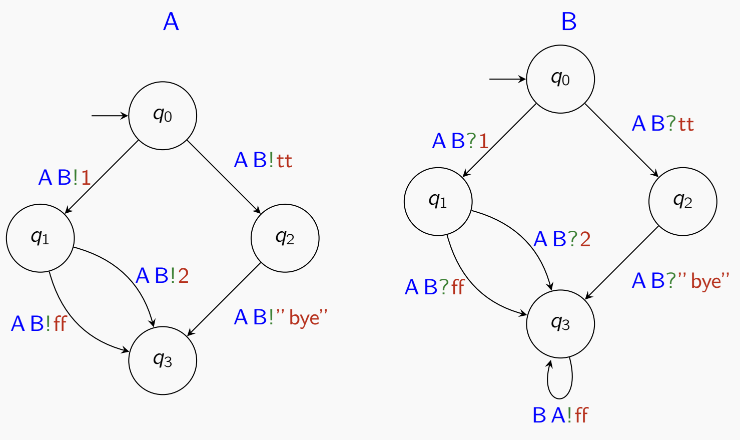
Condition (1b) in Definition 12.4 puts m on channel S R, while (2b) gets m from channel S R. Finally, condition (3b) establishes that internal computations do not modify any communication buffer.

Definition 12.5 (Reachability). Let $s_1 \xRightarrow{\ell_1 \cdots \ell_m} s_{m+1}$ hold iff, for some configurations $s_2, \ldots, s_m$ we have that $s_1 \overset{\ell_1}{\Rightarrow} s_2 \cdots s_m \overset{\ell_m}{\Rightarrow} s_{m+1}$. The set of *reachable configurations of* $\mathbf{S}$ from s is

$$\mathsf{RC}(\mathbf{S}, s) = \left\{ s \;\middle|\; \text{there are } \ell_1, \ldots, \ell_m \text{ and a configuration } s' \text{ of } \mathbf{S} \text{ s.t. } s \xRightarrow{\ell_1 \cdots \ell_m} s' \right\}$$

The set of reachable configurations of $\mathbf{S}$ is $\mathsf{RC}(\mathbf{S}) = \mathsf{RC}(\mathbf{S}, s_0)$ where s_0 is the initial configuration of $\mathbf{S}$.

Exercise 24 *Let* $\mathbf{S}$ *be the communicating system consisting of the following CFSM*



Show that a configuration of the form $\langle (\mathsf{A}_1, \mathsf{B}_2) \; ; \; \mathbf{w} \rangle$ *cannot be in* $\mathsf{RC}_2(\mathbf{S})$.

Exercise 25 *The communicating system of Exercise 24 (on page 93) has an infinite number of reachable configurations. Would the addition of transition $q_3 \xrightarrow{\mathsf{B\,A?ff}} q_3$ make the reachability set of the new system finite? Justify your answer.*

We can now precisely define well-behaved (or safe) communicating system.

Definition 12.6 (Well-behaved (or safe) systems). Fix a communicating system $\mathbf{S}$ and let $s = \langle \mathbf{q} \; ; \; \mathbf{w} \rangle$ be one of its configurations. We say that s is a

deadlock configuration if $\mathbf{w}[\mathsf{A\,B}] = \varepsilon$ for all $\mathsf{A\,B} \in \mathcal{C}$, $\mathbf{q}[\mathsf{A}]$ is a receiving state or final statefor every $\mathsf{A} \in \mathcal{P}$, and there is $\mathsf{R} \in \mathcal{P}$ such that $\mathbf{q}[\mathsf{R}] \xrightarrow{\mathsf{S\,R?m}}_{\mathsf{R}} q$ for a state q, i.e., all the buffers are empty, there is at least one machine waiting for a message, and all the other machines are either in a final state or receiving state;

orphan-message configuration if there is a $\mathsf{A\,B} \in \mathcal{C}$ such that $\mathbf{w}[\mathsf{A\,B}] \neq \varepsilon$ and for all configurations $\langle \mathbf{q}' \; ; \; \mathbf{w}' \rangle$ reachable in $\mathbf{S}$ from $\langle \mathbf{q} \; ; \; \mathbf{w} \rangle$, $\mathbf{w}'[\mathsf{A\,B}]$ is a prefix of $\mathbf{w}[\mathsf{A\,B}]$, i.e., there is at least a non-empty buffer whose top message is never consumed;

unspecified-reception configuration if there exists $\mathsf{R} \in \mathcal{P}$ such that $\mathbf{q}[\mathsf{R}]$ is a receiving state, there is $\mathsf{S\,R} \in \mathcal{C}$ such that $|\mathbf{w}[\mathsf{S\,R}]| > 0$, and $\mathbf{q}[\mathsf{R}] \xrightarrow{\mathsf{S\,R?m}}_{\mathsf{R}} \mathbf{q}'[\mathsf{R}]$ implies that $\mathbf{w}[\mathsf{S\,R}] \notin \mathsf{m}\mathcal{M}^*$, i.e., $\mathbf{q}[\mathsf{R}]$ is prevented from receiving any message from any of its buffers and there is at least a non-empty buffer of R.

Communicating system $\mathbf{S}$ is *well-behaved* if for each of its configurations $s \in \mathsf{RC}(\mathbf{S})$, s is neither a deadlock, nor an orphan-message, nor an unspecified-reception configuration.

The definition of deadlock and unspecified-reception configuration are borrowed from [8, Def. 12].

12.2 Projecting g-choreographies to CFSM

Given two CFSMs $M = (Q, q_0, \rightarrow)$ and $M' = (Q', q_0, \rightarrow')$, write $M \sqcup M'$ for the machine $(Q \cup Q', q_0, \rightarrow \cup \rightarrow')$; observe that M and M' have the same initial state. Also, $M \cap M'$ denotes $Q \cap Q'$. The product of M and M' is defined as usual as $M \otimes M' = (Q \times Q', (q_0, q_0'), \rightarrow'')$ where $\big((q_1, q_1'), e, (q_2, q_2')\big) \in \rightarrow''$ if, and only if,

$$\big((q_1, e, q_2) \in \rightarrow \ \text{ and } q_1' = q_2'\big) \qquad \text{or} \qquad \big((q_1', e, q_2') \in \rightarrow' \ \text{ and } q_1 = q_2\big)$$

We also let $\Delta(M)$ denote the CFSM obtained by *determinising* (using e.g., the classical algorithms [20]) M when interpreting it as finite automata on the alphabet $\mathcal{L}$.

In the following, we let q_0, q_e, $q_e{}'$, $\underline{q_0}$, $\overline{q_0}$, $\underline{q_e}$, and $\overline{q_e}$ range over a fixed set that we use as set[18] of states of CFSMs projected from a graph. Let G be a g-choreography, the function $\mathsf{G} \downarrow_{\mathsf{A}}^{q_0,q_e}$ yields the projection (in the form of a CFSMs) of the choreography over the participant A using q_0 and q_e as initial and sink states respectively.

Definition 12.7 (Projection). The *projection of a g-choreography* $\mathsf{G} \in \mathcal{G}$ *on a participant* $\mathsf{A} \in \mathcal{P}$ *and states* $q_0 \neq q_e$ is the CFSMs defined as follows:

[18] Any infinite set can be chosen to represent states.

$$
\mathsf{G}\downarrow_{\mathsf{A}}^{q_0,q_e} = \begin{cases}
\to \{q_0, q_e\} \to & \text{if } \mathsf{G} = \mathsf{B}\to\mathsf{C}\colon \mathsf{m},\ \mathsf{A} \neq \mathsf{B},\ \text{and } \mathsf{A} \neq \mathsf{C} \\
\to q_0 \xrightarrow{\mathsf{A\,B!m}} q_e \to & \text{if } \mathsf{G} = \mathsf{A}\to\mathsf{B}\colon \mathsf{m} \\
\to q_0 \xrightarrow{\mathsf{B\,A?m}} q_e \to & \text{if } \mathsf{G} = \mathsf{B}\to\mathsf{A}\colon \mathsf{m} \\
\mathsf{G}_1 \downarrow_{\mathsf{A}}^{q_0,q_e'} \sqcup \mathsf{G}_2 \downarrow_{\mathsf{A}}^{q_e',q_e} & \text{if } \mathsf{G} = \mathsf{G}_1 ; \mathsf{G}_2,\ q_0 \neq q_e' \neq q_e, \text{ and} \\
 & \quad \mathsf{G}_1 \downarrow_{\mathsf{A}}^{q_0,q_e'} \cap \mathsf{G}_2 \downarrow_{\mathsf{A}}^{q_e',q_e} = \{q_e'\} \\
\mathsf{G}_1 \downarrow_{\mathsf{A}}^{q_0,q_e} \sqcup \mathsf{G}_2 \downarrow_{\mathsf{A}}^{q_0,q_e} & \text{if } \mathsf{G} = \mathsf{G}_1 + \mathsf{G}_2 \text{ and} \\
 & \quad \mathsf{G}_1 \downarrow_{\mathsf{A}}^{q_0,q_e} \cap \mathsf{G}_2 \downarrow_{\mathsf{A}}^{q_0,q_e} = \{q_0, q_e\} \\
\mathsf{G}_1 \downarrow_{\mathsf{A}}^{q_0,q_e} \times \mathsf{G}_2 \downarrow_{\mathsf{A}}^{q_0,q_e} & \text{if } \mathsf{G} = \mathsf{G}_1 \mid \mathsf{G}_2 \\
\mathsf{G}' \downarrow_{\mathsf{A}}^{q_0,q} \sqcup \mathsf{G}_{1s} \sqcup \mathsf{G}_{es} & \text{if } \mathsf{G} = *\mathsf{G}'@\mathsf{A} \\
\mathsf{G}' \downarrow_{\mathsf{A}}^{q_0,q} \sqcup \mathsf{G}_{1r} \sqcup \mathsf{G}_{er} & \text{if } \mathsf{G} = *\mathsf{G}'@\mathsf{B} \text{ with } \mathsf{A} \neq \mathsf{B} \\
 & \quad \text{and } \mathsf{B} \text{ participant of } \mathsf{G}'
\end{cases}
$$

where in the last two cases we define

$$\mathsf{G}_{1s} = \left(\mathsf{A}\to\mathsf{B}_1\colon \mathsf{loop\_q_0} \mid \cdots \mid \mathsf{A}\to\mathsf{B}_h\colon \mathsf{loop\_q_0}\right) \downarrow_{\mathsf{A}}^{q,q_0}$$

$$\mathsf{G}_{es} = \left(\mathsf{A}\to\mathsf{B}_1\colon \mathsf{exit\_q_e} \mid \cdots \mid \mathsf{A}\to\mathsf{B}_h\colon \mathsf{exit\_q_e}\right) \downarrow_{\mathsf{A}}^{q,q_e}$$

$$\mathsf{G}_{1r} = \ \to q \xrightarrow{\mathsf{A\,B?loop\_q_0}} q_0 \to \qquad \text{and} \qquad \mathsf{G}_{er} = \ \to q \xrightarrow{\mathsf{A\,B?exit\_q_e}} q_e \to$$

with $\{\mathsf{B}_1, \dots, \mathsf{B}_h\}$ the set of participants of G' different from A and q being the only state shared between G_{1s}, G_{es}, and $\mathsf{G}' \downarrow_{\mathsf{A}}^{q_0,q} \sqcup \mathsf{G}_{1s} \sqcup \mathsf{G}_{es}$ (resp. G_{1r}, G_{er}, and $\mathsf{G}' \downarrow_{\mathsf{A}}^{q_0,q} \sqcup \mathsf{G}_{1r} \sqcup \mathsf{G}_{er}$). Moreover, messages $\mathsf{loop\_q_0}$ and $\mathsf{exit\_q_e}$ do not occur elsewhere in the g-choreography.

We now give an intuition of the different cases of Definition 12.7. First note that the map $\_ \downarrow_{\_}^{q_0,q_e}$ always returns a CFSMs with a unique initial state q_0 and final state q_e in all but the first case where states q_0 and q_e collapse into a set. This invariant of the construction allows to compose machines according to the operations of the g-choreographies.

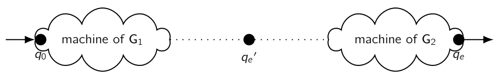
To project an interaction with respect to a participant A we have to consider three possibilities: (i) that A is not involved in the interaction, (ii) that A is the sender of the interaction, and (iii) that A is the receiver of the interaction. Possibility (i) is dealt with in the first case: the resulting machine has a single state (made by "collapsing" q_0 and q_e in a set) and no transitions (as expected). Possibility (ii) is considered in the second case: obviously, the resulting machine has a single transition labelled with the output action. Possibility (ii) is considered in the third case and it is similar to the previous one, the only difference being that the label is now the input action.

Example 12.3. The projections of A→B : m *with respect* A *and* B *and the states and are*

$$\begin{tikzcd}
{} \arrow[r] & \circ \arrow[r, "\mathsf{A\,B!m}"] & \circ \arrow[r] & {}
\end{tikzcd}
\quad \textit{and} \quad
\begin{tikzcd}
{} \arrow[r] & \circ \arrow[r, "\mathsf{A\,B?m}"] & \circ \arrow[r] & {}
\end{tikzcd}$$

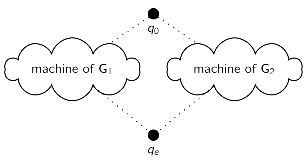
respectively. ◇

The fourth case of Definition 12.7 yields the machine for the sequential composition of two g-choreographies G_1 and G_2. The intuition is very simple and can be explained following this graphical representation:



In words, first one builds the CFSMs for G_1 and G_2 so that the final state of the former is used as the initial state of latter (this is represented by the dotted lines connected to $q_e{}'$). In this way, whenever the transitions of the first machine reach the final state, the transitions of the second machine can start, so realising the (expected) sequential behaviour.

The fifth case, for the branch of G_1 and G_2 is also simple: provided that the respective machines only share initial and final state. Consider the following graphical representation:



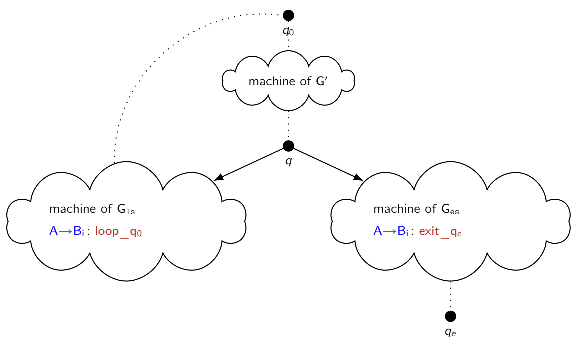
Namely, the resulting machine just follows either the executions of the machine of G_1 or the ones of the machine of G_2 just because the initial state is in common.

The sixth case takes care of the machine of parallel composition. This is obtained pretty straightforwardly by taking the product of the two machines of G_1 and G_2.

Exercise 26 *Give the machines* $\mathsf{G} \downarrow^{\cdot}_{\mathsf{A}}$ *and* $\mathsf{G} \downarrow^{\cdot}_{\mathsf{B}}$ *for the g-choreography* $\mathsf{G} = \mathsf{A} {\to} \mathsf{B} \colon \mathsf{m} \mid \mathsf{A} {\to} \mathsf{B} \colon \mathsf{m}'$.

Exercise 27 *Let* G *be the g-choreography of Exercise 26 (on page 98) and* C *be a participant different from* A *and* B*. Give the CFSM* $\mathsf{G} \downarrow^{(,),(,)}_{\mathsf{C}}$.

Finally, the last two cases deal with iteration. We have to distinguish two cases depending on whether we project with respect to the participant that controls the iteration. If A controls the iteration then the resulting machine simply connects the machine of the body G' of the iteration with a machine that is the projection of a "looping" machine $\mathsf{G}_{\mathtt{ls}}$ or of an exiting one $\mathsf{G}_{\mathtt{es}}$. The former machine sends to any other participant in the body the message of looping back to the initial state, instead $\mathsf{G}_{\mathtt{es}}$ sends the other participants the message to move to the final state q_e. Graphically:


q_0
machine of G′
q
machine of $\mathsf{G}_{\mathtt{ls}}$
A→B_i : loop_q_0
machine of G_{es}
A→B_i : exit_q_e
q_e

The last case considers when the controller of the loop is a participant $\mathsf{B} \neq \mathsf{A}$ and it is similar to the previous one.

References

1. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.
2. G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services - Concepts, Architectures and Applications. Data-Centric Systems and Applications*. Springer, 2004.
3. http://ibm.com/apieconomy.
4. C. Barreto, V. Bullard, T. Erl, J. Evdemon, D. Jordan, K. Kand, D. König, S. Moser, R. Stout, R. Ten-Hove, I. Trickovic, D. van der Rijn, and A. Yiu. Web services business process execution language version 2.0. https://www.oasis-open.org/committees/download.php/23964/wsbpel-v2.0-primer.htm, 2007.
5. J. Bonér. *Reactive Microsystems - The Evolution Of Microservices At Scale*. O'Reilly, 2018.
6. D. Brand and P. Zafiropulo. On Communicating Finite-State Machines. *JACM*, 30(2):323–342, 1983.
7. N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and orchestration conformance for system design. In *Coordination Models and Languages, 8th International Conference, COORDINATION 2006, Proceedings*, pages 63–81, 2006.
8. G. Cécé and A. Finkel. Verification of programs with half-duplex communication. *I&C*, 202(2):166–190, 2005.
9. M. Dalla Preda, M. Gabbrielli, S. Giallorenzo, I. Lanese, and M. Jacopo. Dynamic choreographies - safe runtime updates of distributed applications. In *COORDINATION 2015*, pages 67–82, 2015.
10. P. Deniélou and N. Yoshida. Multiparty session types meet communicating automata. In *ESOP 2012*, pages 194–213, 2012.
11. R. Dijkman and M. Dumas. Service-oriented design: A multi-viewpoint approach. *International Journal of Cooperative Information Systems*, 13(4):337–368, 2004.
12. E. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
13. B. Doerrfeld, C. Wood, A. Anthony, K. Sandova, and A. Laured. The api economy disruption and the business of apis. eBook. Nodic APIs, May 2016. Available at http://nordicapis.com/ebook-release-api-economy-disruption-business-apis.

14. R. W. Floyd. Assigning meaning to programs. In *Proc. Symp. in Applied Mathematics*, volume 19, 1967.

15. R. Guanciale and E. Tuosto. An abstract semantics of the global view of choreographies. In *Proceedings 9th Interaction and Concurrency Experience, ICE 2016, Heraklion, Greece, 8-9 June 2016.*, pages 67–82, 2016.

16. R. Guanciale and E. Tuosto. Semantics of global views of choreographies. *Journal of Logic and Algebraic Methods in Programming*, 95, 2017. Revised and extended version of [15]. Accepted for publication. To appear; version with proof available at http://www.cs.le.ac.uk/people/et52/jlamp-with-proofs.pdf.

17. D. Harel and P. Thiagarajan. Message sequence charts. Available at http://www.comp.nus.edu.sg/~thiagu/public_papers/surveymsc.pdf.

18. T. Hoare. An axiomatic basis of computer programming. *CACM*, 12, 1969.

19. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. *JACM*, 63(1):9:1–9:67, 2016. Extended version of a paper presented at POPL08.

20. J. E. Hopcroft. *Introduction to Automata Theory, Languages, and Computation*. Pearson Addison Wesley, 3rd edition, 2007.

21. N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, and Y. Lafon. http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217. Working Draft 17 December 2004.

22. I. Lanese, C. Guidi, F. Montesi, and G. Zavattaro. Bridging the gap between interaction- and process-oriented choreographies. In *Software Engineering and Formal Methods, SEFM 2008*, pages 323–332, 2008.

23. J. Lange and A. Scalas. Choreography synthesis as contract agreement. In *Proceedings 6th Interaction and Concurrency Experience, ICE 2013*, pages 52–67, 2013.

24. J. Lange, E. Tuosto, and N. Yoshida. From Communicating Machines to Graphical Choreographies. In *POPL15*, pages 221–232, 2015.

25. J. Lewis and M. Fowler. Microservices: a definition of this new architectural term. http://martinfowler.com/articles/microservices.html, March 2014.

26. B. Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992.

27. Z. Micskei and H. Waeselynck. Uml 2.0 sequence diagrams' semantics. http://home.mit.bme.hu/~micskeiz/sdreport/uml-sd-semantics.pdf.

28. J. Misra. Computation orchestration. In M. Broy, J. Grünbauer, D. Harel, and T. Hoare, editors, *Engineering Theories of Software Intensive Systems*, volume 195 of *NATO Science Series*, pages 285–330. Springer, 2005.

29. M. Nordio, R. Mitin, B. Meyer, C. Ghezzi, E. D. Nitto, and G. Tamburrelli. The role of contracts in distributed development. In *SEAFOOD*, pages 117–129, 2009.

30. Object Management Group. Business Process Model and Notation. http://www.bpmn.org.

31. D. Orenstein. Application programming interface, Jan. 2000. Available at http://www.computerworld.com/article/2593623/app-development/application-programming-interface.html.
32. M. Papazoglou. *Web Services and SOA: Principles and Technology*. Pearson-Prentice Hall, 2012.
33. C. Peltz. Web services orchestration and choreography. *IEEE Computer*, 36(10):46–52, 2003.
34. Z. Qiu, X. Zhao, C. Cai, and H. Yang. Towards the theoretical foundation of choreography. In *Proceedings of the 16th International Conference on World Wide Web, WWW 2007*, pages 973–982, 2007.
35. S. R. Talbot. Orchestration and choreography: Standards, tools and technologies for distributed workflows. http://www.nettab.org/2005/docs/NETTAB2005_Ross-TalbotOral.pdf.
36. H. Yang, X. Zhao, C. Cai, and Z. Qiu. Exploring the connection of choreography and orchestration with exception handling and finalization/compensation. In *Formal Techniques for Networked and Distributed Systems - FORTE 2007, 27th IFIP WG 6.1 International Conference, Proceedings*, pages 81–96, 2007.

Appendix A

Refreshing basic notation

A set is a collection of elements. Some special sets are given a name, for instance $\mathbb{Z}$ is the set of integers and $\varnothing$ is the empty set (that is the set that contains no elements). A fundamental notion is the *membership* relation between individuals and sets which states when an element belongs to a set; we write $x \in X$ to express that the individual x is a member of the set X. For example $-1 \in \mathbb{Z}$ means that -1 is a member of the set $\mathbb{Z}$, namely -1 is an integer). The negation of $\in$ is $\notin$ (for example $-1.0 \notin \mathbb{Z}$ means that -1.0 is not an integer).

Another important notion (and natural) is set *inclusion*: X is included in Y, written $X \subseteq Y$ if each element of X is also an element of Y (which could be written more succinctly as if $x \in X$ then $x \in Y$).

Given a set, it is often necessary to consider the set of all its subsets. For instance, one could be interested in all the possible subgroups of a given face-book group. The set of subsets (or *parts*) of a given set X is often denoted as 2^X and it is defined as

$$2^X = \{Y \mid Y \subseteq X\}$$

This notation is evocative of the fact that on finite sets the following fact holds:

If X if a finite number set with n elements then there are 2^n sets in 2^X.

Fixed a set U and two subsets $X \subseteq U$ and $Y \subseteq U$, it is pretty straightforward to define the usual operations on sets as follows:

$$X \cap Y = \{x \in U \mid x \in X \wedge x \in Y\} \tag{A.1}$$

$$X \cap Y = \{x \in U \mid x \in X \vee x \in Y\} \tag{A.2}$$

$$X \backslash Y = \{(x \in U \mid x \in X \wedge x \notin Y\} \tag{A.3}$$

that respectively define *intersection*, *union*, and *difference*.

A set can be specified either by enumerating its elements or by defining the *characteristic* property that each element of the set has. In the first case we just put in curly brackets the comma-separated list of elements while in the second case we write

$$\{x \in Y \mid \cdots\}$$

that reads `the set of elements` x `in (the set)` Y `such that` $\cdots$. For instance, the set V of integers between -1 and 2 can be written

$$\{-1, 0, 1, 2\} \tag{A.4}$$

or

$$\{x \in \mathbb{Z} \mid -1 \leqslant x \leqslant 2\} \tag{A.5}$$

(Sometimes, which set x ranges over is omitted because clear from the context; so the example above could be written as $\{x \in \mathbb{Z} \mid -1 \leqslant x \leqslant 2\}$ provided that it is understood that x is an integer.) Recall that in (A.4) it is immaterial that elements are listed more than once or that they are listed in a particular order. This implies that

$$\{-1, 0, 1, 2\} \qquad \text{and} \qquad \{0, -1, 1, 0, 2\}$$

denote the same set. More generally, when are two sets equal? The answer to this question is simple: sets X and Y are equal, written $X = Y$, when each element of X is also and element of Y, and vice versa. Note that not necessarily $X = Y$ if X is included in Y; for example the set of even numbers is included in $\mathbb{Z}$ but there are integer numbers that are not even. In fact, it is easy to verify that equality of sets is defined in terms of a "double" inclusion:

$$X = Y \text{ if } X \subseteq Y \text{ and, vice versa, } Y \subseteq X.$$

Given two formulae, say F, F_1, and F_2, it is useful to adopt symbols to represent logical connectives such as 'not', 'and', 'or', and 'implies' that are respectively written $\neg$, $\wedge$, $\vee$, and $\implies$. For example,

$$x > 0 \ \wedge \ x \quad \text{mod } 2 = 0 \ \wedge \ x \in \mathbb{Z}$$

states that x is a positive even integer. Implication basically a conditional statement, for instance

$$(x \in \mathbb{Z} \wedge x > 0) \implies (-x < 0 \wedge x \in \mathbb{Z}) \tag{A.6}$$

states that if x is strictly positive its negation is strictly negative (figure out why the paranthesis are important). In the previous implication, the formula $x = 0$ is called *hypothesis* (or *antecedent* and the formula $x \neq 1$ is called *thesis* (or *consequent*). It sometimes happen that both $F_1 \implies F_2$ and $F_2 \implies F_1$ hold; in this case F_1 and F_2 are said (*logically*) *equivalent*, written $F_1 \iff F_2$.

Formulae like (A.6) above are not "really meaningful" until we specify what is the *scope* of x. For instance, can we establish if

$$x = 2x \wedge x \in \mathbb{Z}$$

is true or false? The answer is 'no' because we do not know how variable x is *quantified*. A variable can be quantified either *universally* of *existentially*:

- a universal quantification is written $\forall x \in Y : \cdots$
- an existential quantification is written $\exists x \in Y : \cdots$

The former reads `for all element` x `of` Y`,` $\cdots$ and the latter reads `there is an element` x `in` Y`,` $\cdots$; example

$$\forall x \in \mathbb{Z} : \exists y \in \mathbb{Z} : y = -x$$

states that each integer has an opposite. Conventionally, a variable is assumed universally quantified when quantifiers are missing.

What do the following formulae state?

$$X \subseteq Y \iff \forall x \in X : x \in Y$$
$$X = Y \iff X \subseteq Y \wedge Y \subseteq X$$

Another useful construction on sets is the so called *cartesian product* (or just *product*) of X and Y, denoted as $X \times Y$ and made of all the pairs (x, y) such that $x \in X$ and $y \in Y$. A *relation* on X and Y is a subset of the set $X \times Y$. For example, if S is the set of students and M is the set of modules, the relation containing a pair (s, m) only if student s is registered on module m would be a subset of $S \times M$.

A special category of relations are *functions*: given $R \subseteq X \times Y$, we say that R is a *function from X to Y* (written $R : X \to Y$), if the following hold:

$$\forall x \in X : \forall y, y' \in Y : (x, y) \in R \wedge (x, y) \in R' \implies y = y'$$

which states that for each element x of Y there is a unique element y of Y such that x and y are in the relation R. In such case, we say that X is the *domain* of R and Y is the *codomain* of R. For instance, the relation $o = \{(x, y) \in \mathbb{Z} \mid x =$

$-y\}$ is the function that map each integer to its opposite. If $R : X \to Y$ and $x \in X$ then $R(x)$ denotes the (unique) element that R relates to x; we say that R assigns x (the value) $R(x)$. For instance, $o(3)$ is the number -3. In general, functions may assign the same value to more elements of their domain. For instance, consider the function *nameOf* : *Persons* → *Strings* that assign the string of the name of a person to that person. There are many elements in the domain who have the same value (that is there are may people with the same name). The functions for which this does not happen are called *injective*, more formally, we say that $R : X \to Y$ is an injective function when

$$\forall x, x' \in X : \;\; : \; R(x) = R(x') \implies x = x'$$

Another class of interesting functions is the one characterised by the following property:

$$\forall y \in Y \; : \; \exists x \in X \; : \; R(x) = y$$

Such property establishes that for all elements y of the codomain there is an element of the domain whose value through R is y. For instance, the function *ownerOf* : *carOwner* → *soldCars* is surjective. When a function is both injective and surjective we call it *bijective*. Such functions establish that each element of the domain has a unique related element in the codomain, and vice versa. This means that their inverse is also a function, where the inverse R^{-1} of a relation R is defined as:

$$R^{-1} = \big\{(y, x) \;\big|\; (x, y) \in R\big\}$$

An example of bijiective function is the function o returning the opposite of an integer (defined above). Bijective function are important because the define one-to-one relations among sets.

Index