A choregraphy-based approach to distributed applications

December 5, 2017

Contents

Part I Overview **11**

1 Context and motivations **13**

1 Distributed applications 13

2 “I have this terrible feeling of deja vu” 15

3 Distributed coordination 19

Part II Architectural Design **23**

2 Software Architectures **25**

4 Complexity of software 25

5 Architectural elements & distributed applications 27

3 A simple notation for software architectures **31**

6 Representing software architectures 31

7 Design, runtime, and style 32

7.1 Balancing coupling & cohesion 34

7.2 A matter of style 35

7.2.1 Client-Server style 37

7.2.2 Pipe-filter style 38

7.2.3 Blackboard style 39

Part III From Global Specifications... 41

4 What is a choreography? 43

8 An intuitive account . . . 43
9 Global views as interaction diagrams . . . 46
10 An execution model, informally . . . 49
10.1 From sequence diagrams to "programs" . . . 51
10.2 Running sequence diagrams . . . 52
11 More expressiveness . . . 56
11.1 Branching . . . 56
11.2 A note on choice and non-determinism . . . 58
11.2.1 Distributed choices . . . 58
11.2.2 Non-determinism . . . 59
11.3 Problems with branching . . . 60

5 A more precise framework 65

12 Global views as graphs . . . 65
13 Semantics of g-choreographies . . . 69
13.1 Basic definitions . . . 70
13.2 Hypergraphs and their order . . . 71
13.3 Composing happens-before relations . . . 76
14 The semantics of graphs . . . 78
14.1 Active & passive roles...easily . . . 81
14.2 Applying semantic equations . . . 84
14.3 A generalisation of well-branchedness . . . 91
14.3.1 Active and passive roles . . . 94
14.3.2 Some examples . . . 95

Part IV ...To Local Specifications 97

6 An automata model of local views 99

15 An intuition of projections . . . 99

15.1 Communicating Finite-State Machines 101
15.2 Projecting g-choreographies to CFSM 106
References 111

A Refreshing basic notation **115**

Index **121**

List of Figures

1 Two Erlang programs 26

2 Lifeline 47
3 Asynchronous call 47
4 The “unique selector” problem 61
5 Deadlock issue 61
6 Determinacy issue 63

7 A graphical notation for g-choreographies 67
8 Examples of choreographies 68
9 A happens-before relation 72
10 Some happens-before relations 75
11 Examples of sequential composition 78
12 Reflectivity 93

13 A sequence diagram 100

Foreword

These notes provide an introduction to the use of choreographies to the engineering 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.

Part I

Overview

Chapter 1
Context and motivations

Distribution, interactions, and all that

After discussing the importance of distribution in modern software, this chapter discusses the relations between software production and some theories and formal methods developed a few years ago. The chapter concludes with some basic thoughts on distributed coordination in general and about orchestration and choreography.

1 Distributed applications

Distributed applications are nowadays ubiquitous. Practically, there is no single application which is today conceived to be running entirely by its own without needing to dynamically interact with other applications. The combined effect of the ubiquitous network connectivity (which is characterising the urban areas of the most advanced countries) and the possibility of carrying devices capable of communicating and computing determined a transition of applications. Think of the shift due to the diffusion of mobile phones which imposed the development of 'app' version of software. Big vendors, as well as small software companies, had to satisfy the appetite of users who want their data and applications "always handy".

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.

As a matter of fact, software becoming ubiquitous and distributed applications are becoming predominant in almost all sectors; not only they are used in commercial applications, they are increasingly 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.

Paradigms like *service-oriented computing* or its recent offspring *cloud computing* envisage distributed applications as autonomous components that dynamically discover and bind to each other. Recent approaches to software development are paradigmatic in this sense. Specifically, *microservices* take to an extreme the idea of software development consisting of gluing together components that are loosely coupled and interact by exchanging messages. This is revolutionising the traditional *monolithic* approach, where applications are built around databases that need to be shared by the different components. This shift is determining the era of the so called *API economy*, which advocates the use of elaborated APIs (after application programming interfaces) to allow the smooth integration of software. There is already some

evidence that this new approaches yield substantial improvements to software development. For instance, microservices are considered crucial when scalability and reduced time-to-market are key requirements. On the other hand, developers highlight some drawbacks. More precisely, 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.

2 “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 what we aim at. Consider the following simple Erlang program:

```
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,
  io:format("ping finished~n", []);

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

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

Function `ping` consists of two clauses (starting at lines 1 and 3 respectively); the first clause is executed 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. 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?

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[1] to the seminal work of Floyd [12], Dijkstra [11] and Hoare [16] 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 [23]. 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[2]

```
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 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 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.

In recent years, sofware has been shifting from 'stand-alone' applications to mobile and dynamically composable ones. This has in turn also increased

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

[2] I use a fictional syntax.

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.

Remark 1 *The so called* behavioural types *(that constrain the reciprocal interactions of distributed participants) are an example of such formalisms. Notably, behavioural types are at the basis of the recent project* Behavioural Types for Reliable Large-Scale Software Systems *(BETTY, oc-2011-2-10054-EU COST Action) which aims to develop new foundations, programming languages, and software development methods for communication-intensive distributed systems.*

Unlike in the classical DbC approach (where contracts are specified by directly annotating software with assertions) application-level protocols specify high-level contracts at design level. Such difference should probably be more deeply explored as it introduces interesting questions (that will not be addressed here). 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.

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 mentioned above.*[3] *However, this seems to be a necessary price to pay; since the problem is complex and simplistic approaches do not provide satisfactory solutions.*

[3] In this respect it is interesting the reaction of students in software engineering from different universities reported in [26].

3 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" [29], whereas choreography describes the behaviour of involved parties from a "global viewpoint" [19]. 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., [29, 6]) 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 expected communication behaviour in terms of message exchanges. This intuition is best described in W3C words [19]:

> *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 [20] 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 [22] that synthesise choreographies from local specifications. This makes choreography models more flexible (for instance, choreographies have been exploited in [21] as a contract model for service composition). Note that – adapting the terminology of [20] – we use communicating machines as *automata-oriented* choreography, as in [22].

The concept of orchestration is more controversial. We adopt a widely accepted notion of orchestration [10, 2, 28] nicely described by Ross-Talbot's as [31]:

> *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 choreography 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 [29]:

> *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 [25]. 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 [22], 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 [30, 33] 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 at will (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
Architectural Design

Chapter 2
Software Architectures

To create architecture is to put in order. Put what in order? Function and objects. Le Corbusier

This chapter gives a very basic account of software architectures. There is not enough time in this module to develop this topic properly, therefore the best we can do is to touch upon the most important concepts of the field and leave the reader free to satisfy their curiosity by following the references provided.Section 4

4 Complexity of software

Modern applications are built by assembling together various software elements (such as off-the-shelf software, libraries, or services), rely on various layers of middlewares, and interact with each other through several types of communication gateways. Surely this rich environment allows developers to focus on their applications, however it also brings in new levels of complexity. To tackle such complexity, *software architectures* have been proposed. A possible way to think of software architectures is to consider them as a level of abstraction between (high-level) requirements and (low-level) implementations. This abstraction allows developers to apply fundamental software engineering criteria like

- de-/composition and reuse of computational elements (for instance, determining pattern of interactions or communications)
- separation of concerns while addressing requirements (for instance, by distinguishing between requirements such as scalability or throughput from e.g., responsiveness of services)
- identification of application-dependent aspects to properly reflect them in implementations (e.g., software elements may reflect organisational aspects).

Another advantage of using software architectures is that they are an effective way of documenting and maintaining software systems. We illustrate this by considering the Erlang programs of Section 2 that we recall in Fig. 1:

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

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

Fig. 1: Two Erlang programs

Both programs can be thought of as the realisation of *configuration*[4] using the architectural elements of the functions `ping` and `pong` of Section 2. It is obvious that the programs make a different use of such components; so we should wonder if both uses are "legitimate". We will see that we can define software architecturess and impose constraints on how to compose architectural elements to specify what "legitimate" means.

As a matter of fact, there is still lack of agreement about a shared definition of software architecture. A (vague) description is the one in [4]

> "software architecture for a system is the structure or structures of the system, which comprise elements, their externally-visible behaviour, and the relationships among them."

[4] What "architectural configuration" means is going to be explained later.

(alternative and equally vague descriptions can be found at `www.sei.cmu.edu/architecture/definitions.html`). Fixing a precise definition of software architectures is out of the scope for us; more simply, we need to frame some of the basic notions of software architectures in the context of distributed applications. This is done in the next section.

5 Architectural elements & distributed applications

An appealing aspect of software architectures when considering distributed applications is the intrinsic separation between *computational* aspects vs *interaction* and *communication*. In fact, common to most definitions of software architectures is the distinction between two different types of architectural elements:

- *components* and
- *connectors*.

There is no precise definition of connectors and components. We will first give some intuition about them; we will see that the acceptation of these elements given in [32] naturally corresponds to useful principles for the design of distributed applications.

Intuitively, a component is an element of a system in charge of computation and a connector provides functionalities related to the data communication and control flow among components. A component usually contains functionalities (including data or resource processing, access, and management) while a connector is an element to manage components' interactions in order to specify some types of communication, adaptation, or coordination.

Example 5.1. In a client-server architecture, clients and servers are the components while the connectors represent the communication middleware. Consider the simple client[5] *below*

[5] This python code should be fairly clear even if you are not familiar with python.

```
import socket
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect(('localhost', 5001))
print 'Enter something (type q or Q to quit))'
data = ''
while (data <> 'Q' and data <> 'q'):
    data = raw_input ('|- ')  # reads a line from input
    client_socket.send(data)
    print '-| ' + client_socket.recv(64)
client_socket.close()
```

which simply reads a string from the standard input and sends it to the simple server below

```
import socket
serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serversocket.bind(('localhost', 5001))
serversocket.listen(5) # become a server socket, maximum 5 connections
connection, address = serversocket.accept()
req = ''
while (req <> 'Q' and req  <> 'q'):
  req = connection.recv(64)
  reply = req + ' is ' + str(len(req)) + ' characters long'
  connection.send(reply)
print 'bye bye'
connection.close
```

which returns the length of the string received by the client. It should be clear what part of the code corresponds to the components; it is probably less clear what part of the code corresponds to the connector! ◇

This intuitive descriptions are too vague; after all, why cannot we regard also communication or control flow as computations? We adopt a more revealing distinction between components and connectors given in [32]:

component: an architectural element providing functionalities *application-dependent* for information processing

connector: an architectural element providing *application-independent* functionalities for handling the communication or control flow of components.

Although this definition makes our notions depend on the application domain, it yields a precise criterion to distinguish components and connectors and to allocate functionalities when designing applications.

Exercise 1 *Suppose that we want to extend the client-server application of Example 5.1 with the following functionalities*

1. *given a string the server returns the list of names of employees of a (given) company*
2. *the server replies to the client on a different socket than the one used to send requests from the client to the server*

Say if functionalities (1) *and* (2) *belong to a component or to a connector.*

The notion of connector and its role in a software architecture may be not as clear as the ones of components. According to our definition, connectors incorporate application-independent functionalities required for the coordination of components. Roughly, connectors are used to facilitate the interoperability of components. This means that the functionalities that connectors offer may be quite diversified and related to

- data flow (e.g., the data for a request from the client have to be sent to the server)
- control flow (e.g., the occurrence of an exception must trigger the execution of another component)
- data transformation (e.g., XML documents must be transformed in HTML)
- persistence (e.g, data must be stored in a no-SQL data base)
- load balancing (e.g., requests from clients must be dispatched to the less loaded server)
- etc.

Also, observe that it is often hard to map an architecture to some code realising it. Even the code of simple application of Example 5.1 has a tangible correspondence between components in the architecture and code, while the connector element "materialises" in the code only implicitly being it scattered through the components' lines of code.

Such considerations show that there is a wide range of possible types of connectors; nevetherless, we can classify connectors for

1. data exchange / communication
2. control flow management

Examples of connectors' types (at different levels of abstraction) are

- for (1) procedure calls, shared memory, DBMS, data adapters, communication middlewares, ...
- for (2) task dispatchers, transactional functionalities, wrappers, ...

to mention but a few.

We conclude by spelling out the architectural elements of the Erlang example in Section 2.

Example 5.2. Any process executing the two functions is, according to our definition, a component; in fact, processes progress and compute[6] *in a rather ad-hoc way. The connectors are the communication channels specified in the semantics of the language.* ◇

[6] The example is very basic: the functions just 'compute' strings to print on the screen.

Chapter 3

A simple notation for software architectures

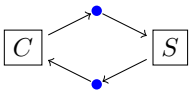
Form follows function. Louis Sullivan

The basic concepts of software architecturess can be easily expressed with the so-called "box&arrow" notation. This chapter describes such notation and illustrates how to use it. Finally, the chapter discusses important features that the software architect should bear in mind when designing systems.

6 Representing software architectures

There are many architectural languages to express software architectures. We do not have time to introduce a full-fledged architectural language and will use a graphical notation therefore we adopt a typical "box-and-wire" graphical notation. More precisely, components are represented as boxes and connectors are circles; these elements are then linked together by "wires".

Example 6.1. A graphical representation of a client-server architecture (cf. Example 5.1) where requests and response ports are different is given by the following diagram



(1)

that represents a client C and a server S interacting through two connectors (represented as bullets). ◇

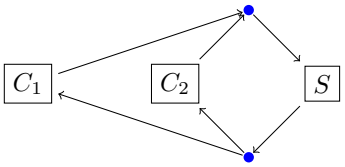
Exercise 2 *What is the software architectures of the first Erlang program of Section 2?*

The diagrams we use to specify software architectures are informal and require a little ingenuity to be understood. For instance, the connector in Example 6.1 may be realised in many different ways (e.g., they can be TCP/IP connections, complex protocols, or communication middlewares like RMI, or a MOM). Although diagram (1) does not explicitly specify the communication pattern of the architecture, one could imagine that a connector is used by the client C to make a request to the server while the other is used by the server S for the response. Also, such diagrams do not specify e.g., where data are stored or where computation happens and common sense must be exercised. For instance in a client-server architecture typically data are on the server side and computations may happen on the client side or on other servers (if e.g., thin-clients are used).

7 Design, runtime, and style

An important concept in software architectures is the notion of *architectural configuration*. This concept refers to the structural layout software elements can actually assume at runtime. For instance, in a client-server architecture typically many clients can make requests to a server. This is not usually specified in the diagram describing the architecture as the one in Example 5.1.

Example 7.1. Using the same graphical notation of software architectures, a configuration with two clients connected to the same server of the client-server architecture of Example 5.1 could be depicted as:



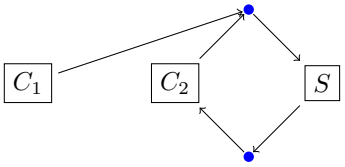
The fact that clients share the same connectors is a design choice and may depend on the nature of the connectors and/or be an application-dependent choice. ◇

Exercise 3 *Is the architecture in diagram* (1) *of Example 6.1 valid for the application in Example 5.1? Briefly justify your answer.*

Exercise 4 *Consider the client-server architecture in Example 5.1. Assume that at each new request from a client the server replies with the reference to a new connector where further interactions with the client take place.*
Draw a diagram for a configuration with two clients, one that is connected to the server but has not made any request yet, and another one that has made a request.

Although we will not deal with runtime configurations and their consistency with software architectures, consistency is an important property that one should guarantee. It is important to observe that at runtime such consistency can be lost as shown in the following example.

Example 7.2. In the configuration of Example 7.1 the connection between C_1 and the response connector may fail leading to the configuration



which is no longer consistent with the architecture in Example 5.1. ◇

Typically, if consistency between configurations and their architecture is lost, re-configurations are necessary (and can be applied manually or (semi-)automatically).

7.1 Balancing coupling & cohesion

Modern distributed applications have two specific characteristics: they are *open* and *dynamic*. This means that applications are designed and implemented to be composed with other applications (openness) and that some parts may be changed (dynamism). All this may possibly happen on-the-fly (i.e., at runtime)! A paradigmatic example is offered by applications developed according to a service-oriented architecture where functionalities are envisaged as *services* to be invoked and applications are built by binding (i.e., composing) services together.

Such kind of applications require a suitable architectural design. More precisely, a good software architecture for open and dynamic applications should minimise *coupling* among components while maximising the *cohesion* of each module. But what do we mean by 'coupling' and 'cohesion'?

coupling can be understood as the degree of dependency among components; an application is loosely coupled when each component does not rely on how other components are implemented. For instance, a service in service-oriented computing should only rely on the interfaces of the other services it invokes and not on how they are implemented.

cohesion is the degree of "similarity" of the functionalities offered by each component. A low cohesive application is an application where there are components offering functionalities which are not very related to each other. For instance, developing an application in the cloud offering a data warehousing service developed as a single component that gives access to databases, offers reporting functionalities, and data analysis functionali-

ties would have low cohesion. A high cohesive design would instead have (at least) three components, each one grouping the different types of functionalities.

Exercise 5 *Consider the diagrams below*

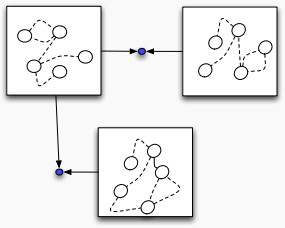


Diagram A

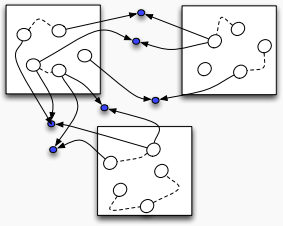


Diagram B

where boxes represent components, circles represent their functionalities, dashed lines represent similarities among functionalities, and solid headed-lines represent dependencies among components induced by their interactions.
For each diagram above say if it represent a low/high coupled application and a low/high cohesive application.

7.2 A matter of style

The design of an application requires many decisions to be taken. Some of those decisions form what is called an *architectural style*. Those decisions are the ones that

- can be applied in a given development context
- add some constraints to the architectural structure
- distill some specific properties of the interactions

A style can be thought of as a "type for architectures", namely an abstract description of classes of architectures that obey the rules dictated by the style. There are three basic ingredients in a style:

- a *vocabulary*,
- a set of *structural constraints*, and
- a description of the *interaction invariants* of the architectural elements.

We now define these concepts.

The vocabulary of a style identifies the types/roles of architectural elements. For instance, the vocabulary of the style of the client-server architecture of Example 5.1 would specify a type for each component, say `client` and `server`, and a type for each connector, say `request` and `response`. Typically, such types identify specific roles of components (e.g., services, controllers, presenters, etc.) and of connectors (e.g., stream manipulators, events, mechanisms to control execution flows, etc.).

The structural constraints specify restrictions on the topological structure of the architectures obeying the style. For instance, a style for a client-server architecture may impose that no more than two components of type `client` can be connected to a connector of type `request` or that a connector of type `response` is connected to only one component of type `server`, or else that interactions are always triggered by `client` components.

Finally, the style specifies invariants that interactions must preserve. For instance, in the style of a client-server architecture one could impose the invariant that interactions are one-way, e.g., connectors of type `request` "go" from components of type `client` to components of type `server` and, vice versa, connectors of type `response` "go" from components of type `server` to components of type `client`.

There are many styles in the literature; we consider few styles suitable for distributed applications. In particular,

- client-server[7]
- pipe-filter
- blackboard

We discuss such styles by considering the running example of a (simple) online multi-player game.

Example 7.3. The game consists of an unspecified number of players each controlling a spaceship with which they explore an unknown galaxy in search of a planet where human beings could live. Players control the flight of their spaceship (throttle of the engine, fuel level, altitude and speed) and communicate with other players by sending them information about the planets they explore. ◇

7.2.1 Client-Server style

This is probably the simplest style for modelling this kind of applications (and possible for any distributed application).
Vocabulary. The types of architectural elements include

- `game-master`
- `player`
- `game-bus`
- `player-bus`

(the names of the types, although immaterial, are reminiscent of the functionalities and role of each element).

> **Exercise 6** *For each type in the above vocabulary say if it is a type for a component or a connector.*

[7] Due to historical reasons, the term 'client-server' is unfortunately overloaded and can mean either a software architecture or an architectural style

Structural constraint. There is a single `game-master` while there is an arbitrary number of `player` elements. Connectors are TCP/IP connections and are shared by only one instance of each type of component (point-to-point connections).

Interaction invariants. The communication among elements of type `player` happens through a connection controlled by elements of `game-master`.

Exercise 7 *Using our informal graphical notation, draw an architecture that has the style described above.*

7.2.2 Pipe-filter style

In this style components, called *filters*, transform streams of inputs into streams of outputs. The output data streams are delivered to other filters by the connectors, called *pipes*.

Example 7.4. The pipe-filter style is a core feature of Linux and it is heavily used shell programming. The components are Linux commands that input data from `stdin` *and send output to* `stdout`*. For example, the execution of the Linux pipe*

```
ls -l | wc -l
```

returns the number of files in a directory, in fact

- *the command* `ls -l` *lists all the files in the current directory line-by-line*
- *the command* `wc -l` *counts the lines in a text*
- *and the pipe symbol* `|` *connects the* `stdout` *from* `ls -l` *to the* `stdin` *of* `wc -l`*.*

◇

An important invariant of this style is that filters have to be completely independent (e.g., they cannot share information).

Vocabulary. Ignoring the inter-player communications, the types of architectural elements for the application in Example 7.3 include

- `game-master`
- `player`
- `game-bus`

> **Exercise 8** *For each type in the above vocabulary say if it is a filter or a pipe.*

Structural constraint. There is a single `game-master` while there is an arbitrary number of elements of type `player` to be connected according to the pipe-filter style. A more specific constraint could be that player/server have a dedicated (pair) of pipes they are connected to.

Interaction invariants. Players continuously sends data to `game-master` that sends back the new game scenario when updates are necessary.

7.2.3 Blackboard style

This style is very convenient when a high degree of loose coupling is necessary. The style features a special type of connector, called *blackboard*, which can be thought of as a central data-structure accessed by other components. In general, in this style connectors specify very abstract access policies to the data-structure.

Vocabulary. The types of architectural elements may include

- `ship-control`
- `flight-blackboard`
- `communication-blackboard`
- `scenario-updater`
- `display`
- `game-master`

Structural constraint. There is a single `game-master` while there is an arbitrary number of `player` elements. For each player element there is a single `ship-control`, a single `display`, and a single `scenario-updater` element all connect through a `flight-blackboard`. `display` elements also connect with players through a `communication-blackboard`.

Interaction invariants. The communication among elements happens by storing and removing data from blackboards. Components may access blackboard any time and data has to be time-stamped so to allow the most update information to be processed.

Exercise 9 *Using our informal graphical notation, draw an architecture that has the style described above.*

Part III

From Global Specifications...

Chapter 4

What is a choreography?

Think global, ...

Before venturing in the technical and formal details of our models, 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 and discusses the quest for more precision.

8 An intuitive account

Let us consider again the excerpt from [19] (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** solutions 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:*

(2)

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 [17], programs [8], graphs and automata [22, 9], executable models [19, 3], 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 (2); 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 allow 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:

if *cond*(global artefact) **then** *behave*(*projection*(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, is not 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 messages: all sent messages are eventually received.

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

Exercise 10 *Does the Erlang program obtained by replacing* `3` *with* `-1` *in the first program of Section 2 gracefully terminate? Briefly justify your answer.*

Exercise 11 *Consider the* `ping` *and* `pong` *functions of Section 2. Is the second Erlang program of Section 2 well-behaved? Briefly justify your answer.*

We will first consider a (semi-)formal language to represent global specifications. Such language resembles UML's sequence diagrams (actually, it is similar to message sequence charts [15]).

9 Global views as interaction diagrams

Global choreographies can be represented using *sequence diagrams* [24] (see also `http://www.uml-diagrams.org/sequence-diagrams.html`). A sequence diagram is an interaction diagram describing message communications among some components. Graphically, components are represented as *headed lifelines* as depicted in Fig. 2.

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*

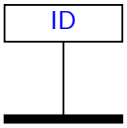


Fig. 2: Lifeline

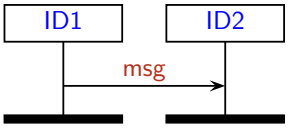


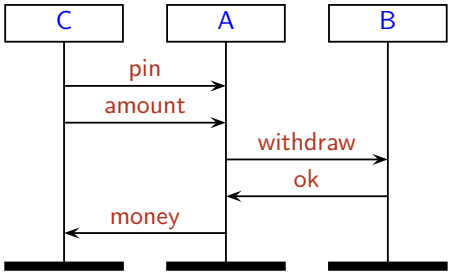
Fig. 3: Asynchronous call

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[8]*, namely sequential components that execute distributively. Hereafter, we use the terms 'component' and 'participant' interchangeably.*

For us, a participant could be either a human being interacting with the system 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. 2), we may omit the class name of participants from the lifeline head of our sequence diagrams. Our basic assumption is that interaction is possible 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 9.1. A choregraphy for a simplified ATM-application can be depicted as follows:

[8] 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.



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 9.1:

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 choreography in Example 9.1, 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 but optionally they could be explicitly represented. For instance, in the first interaction between C and A of the choreography in Example 9.1, 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 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 A B!m denote that participant A sends participant B the message m and let 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).*

10 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[9] not represented in the diagram).

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.

[9] What do we mean by "local decision"? Or more generally 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 localstate state of A.

Example 10.1. It is easy to verify that, for Example 9.1,

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. ◇

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 12 *Give all the sequences of communication actions which respect the order in Example 10.1.*

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

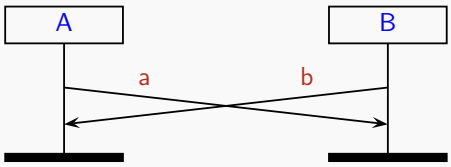
C A!pin; C A!amount; C A?amount; C A?pin; A B!widthraw;
A B?widthraw; B A!ok; B A?ok; A C!money; A C?money

is not a correct answer to Exercise 12.

Although appealing for their intuitive clarity, sequence diagrams may look odd. The following exercise is meant to highlight this.[10]

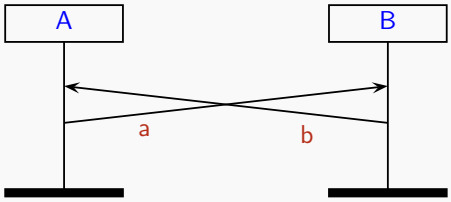
[10] Exercise 13 is tricky because our execution model is not fully described yet; the exercise should be doable with some ingenuity, but do not be worry if you cannot solve it, you will manage after having studied the whole chapter.

Exercise 13 *Is the choreography below executable with asynchronous communications?*



Justify your answer.

And what about this one?



Reflect on "how time flows".

10.1 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 10.2. For the participants of the first choreography in Exercise 13

A B!a; B A?b *and* B A!b; A B?a

are the programs of participants A *and* B *respectively.*[11] ◇

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

- 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.

10.2 Running sequence diagrams

The third and last step finally answers the question: How do we execute sequence diagrams? Or, putting it more precisely, how do we describe the execution*s* of sequence diagrams? As said before, we use traces for this. In-

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

tuitively, a trace describes a possible run[12] 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 $\mathsf{A}_{(i)}$ when the "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 10.3. For participant A *in Example 10.2,*

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

It is a simple observation that $\mathsf{A}_{(0)}$ *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 $\mathsf{m}_1 \cdots \mathsf{m}_k$ it contains (with m_1 the top of the queue and m_k the last message of the queue.

[12] 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.

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 $\mathsf{A}_{(i)}$ 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 10.4. The initial configuration of the sequence diagram in Example 10.2 and the one after A *executed its output are respectively written as*

$$\langle \mathsf{A}_{(0)}, \mathsf{B}_{(0)}, \overset{\mathsf{A\,B}}{[\,\vdots\,]}, \overset{\mathsf{B\,A}}{[\,\vdots\,]} \rangle \qquad \textit{and} \qquad \langle \mathsf{A}_{(1)}, \mathsf{B}_{(0)}, \overset{\mathsf{A\,B}}{[\,\mathsf{a}\,\vdots\,]}, \overset{\mathsf{B\,A}}{[\,\vdots\,]} \rangle$$

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:

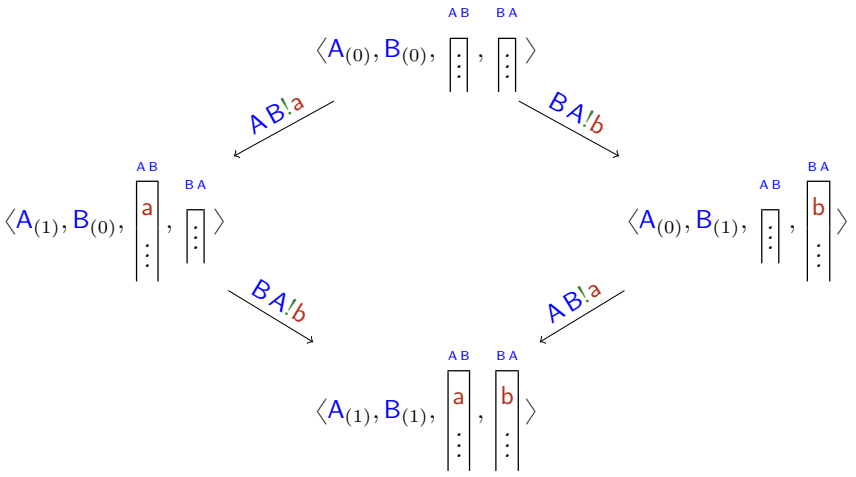
1. choose a configuration, say $\mathfrak{C}$, that has not been explored yet (initially, $\mathfrak{C}$ can only be the initial configuration)
2. for all the enabled action α in a program $\mathsf{A}_{(i)}$ in $\mathfrak{C}$, create a configuration $\mathfrak{C}'$ obtained from $\mathfrak{C}$ by
   - 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 10.5. Building the execution graph of the choreography Exercise 13 we get



Note that this is not the whole graph! ◇

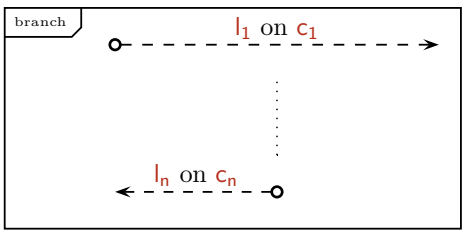
Exercise 14 *Complete the execution graph of Example 10.5.*

11 More expressiveness

As said, the choreography in Example 9.1 is very basic. Usually, we need to add constructs to express more interesting (and precise) choreographies. For instance, one would be able to express what happens when C enters the wrong pin or tries to withdraw more money than those on her account. We develop our language of sequence diagrams to cope with this issue. hereafter, we adopt the channel-based interpretation (cf. page 48).

11.1 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:

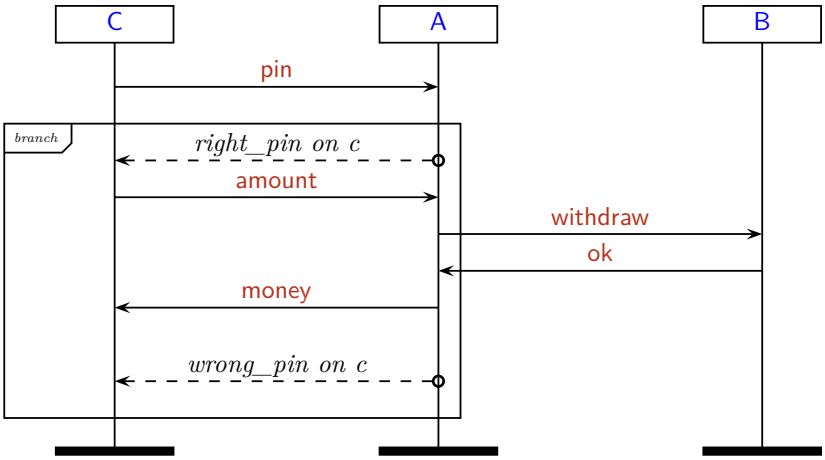


In the branching block above, *labels* $l_1, \ldots, l_n$ are sent by a participant – the one whose lifeline intersects the circle (not represented in the picture above) – on channel c to a partner – the one pointed by the empty-headed arrow – to communicate how the choregraphy should proceed.

Remark 9 *The 'on c' part specifies on which channel the selection/branching is made; we can 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 9.1 and show how branching caters for more expressiveness.

Example 11.1. 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 11.1 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.*

11.2 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, in concurrent and distributed computations the control flow is not unique; in fact, by definition there may be many parts of a system that concurrently execute. This makes non-determinism unavoidable (besides being sometimes desirable) in distributed applications. Also, the interplay between distributed choice and non-determinism may lead to problems in distributed choreographies.

11.2.1 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 interac-

tions with other participants (in our framework, this means that A has to wait for inputs from other participants).

Example 11.2. In the branch of the choreography of Example 11.1, participant A *makes an internal choice when she decides to send either label* right *or label* wrong*; 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).* ◇

11.2.2 Non-determinism

In general a non-deterministic computation is a computation where at least one step of execution is not a function of the state 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 with-

out 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 11.1 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 15 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″.

11.3 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 choreography in Fig. 4. After A starts by triggering B and C on channels a1 and a2 respectively all the participants engage in a choice so that

- if A sends label l1 to C then B sends something on b to C
- if instead B sends label l2 to C then C sends something on c to A.

In a distributed asynchronous execution, this may lead to a deadlock. In fact, in the branch A may (locally) decide to either communicate label l1 to C and then terminate, or wait for C to send something on channel c. Similarly, after receiving on a2, B (locally) decides whether to send a label to C and terminates or to send a message on channel b. If both A and B choose to send the label to C, they will both terminate hence, whatever branch C takes, the

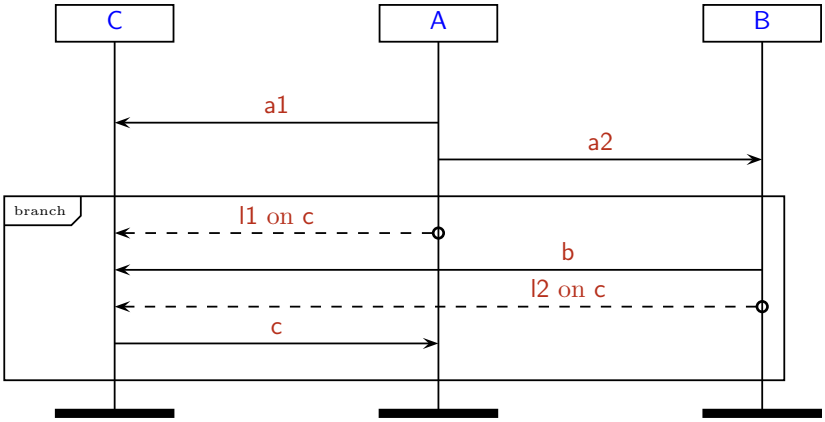


Fig. 4: The "unique selector" problem

communications on b or c will not be completed as the message sent by C will not be received.

Exercise 15 *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 11.3. Deadlock configurations can be found by building the execution graph from the sequence diagram. Consider the choreography in Fig. 5 under the communication-flow interpretation. Intuitively, a deadlock occurs when

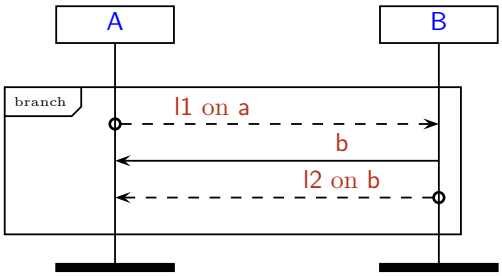
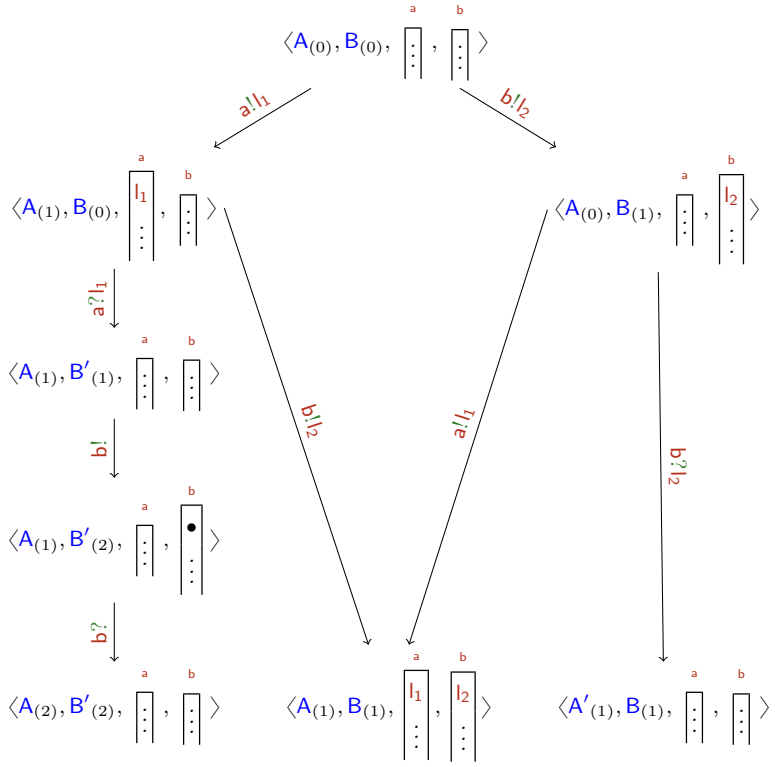


Fig. 5: Deadlock issue

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. 6 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.

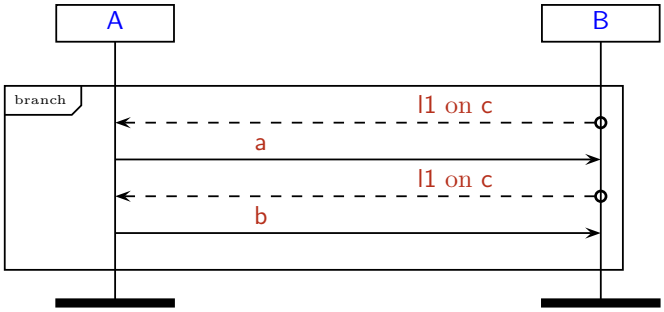
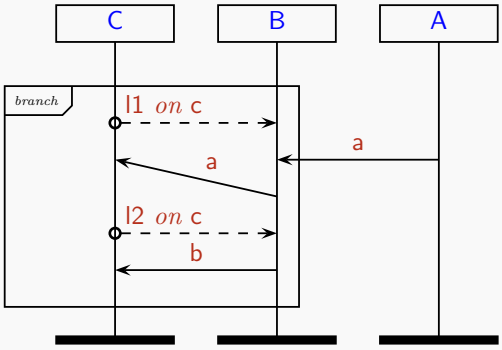


Fig. 6: Determinacy issue

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 11.1; there B was 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 16 *Modify the choreography in Example 11.1 so that the un-notified problem is solved.*

Exercise 17 *Which of the conditions of well-behaviour the second choreography of Exercise 13 and the choreography below violate.*



Chapter 5

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 models with precise syntax and semantics. There are many of such models and we decided to consider only one of them based on graphs (for global views) and automata (for local views). An advantage is that our model is at the same time quite expressive and close to other design languages such as BPMN or UML's state machines.

12 Global views as graphs

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. In the global view, this is modelled with *interactions* $\mathsf{A}\to\mathsf{B}\colon \mathsf{m}$, which represent the fact that participant A sends message m to participant B, which is expected to receive m.

Definition 1 (g-choreography). The set $\mathcal{G}$ of *global choreographies* (g-choreography for short) consists of the terms G derived by the grammar

$$\mathsf{G} ::= \mathsf{A} \xrightarrow{\mathsf{i}} \mathsf{B} : \mathsf{m} \tag{3}$$

$$\mid \mathsf{G} ; \mathsf{G}' \tag{4}$$

$$\mid \mathsf{i}.(\mathsf{G} \mid \mathsf{G}') \tag{5}$$

$$\mid \mathsf{i}.(\mathsf{G} + \mathsf{G}') \tag{6}$$

$$\mid \mathsf{i}.(*\mathsf{G}@\mathsf{A}) \tag{7}$$

that satisfy the following conditions:

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

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

A g-choreography can be a simple interaction (3), the sequential (4) or parallel (5) composition of g-choreographies the choice between two g-choreographies (6), or the iteration of a choreography (7).

Control points i tag interaction and the non-deterministic or parallel composition of g-choreographies. In the following, we may omit control points when immaterial, e.g., writing $\mathsf{G} + \mathsf{G}'$ instead of $\mathsf{i}.(\mathsf{G} + \mathsf{G}')$. 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:

$$\mathsf{1}.(\mathsf{A} \xrightarrow{\mathsf{2}} \mathsf{B} : \mathsf{m} \mid \mathsf{C} \xrightarrow{\mathsf{3}} \mathsf{D} : \mathsf{y}) \qquad \text{and} \qquad \mathsf{5}.(\mathsf{A} \xrightarrow{\mathsf{3}} \mathsf{B} : \mathsf{m} \mid \mathsf{C} \xrightarrow{\mathsf{2}} \mathsf{D} : \mathsf{y})$$

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,

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

are equivalent.

Exercise 18 *Give a g-choreography corresponding to the choreography in Exercise 16.*

Exercise 19 *Give a global graph equivalent, but different from the one in Exercise 18.*

The syntax in Definition 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" and a single "exit" control points called *source* and *sink* respectively. Fig. 7 illustrates our graphical notation and,

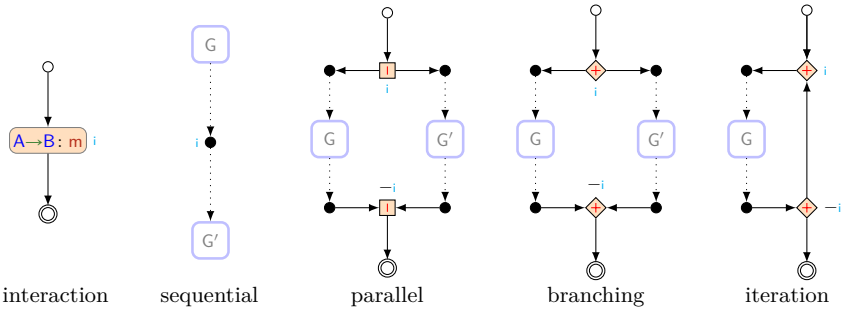
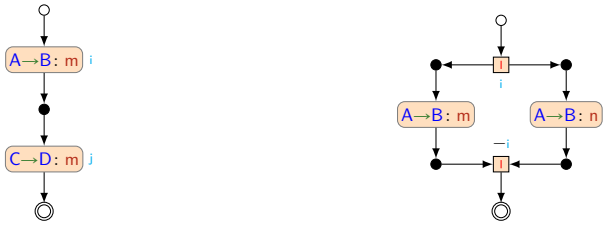


Fig. 7: A graphical notation for g-choreographies

before commenting it, a remark is worthwhile: each fork or branch gate with control point i in our pictures will have a corresponding join and merge gate with control point $-\mathsf{i}$; 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. 7 $\circ$ and $\circledcirc$ respectively denote the source node the sink node; other nodes are drawn as $\bullet$ and a dotted edge from/to a $\bullet$-control points singles out the source/sink control point the edge connects to. More precisely, a dotted edge from $\bullet$ to a boxed G means that $\bullet$ is the



(a) $\mathsf{G}_{(8a)}$, a sequential composition

(b) $\mathsf{G}_{(8b)}$, a parallel composition

Fig. 8: Examples of choreographies

source of G; similarly, a dotted edge from a boxed G to $\bullet$ means that $\bullet$ 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, $\bullet$ 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 $\mathsf{G} \in \mathcal{G}$, each control point i 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 $-\mathsf{i}$ marking the merge/join point of the execution flow. Labels will not be depicted when immaterial.

Figs. 8a and 8b 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. 8a 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 $\rightarrow \bullet \rightarrow$ in a graph does not yield any important event to trace and in the following we will often replace it with a simple edge $\rightarrow$). Note that the graph[13] $\mathsf{G}_{(8b)}$ in Fig. 8b (where the control points of interactions are omitted for readability) represents a choreography where A sends B messages m and n in any order.

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

Akin to BPMN [27] diagrams, our graphs 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 shave a special standing in our visual notation.

Our graphs resemble the ones in [9, 22] the only differences being that

- by construction, forking and branching control points i have a corresponding join and merge control point $-_{\mathsf{i}}$;
- there is a unique sink control point with a unique incoming edge (as in [9, 22], there is also a unique source control point with a unique outgoing edge).

Example 12.1. The graph in Fig. 8b represents a choreography where A *sends* B *messages* m *and* n *in any order.* ◇

Exercise 20 *Give a graph corresponding to the g-choreography in Fig. 5.*

Exercise 21 *Give an expression in the syntax of g-choreographies for the graph in Fig. 8b.*

Exercise 22 *Add control points to the expression that you gave as a solution of Exercise 20 so that the resulting term is in* $\mathcal{G}$*.*

13 Semantics of g-choreographies

The semantics of a choice-free g-choreography $\mathsf{G} \in \mathcal{G}$ (i.e. a 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 $\mathsf{i}.(\mathsf{G} + \mathsf{G}')$ 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.

13.1 Basic definitions

Communication actions happen on channels, which we identify by the names of the participants involved in the communication. Formally, a channel is an element of the set

$$\mathcal{C} \quad := \quad \{(\mathsf{A}, \mathsf{B}) \mid \mathsf{A}, \mathsf{B} \in \mathcal{P} \quad \text{and} \quad \mathsf{A} \neq \mathsf{B}\} \tag{8}$$

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

$$\mathcal{E} \quad := \quad \mathcal{E}^{!} \cup \mathcal{E}^{?} \cup \mathbb{Z}_0 \tag{9}$$

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

Sets $\mathcal{E}^{!}$ and $\mathcal{E}^{?}$, the output events and input events, respectively represent *sending actions* and *receiving actions*. We shorten $(\mathsf{A}\,\mathsf{B}, !, \mathsf{i}, \mathsf{m})$ as $\mathsf{A}\,\mathsf{B}!^{\mathsf{i}}\mathsf{m}$ and $(\mathsf{A}\,\mathsf{B}, ?, \mathsf{i}, \mathsf{m})$ as $\mathsf{A}\,\mathsf{B}?^{\mathsf{i}}\mathsf{m}$. 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. 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

$$\mathsf{sbj}(\mathsf{A}\,\mathsf{B}!^{\mathsf{i}}\mathsf{m}) = \mathsf{A} \qquad \text{and} \qquad \mathsf{sbj}(\mathsf{A}\,\mathsf{B}?^{\mathsf{i}}\mathsf{m}) = \mathsf{B}$$
$$\mathsf{act}(\mathsf{A}\,\mathsf{B}?^{\mathsf{i}}\mathsf{m}) = \mathsf{A}\,\mathsf{B}?\mathsf{m} \qquad \text{and} \qquad \mathsf{act}(\mathsf{A}\,\mathsf{B}!^{\mathsf{i}}\mathsf{m}) = \mathsf{A}\,\mathsf{B}!\mathsf{m}$$

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 input or output 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, so $\mathsf{cp}(e)$ denotes the control point of an event e.

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

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

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

13.2 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 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 a vertexes, that is: if $h = (\!|\, E, E' \,|\!)$ then $\mathsf{cs}(h) = E$ and $\mathsf{ef}(h) = E'$. Given $H, H' \subseteq 2^{\mathcal{E}} \times 2^{\mathcal{E}}$, define the hypergraph

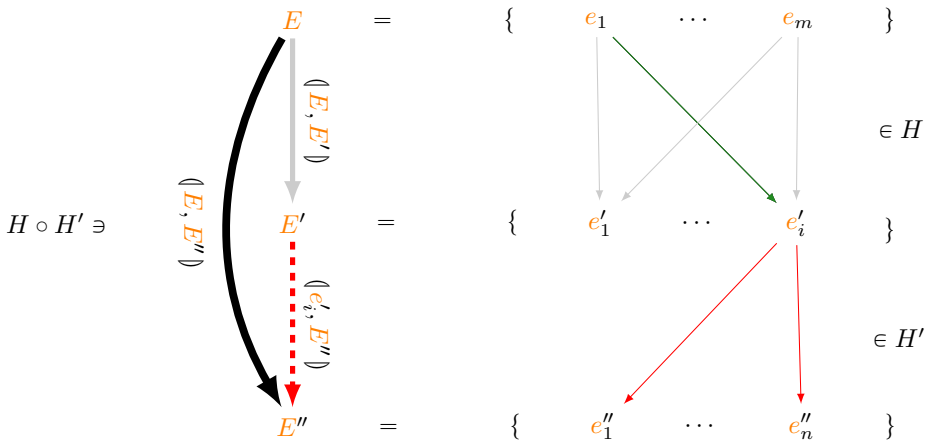


Fig. 9: A happens-before relation

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

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 13.1. Fig. 9 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

$$(\!| E_0, E_1 |\!), (\!| E'_1, E_2 |\!), \ldots, (\!| E'_k, E'_{k+1} |\!) \tag{10}$$

is a *chain* if $E_i \cap E'_i \neq \varnothing$ 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* (10)*) 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 2. 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'.

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

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

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

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.

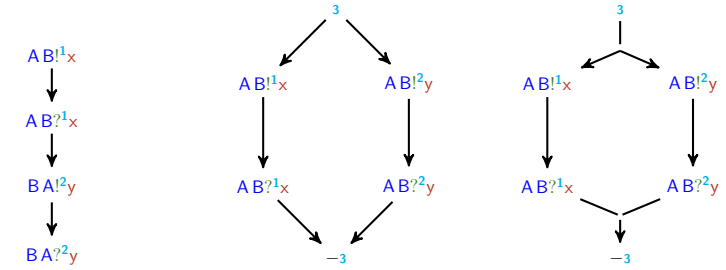
Exercise 23 *Give two happens-before relations H and H' such that $H \circ H'$ is not a happens-before relation.*

Example 13.2. Some happens-before relations are depicted in Fig. 10; relation $H_{(10a)}$ and $H_{(10b)}$ contain only simple arcs, while the relation $H_{(10c)}$ contains two hyperedges:

$$(\!| 3, \{\mathsf{A}\,\mathsf{B}!^1\mathsf{x}, \mathsf{A}\,\mathsf{B}!^2\mathsf{y}\} |\!) \qquad \textit{and} \qquad (\!| \{\mathsf{A}\,\mathsf{B}?^1\mathsf{x}, \mathsf{A}\,\mathsf{B}?^2\mathsf{y}\}, -3 |\!)$$

Intuitively,

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



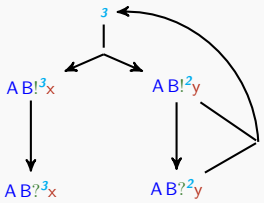
(a) $H_{(10a)}$ (b) $H_{(10b)}$ (c) $H_{(10c)}$

Fig. 10: Some happens-before relations

- $H_{(10a)}$ *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.*
- $H_{(10b)}$ *represents a choice at control point* 3 *between the left and the right branch; basically after* 3 *happens either one but not both of its successors happens.*
- $H_{(10c)}$ *represents the parallel execution of two threads at the control point* 3*; note that the hyperedge* $(\!| 3, \{\mathsf{A\,B!}^1\mathsf{x}, \mathsf{A\,B!}^2\mathsf{y}\} |\!)$ *of* $H_{(10c)}$ *relates the event* 3 *to both* $\mathsf{A\,B!}^1\mathsf{x}$ *and* $\mathsf{A\,B!}^2\mathsf{y}$ *"at the same time", which formalises the idea that* 3 *has to happen before both its successors and both successors happen.*

Note the correspondence among the control points in $H_{(10b)}$ *and in* $H_{(10c)}$*.* ◇

Exercise 24 *Consider the following hypergraph*



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

13.3 Composing happens-before relations

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

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

Example 13.3. With reference to Fig. 10, we have $\mathsf{min}\, H_{(10b)} = \mathsf{min}\, H_{(10c)} = \{3\}$ *and* $\mathsf{max}\, H_{(10b)} = \mathsf{max}\, H_{(10c)} = \{-3\}$*, while the minimal and maximal elements of* $H_{(10a)}$ *are* $\mathsf{A}\,\mathsf{B}!^{1}\mathsf{x}$ *and* $\mathsf{B}\,\mathsf{A}?^{2}\mathsf{y}$ *respectively.* ◇

> **Exercise 25** *Give the sets of maximal and minimal events for the graph in Exercise 24.*

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 $(\!|\, \mathsf{A}\,\mathsf{B}!^{\mathsf{i}}\mathsf{m}, \mathsf{A}\,\mathsf{B}?^{\mathsf{i}}\mathsf{m} \,|\!)$ for each interaction $\mathsf{A} \xrightarrow{\mathsf{i}} \mathsf{B} : \mathsf{m}$ 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

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

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 13.4. With reference to Fig. 10, we have

$$\mathsf{fst}\, H_{(10a)} = \{(\!|\, \mathsf{A\,B!}^{1}\mathsf{x}, \mathsf{A\,B?}^{1}\mathsf{x} \,|\!)\} \qquad \textit{and} \qquad \mathsf{lst}\, H_{(10a)} = \{(\!|\, \mathsf{B\,A!}^{2}\mathsf{y}, \mathsf{B\,A?}^{2}\mathsf{y} \,|\!)\}$$

while $H_{(10b)}$ *and* $H_{(10c)}$ *have the same set of "first" and the "last" communication actions (*$\mathsf{fst}\, H_{(10b)} = \mathsf{fst}\, H_{(10c)} = \{(\!|\, \mathsf{A\,B!}^{1}\mathsf{x}, \mathsf{A\,B?}^{1}\mathsf{x} \,|\!), (\!|\, \mathsf{A\,B!}^{2}\mathsf{y}, \mathsf{A\,B?}^{2}\mathsf{y} \,|\!)\} = \mathsf{lst}\, H_{(10b)} = \mathsf{lst}\, H_{(10c)}$*).* ◇

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 13.5. Fig. 11 depicts the sequential composition $\mathsf{seq}(H, H')$ *where* $H = \mathsf{A\,B!}^{i}\mathsf{x} \rightarrow \mathsf{A\,B?}^{i}\mathsf{x}$ *and* H' *ranges over the happens-before relations*

$$\begin{array}{ccccc} \mathsf{A\,C!}^{j}\mathsf{y} & \mathsf{B\,C!}^{j}\mathsf{y} & \mathsf{C\,B!}^{j}\mathsf{y} & \mathsf{A\,B!}^{j}\mathsf{y} & \mathsf{C\,D!}^{j}\mathsf{y} \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ \mathsf{A\,C?}^{j}\mathsf{y} & \mathsf{B\,C?}^{j}\mathsf{y} & \mathsf{C\,B?}^{j}\mathsf{y} & \mathsf{A\,B?}^{j}\mathsf{y} & \mathsf{C\,D?}^{j}\mathsf{y} \end{array} \tag{11}$$

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

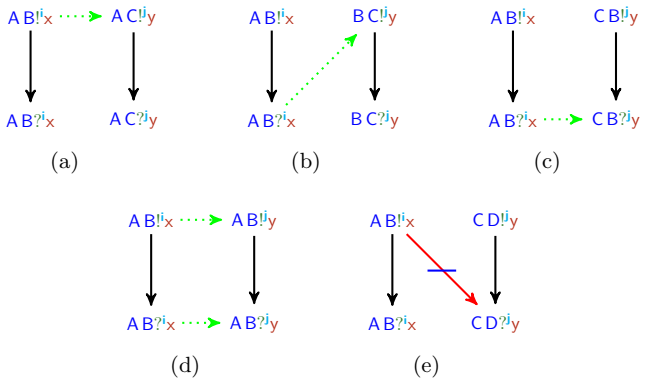


Fig. 11: Examples of sequential composition

14 The semantics of graphs

The semantics of g-choreography is the partial map $[\![\_]\!]$ that given a g-choreography G returns a happens-before relation, provided that G is a "good" g-choreography. Following [14], function $[\![\_]\!]$ is defined as per the clauses from Eqs. (12) to (17) below.[15]

The semantics of an interaction $\mathsf{A}\xrightarrow{\mathsf{i}}\mathsf{B}\colon \mathsf{m}$ is straightforward:

$$[\![\mathsf{A}\xrightarrow{\mathsf{i}}\mathsf{B}\colon \mathsf{m}]\!] = \{(\!|\, \mathsf{A\,B!^{i}m}, \mathsf{A\,B?^{i}m} \,|\!)\} \tag{12}$$

namely, the send part $\mathsf{A\,B!^{i}m}$ of the interaction must precede its receive $\mathsf{A\,B?^{i}m}$ 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 necessary 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:

$$[\![*\mathsf{G@A}]\!] = [\![\mathsf{G}]\!] \tag{13}$$

[15] Here we first give a simplified version of the semantics which is less general than the one defined in [14].

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

$$[\![\mathsf{G};\mathsf{G}']\!] = \begin{cases} \mathsf{seq}([\![\mathsf{G}]\!],[\![\mathsf{G}']\!]) & \text{if } ws(\mathsf{G},\mathsf{G}') \\ \text{undef} & \text{otherwise} \end{cases} \tag{14}$$

where $ws(\mathsf{G},\mathsf{G}') \iff \big(\widehat{\mathsf{seq}([\![\mathsf{G}]\!],[\![\mathsf{G}']\!])}\big)^\star \supseteq \widehat{R}$ with $R := \mathsf{cs}\big(\mathsf{lst}\,[\![\mathsf{G}]\!]\big) \times \mathsf{ef}\big(\mathsf{fst}\,[\![\mathsf{G}']\!]\big)$ is the *well-sequencedness* condition. The semantics of sequential composition $\mathsf{G};\mathsf{G}'$ establishes 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 14.1. The compositions in Fig. 11 correspond to the composition of the semantics of $\mathsf{A}\xrightarrow{j}\mathsf{B}\colon \mathsf{x}$ *with the semantics of*

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

respectively. All those compositions are sound, barred the one in Fig. 11e which violates well-sequencedness; in fact, the stroken edge depicts the missing dependency required by well-sequencedness. ◇

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

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

$$[\![\mathsf{i}.(\mathsf{G} \mid \mathsf{G}')]\!] = \begin{cases} [\![\mathsf{G}]\!] \cup [\![\mathsf{G}']\!] \cup H & \text{if } wf(\mathsf{G},\mathsf{G}') \\ \text{undef} & \text{otherwise} \end{cases} \tag{15}$$

where $H = \{(\!|\, \mathsf{i}, \min [\![\mathsf{G}]\!] \cup \min [\![\mathsf{G}']\!] \,|\!), (\!|\, \max [\![\mathsf{G}]\!] \cup \max [\![\mathsf{G}']\!], -_{\mathsf{i}} \,|\!)\}$ and the side condition $wf(\mathsf{G},\mathsf{G}') \iff \mathsf{act}([\![\mathsf{G}]\!]) \cap \mathsf{act}([\![\mathsf{G}']\!]) \cap \mathcal{L}^{?} = \varnothing$ is 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 26 *Draw the happens-before relation for the semantics of the g-choreography*

$$\mathsf{G} = \mathsf{3}.(\mathsf{A} \xrightarrow{1} \mathsf{B} : \mathsf{x} \mid \mathsf{A} \xrightarrow{2} \mathsf{B} : \mathsf{y}) \tag{16}$$

Justify your answer.

Although similar to the previous two cases, the semantics of non-deterministic choice involves the most complex side condition:

$$[\![\mathsf{i}.(\mathsf{G} + \mathsf{G}')]\!] = \begin{cases} [\![\mathsf{G}]\!] \cup [\![\mathsf{G}']\!] \cup H & \text{if } wb(\mathsf{G},\mathsf{G}') \\ \text{undef} & \text{otherwise} \end{cases} \tag{17}$$

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

Definition 3. 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 14.1 we define active and passive participants (cf. Definitions 4 and 5 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', $[\![\mathsf{i}.(\mathsf{G} + \mathsf{G}')]\!]$ contains those making i (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 $-\mathsf{i}$). 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.

14.1 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 III which is based on the notions of active and passive participant, that respectively single out participants A that do not make an internal choice, namely it is not A selecting 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

$$H^{@\mathsf{A}} = \bigcup_{(\!| E, E' |\!) \in H} \{(\!| E^{@\mathsf{A}}, E'^{@\mathsf{A}} |\!)\}$$

where

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

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 4. 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 $E = \bigcup_{h \in \mathsf{fst}\,[\![\mathsf{G}]\!]^{@\mathsf{A}}} (\mathsf{cs}(h) \cup \mathsf{ef}(h))$ and $E' = \bigcup_{h \in \mathsf{fst}\,[\![\mathsf{G}']\!]^{@\mathsf{A}}} (\mathsf{cs}(h) \cup \mathsf{ef}(h))$. 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 4 (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 27 *Identify the passive participant of the graph in Fig. 10b*

Definition 5. 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 $E = \bigcup_{h \in \mathsf{fst}\,[\![\mathsf{G}]\!]^{@\mathsf{A}}} (\mathsf{cs}(h) \cup \mathsf{ef}(h))$ and $E' = \bigcup_{h \in \mathsf{fst}\,[\![\mathsf{G}']\!]^{@\mathsf{A}}} (\mathsf{cs}(h) \cup \mathsf{ef}(h))$. 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 28 *Identify the active participant of the graph in Fig. 10c. Justify your answer.*

Example 14.2. The following g-choreography:

$$\mathsf{G}_{(10b)} = 3.\big(\mathsf{A} \overset{1}{\to} \mathsf{B}\colon \mathsf{x} \;+\; \mathsf{A} \overset{2}{\to} \mathsf{B}\colon \mathsf{y}\big)$$

has a defined semantics. In fact, the choice in $\mathsf{G}_{(10b)}$ *is well-branched:*

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

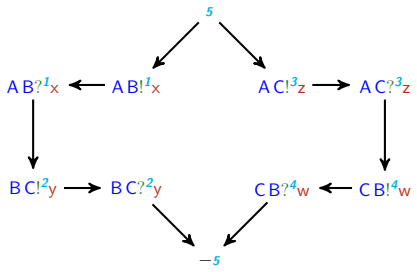
The hypergraph in Fig. 10b is the happens-before relation yielding the semantics of $\mathsf{G}_{(10b)}$. ◇

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

Example 14.3. Consider the following g-choreography:

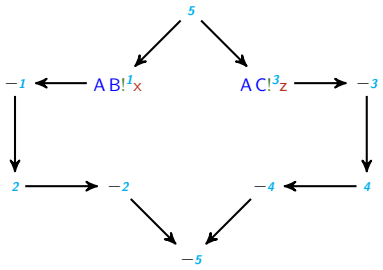
$$\mathsf{G}_{Example\ 14.3} = 5.\Big(\Big(\mathsf{A} \overset{1}{\to} \mathsf{B}\colon \mathsf{x}; \mathsf{B} \overset{2}{\to} \mathsf{C}\colon \mathsf{y}\Big) \;+\; \Big(\mathsf{A} \overset{3}{\to} \mathsf{C}\colon \mathsf{z}; \mathsf{C} \overset{4}{\to} \mathsf{B}\colon \mathsf{w}\Big)\Big) \tag{18}$$

We now verify that the semantics of $\mathsf{G}_{Example\ 14.3}$ *is*



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

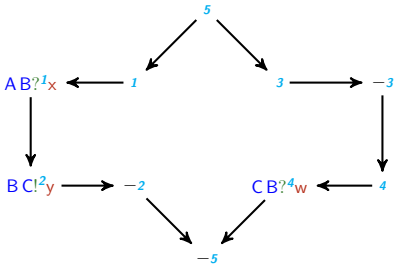
$[\![\mathsf{G}_{Example\ 14.3}]\!]^{@\mathsf{A}} =$



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*

$[\![\mathsf{G}_{Example\ 14.3}]\!]^{@\mathsf{B}} =$



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′*).* ◇

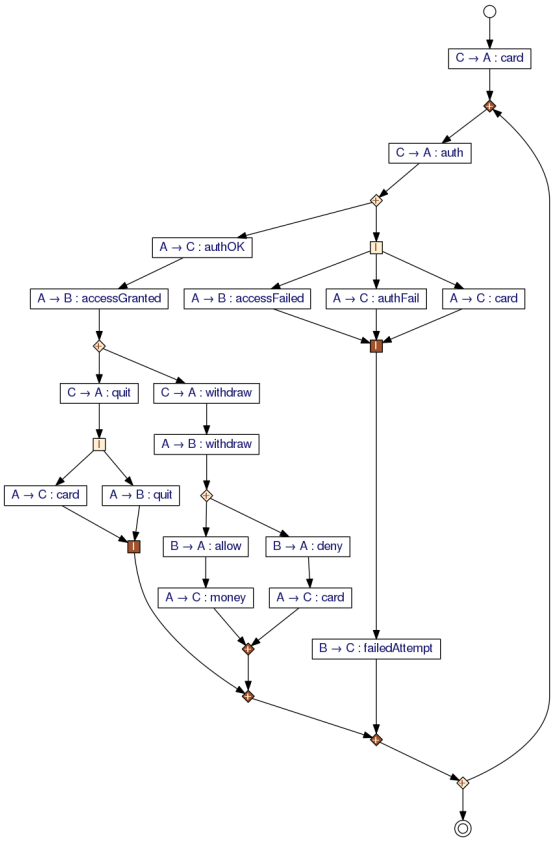
Exercise 29 *Draw the happens-before relation for the semantics of the g-choreography*

$$\mathsf{G} = 3.\big(\mathsf{A} \xrightarrow{1} \mathsf{B} : \mathsf{x} \ + \ \mathsf{A} \xrightarrow{2} \mathsf{C} : \mathsf{y}\big) \tag{19}$$

Justify your answer.

14.2 Applying semantic equations

Let us see how to apply the equations defining 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 11.1 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 semantic equations are defined according to the syntax of g-choreographies, it is better to consider g-choreography corresponding to the global graph above.

[17] control points are omitted for simplicity

$$\mathsf{G}_{\mathrm{atm}} = \mathsf{C}\to\mathsf{A}\colon \mathsf{card}\ ;\ *\mathsf{G}_{\mathrm{auth}}@\mathsf{C}$$

$$\mathsf{G}_{\mathrm{auth}} = \mathsf{C}\to\mathsf{A}\colon \mathsf{auth}\ ;\ 1.(\mathsf{G}_{\mathrm{ok}} + \mathsf{G}_{\mathrm{ko}})$$

$$\mathsf{G}_{\mathrm{ok}} = \mathsf{A}\to\mathsf{C}\colon \mathsf{authOk}; \mathsf{A}\to\mathsf{B}\colon \mathsf{accessGranted}; 3.(\mathsf{G}_{\mathrm{wdw}} + \mathsf{G}_{\mathrm{quit}})$$

$$\mathsf{G}_{\mathrm{wdw}} = \mathsf{C}\to\mathsf{A}\colon \mathsf{withdraw}; \mathsf{A}\to\mathsf{B}\colon \mathsf{withdraw}; 5.(\mathsf{G}_{\mathrm{alw}} + \mathsf{G}_{\mathrm{dny}})$$

$$\mathsf{G}_{\mathrm{alw}} = \mathsf{B}\to\mathsf{A}\colon \mathsf{allow}; \mathsf{A}\to\mathsf{C}\colon \mathsf{money}$$

$$\mathsf{G}_{\mathrm{dny}} = \mathsf{B}\to\mathsf{A}\colon \mathsf{deny}; \mathsf{A}\to\mathsf{C}\colon \mathsf{card}$$

$$\mathsf{G}_{\mathrm{quit}} = \mathsf{C}\to\mathsf{A}\colon \mathsf{quit}; 4.(\mathsf{A}\to\mathsf{C}\colon \mathsf{card} \mid \mathsf{A}\to\mathsf{B}\colon \mathsf{quit})$$

$$\mathsf{G}_{\mathrm{ko}} = 2.(\mathsf{A}\to\mathsf{B}\colon \mathsf{accessFailed} \mid \mathsf{A}\to\mathsf{C}\colon \mathsf{authFail} \mid \mathsf{A}\to\mathsf{C}\colon \mathsf{card}); \mathsf{B}\to\mathsf{A}\colon \mathsf{failedAttempt}$$

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 sub-g-choreographies are well-forked.

We have

$$\begin{aligned}[\![\mathsf{G}_{\mathrm{atm}}]\!] &= \mathsf{seq}([\![\mathsf{C}\to\mathsf{A}\colon \mathsf{card}]\!], [\![*\mathsf{G}_{\mathrm{auth}}@\mathsf{C}]\!]) \\ &= \mathsf{seq}(\underset{\mathsf{C}\ \mathsf{A}?\mathsf{card}}{\overset{\mathsf{C}\ \mathsf{A}!\mathsf{card}}{\downarrow}}, [\![\mathsf{G}_{\mathrm{auth}}]\!]) \qquad \text{if } ws(\mathsf{C}\to\mathsf{A}\colon \mathsf{card}, \mathsf{G}_{\mathrm{auth}})\end{aligned}$$

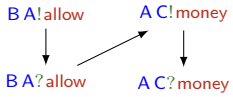
To determine the semantics of $\mathsf{G}_{\mathrm{atm}}$ is therefore necessary to first find out what is the semantics of $\mathsf{G}_{\mathrm{auth}}$. To do this, it is convenient to start ascertaining the semantics of the last four sub-g-choreographies (since they do not depend on any other sub-g-choreography);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

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

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

$$= \mathsf{seq}\Big(\overset{\mathsf{B\,A!allow}}{\downarrow}{}_{\mathsf{B\,A?allow}}\ ,\ \overset{\mathsf{A\,C!money}}{\downarrow}{}_{\mathsf{A\,C?money}}\Big) \qquad \text{if } ws(\mathsf{B}{\to}\mathsf{A}\colon \mathsf{allow}, \mathsf{A}{\to}\mathsf{C}\colon \mathsf{money})$$

$$=$$

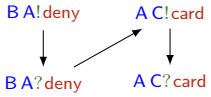


where in the last equation the diagonal arrow is the one added by $\mathsf{seq}(,)$, which makes the side condition $ws(\mathsf{B}{\to}\mathsf{A}\colon \mathsf{allow}, \mathsf{A}{\to}\mathsf{C}\colon \mathsf{money})$ hold. Basically, We have to verify that the order induced by the happens-before relation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

$$\mathsf{cs}\Big(\mathsf{lst}\ \overset{\mathsf{B\,A!allow}}{\downarrow}{}_{\mathsf{B\,A?allow}}\Big) \times \mathsf{ef}\Big(\mathsf{fst}\ \overset{\mathsf{A\,C!money}}{\downarrow}{}_{\mathsf{A\,C?money}}\Big) = \{(\mathsf{B\,A!allow}, \mathsf{A\,C?money})\}$$

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

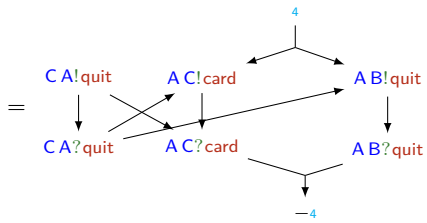
$$[\![\mathsf{G}_{\mathrm{dny}}]\!] =$$



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

$$[\![\mathsf{G}_{\mathrm{quit}}]\!] = \mathsf{seq}\Big(\overset{\mathsf{C\,A!quit}}{\downarrow}{}_{\mathsf{C\,A?quit}}\ ,\ [\![4.(\mathsf{A}{\to}\mathsf{C}\colon \mathsf{card} \mid \mathsf{A}{\to}\mathsf{B}\colon \mathsf{quit})]\!]\Big)$$

$$=$$

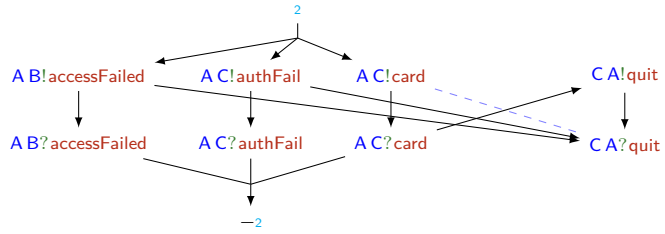


where it is easy to see that also in the case the well-sequecedness condition holds.

Provided that $ws(2.(\mathsf{A}{\to}\mathsf{B}\colon \mathsf{accessFailed} \mid \mathsf{A}{\to}\mathsf{C}\colon \mathsf{authFail} \mid \mathsf{A}{\to}\mathsf{C}\colon \mathsf{card}), \mathsf{C}{\to}\mathsf{A}\colon \mathsf{quit})$ holds, for G_{ko} we have:

$$[\![\mathsf{G}_{\mathrm{ko}}]\!] = \mathsf{seq}([\![2.(\mathsf{A}{\to}\mathsf{B}\colon \mathsf{accessFailed} \mid \mathsf{A}{\to}\mathsf{C}\colon \mathsf{authFail} \mid \mathsf{A}{\to}\mathsf{C}\colon \mathsf{card})]\!], \begin{array}{c}\mathsf{C\,A!quit}\\ \downarrow \\ \mathsf{C\,A?quit}\end{array})$$

$=$



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 $\mathsf{C\,A?quit}$); note also that the dashed arrow can be omitted because it can be obtained by transitive closure.

To compute the semantics of $\mathsf{G}_{\mathrm{wdw}}$ 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:

$$[\![\mathsf{G}_{\mathrm{wdw}}]\!] = \mathsf{seq}([\![\mathsf{C}{\to}\mathsf{A}\colon \mathsf{withdraw}; \mathsf{A}{\to}\mathsf{B}\colon \mathsf{withdraw}]\!], [\![5.(\mathsf{G}_{\mathrm{alw}} + \mathsf{G}_{\mathrm{dny}})]\!]) \quad (20)$$

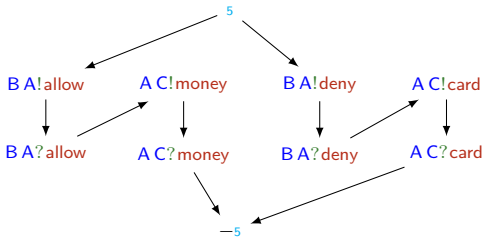
Note that the first part is very similar to $\mathsf{G}_{\mathrm{dny}}$, hence:

$$[\![\mathsf{C}{\to}\mathsf{A}\colon \mathsf{withdraw}; \mathsf{A}{\to}\mathsf{B}\colon \mathsf{withdraw}]\!] = \begin{array}{ccc}\mathsf{C\,A!withdraw} & & \mathsf{A\,B!withdraw}\\ \downarrow & \nearrow & \downarrow \\ \mathsf{C\,A?withdraw} & & \mathsf{A\,B?withdraw}\end{array} \quad (21)$$

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

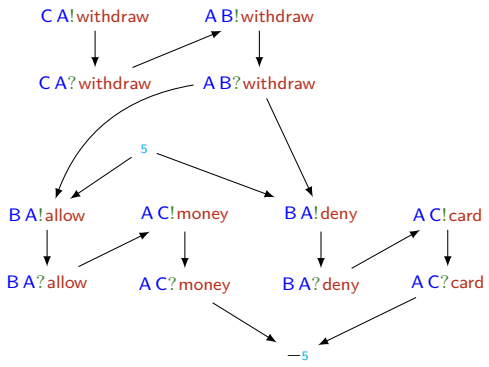
- B is active; in fact, its the first actions in both branches are different outputs
- both A and C are passive since their first actions in the two branches are different inputs

we have that the choice is well-branched, hence

$[\![5.(\mathsf{G}_{\mathrm{alw}} + \mathsf{G}_{\mathrm{dny}})]\!] =$ 

(22)

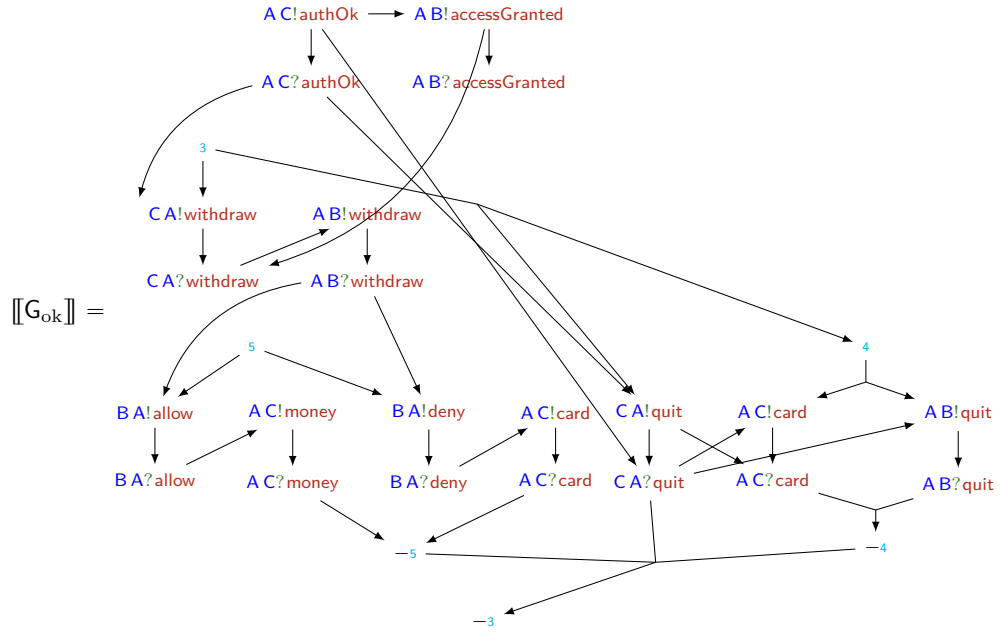
Hence, from (20), (21), and (22) we can check that the well-sequecedness condition hold and we derive:

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

where the edges obtainable by transitive closure have been omitted.

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

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

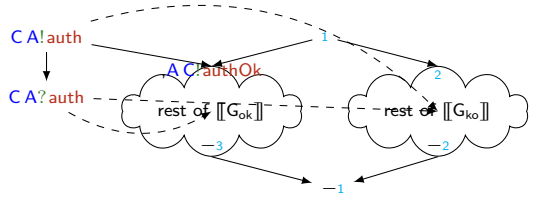


We can now reconsider

$$[\![\mathsf{G}_{\mathrm{auth}}]\!] = \mathsf{seq}([\![\mathsf{C}{\to}\mathsf{A}\colon \mathsf{auth}]\!], [\![1.(\mathsf{G}_{\mathrm{ok}} + \mathsf{G}_{\mathrm{ko}})]\!])$$

$$= \mathsf{seq}\left(\begin{array}{c}\mathsf{C\,A!auth}\\ \downarrow \\ \mathsf{C\,A?auth}\end{array}, [\![1.(\mathsf{G}_{\mathrm{ok}} + \mathsf{G}_{\mathrm{ko}})]\!]\right) \qquad \text{if } ws(\mathsf{C}{\to}\mathsf{A}\colon \mathsf{auth}, 1.(\mathsf{G}_{\mathrm{ok}} + \mathsf{G}_{\mathrm{ko}}))$$

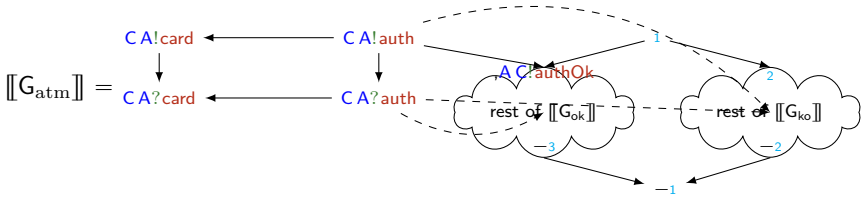
$=$



where the dashed edges represent the hyperedges connecting events with the same subject C 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 $\mathsf{G}_{\mathrm{atm}}$ is obtained by applying once more the definition of $\mathsf{seq}(\mathsf{G}, \mathsf{G}')$ and verifying that the well-sequecedness condition holds

between $\begin{matrix}\mathsf{C\,A!card}\\ \downarrow\\ \mathsf{C\,A?card}\end{matrix}$ and $[\![\mathsf{G}_{\text{auth}}]\!]$; so, we obtain



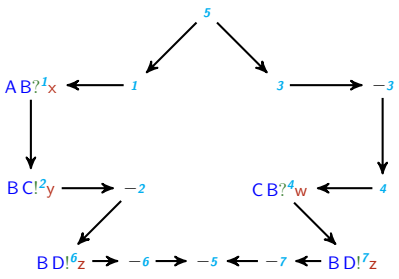
14.3 A generalisation of well-branchedness

The notion of well-branchedness given in Section 14.1 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 14.4. Consider the following variant of the g-choreography $\mathsf{G}_{Example\ 14.3}$ *in Example 14.3:*

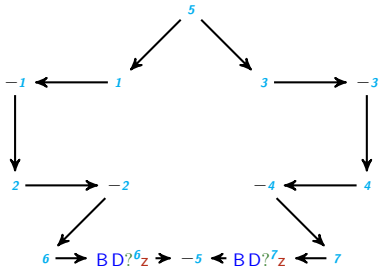
$$\mathsf{G}_{Example\ 14.4} = 5.\Big(\Big(\mathsf{A}\xrightarrow{1}\mathsf{B}\colon \mathsf{x};\mathsf{B}\xrightarrow{2}\mathsf{C}\colon \mathsf{y};\mathsf{B}\xrightarrow{6}\mathsf{D}\colon \mathsf{z}\Big) + \Big(\mathsf{A}\xrightarrow{3}\mathsf{C}\colon \mathsf{z};\mathsf{C}\xrightarrow{4}\mathsf{B}\colon \mathsf{w};\mathsf{B}\xrightarrow{7}\mathsf{D}\colon \mathsf{z}\Big)\Big)$$

We have that $[\![\mathsf{G}_{Example\ 14.4}]\!]^{@\mathsf{A}}$ *and* $[\![\mathsf{G}_{Example\ 14.4}]\!]^{@\mathsf{C}}$ *are very similar to those in Example 14.3. For* $[\![\mathsf{G}_{Example\ 14.4}]\!]^{@\mathsf{B}}$ *we have:*



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* $[\![\mathsf{G}_{Example\ 14.4}]\!]^{@\mathsf{D}}$ *we have:*



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* $\mathsf{G}_{Example\ 14.4}$ *according to Definition 3.* ◇

Intuitively, the g-choreography $\mathsf{G}_{Example\ 14.4}$ in Example 14.4 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).

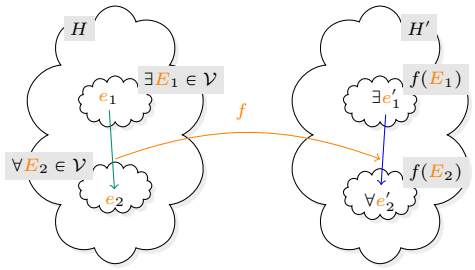
We will now introduce a more general notion which admits more well-branched g-choreography, including $\mathsf{G}_{Example\ 14.4}$ in Example 14.4. The generalised notion of well-branchedness relies on the concept of "common" part, with respect to a participant A, of the two happens-before relations 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

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



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. 12: Reflectivity

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 6. 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 $\mathsf{act}(E) = \mathsf{act}(f(E))$
- $\forall E_2 \in \mathcal{V}, e_2 \in E_2,\ \langle e_1, e_2 \rangle \in \widehat{H} : \mathsf{sbj}(e_1) = \mathsf{A} \implies (\exists E_1 \in \mathcal{V} : e_1 \in E_1 \ \wedge\ \forall e \in E_2, e' \in e_1 : \langle e, e' \rangle \notin \widehat{H} \ \wedge\ \forall e'_2 \in f(E_2) \exists e'_1 \in f(E_1) : \langle e'_1, e'_2 \rangle \in \widehat{H'})$
- $\forall E'_2 \in \mathcal{V}', e'_2 \in E'_2,\ \langle e'_1, e'_2 \rangle \in \widehat{H'} : \mathsf{sbj}(e'_1) = \mathsf{A} \implies (\exists E'_1 \in \mathcal{V}' : e'_1 \in E'_1 \ \wedge\ \forall e \in E'_2, e' \in e'_1 : \langle e, e' \rangle \notin \widehat{H'} \ \wedge\ \forall e_2 \in f^{-1}(E'_2) \exists e_1 \in f^{-1}(E'_1) : \langle e_1, e_2 \rangle \in \widehat{H})$.

An intuitive explanation of this notion is given in Fig. 12. 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

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

when E A-reflects E' with respect to $\mathcal{V}$ and $\mathcal{V}'$, we write $(E_1, E_2) = \mathsf{div}_{\mathsf{A}}^{\mathcal{V},\mathcal{V}'}(\mathsf{G}, \mathsf{G}')$ (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'.

14.3.1 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.

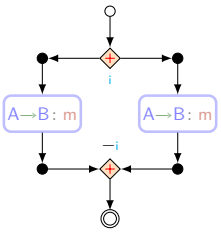
A participant $\mathsf{A} \in \mathcal{P}$ is *active* in $\mathsf{G} + \mathsf{G}'$ with respect to $\mathcal{V}$ and $\mathcal{V}'$ if, assuming $(E, E') = \mathsf{div}_{\mathsf{A}}^{\mathcal{V},\mathcal{V}'}(\mathsf{G}, \mathsf{G}')$,

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

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 3.

14.3.2 Some examples

When it exists, the active participant is the selector of the choice. Unlike the one in Definition 3, the more general notion well-branchedness does not require the selector to exist. For instance, the choreography



$$= \quad \mathsf{i}.\big(\mathsf{A}\to\mathsf{B}\colon \mathsf{m} + \mathsf{A}\to\mathsf{B}\colon \mathsf{m}\big)$$

is well-branched even if it has no active participant. The simple notion of Definition 3 does not hold also for the following example:

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

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 $(\!|\, \mathsf{A}\,\mathsf{B}!^{\mathsf{i}}\mathsf{m}, \mathsf{A}\,\mathsf{B}?^{\mathsf{i}}\mathsf{m} \,|\!)$ and $(\!|\, \mathsf{A}\,\mathsf{B}!^{\mathsf{j}}\mathsf{m}, \mathsf{A}\,\mathsf{B}?^{\mathsf{j}}\mathsf{m} \,|\!)$, our framework establishes that B is active, and both A and C are passive, making the choicewell-branched.

Part IV

...To Local Specifications

Chapter 6

An automata model of local views

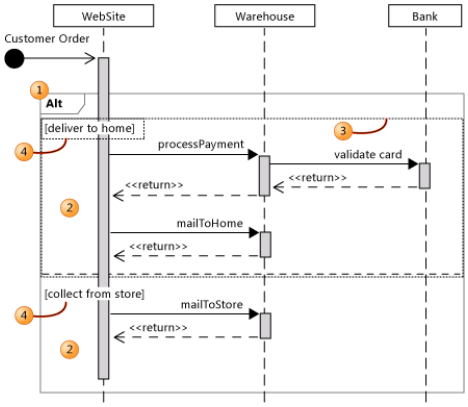
...act local!

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 III). Remarkably, our automata model is very close to modern programming languages such as Erlang and the actor model of Scala.

15 An intuition of projections

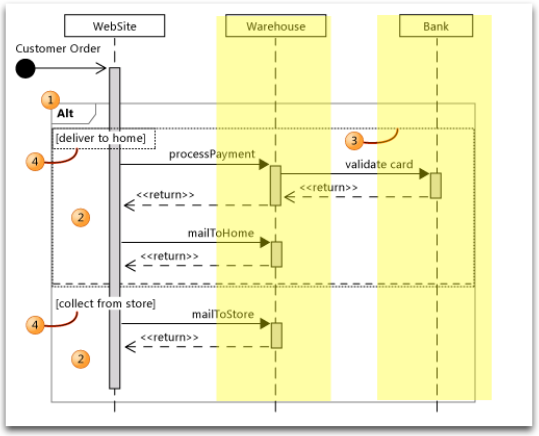
As discussed in § 1, an appealing aspect of choreography is the possibility of obtaining local viewpoints from global ones. This is often achieved by *projecting* the global viewpoint.

An intuitive description of projections can be given using sequence diagrams. For this we consider the diagram in Fig. 13 (the actual choreography described in such diagram is immaterial for our purposes). The diagram in Fig. 13 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 the diagram with respect to WebSite could be thought of as being represented by the following figure

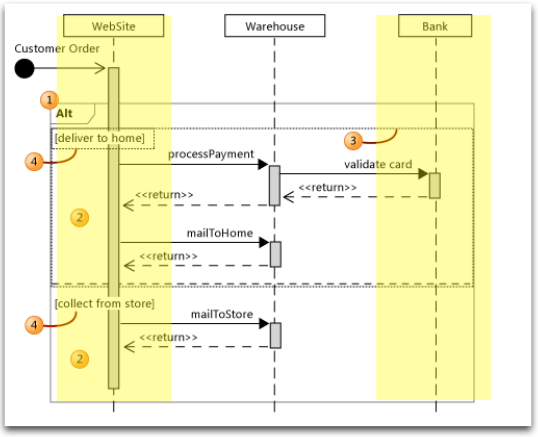


borrowed from `http://msdn.microsoft.com/en-us/library/dd465153.aspx`

Fig. 13: A sequence diagram



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.

15.1 Communicating Finite-State Machines

The model of *communicating finite-state machines* (CFSMs) was proposed in [5] 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

$$\mathcal{L} \quad := \quad \{\varepsilon\} \quad \cup \quad \mathcal{C} \times \{!\} \times \mathcal{M} \quad \cup \quad \mathcal{C} \times \{?\} \times \mathcal{M} \tag{23}$$

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*.

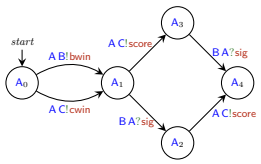
Given a set X, one can form *words on* X, namely sequences $x_1 \cdots x_n$ of elements of X; the set of words on X is customarily denoted as $X^\star$. We consider two types of words, the words on $\mathcal{M}$ and those on $\mathcal{L}$; for this, let $\mathcal{M}^*$, ranged over by w, (resp. $\mathcal{L}^*$, 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 1 (CFSM). A *communicating finite-state machine* is a finite transition system given by a 4-tuple $M = (Q, q_0, \mathcal{M}, \rightarrow)$ where

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

Example 15.1. The following machine



specifies the behaviour of the first participant of the choreography in Example 12.1 (cf. page 69). ◇

Exercise 30 *Give the machine of participant* B *of the choreography in Example 12.1.*

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 31 *List all the final, sending, receiving, and mixed stated of the machine in Example 15.1.*

Hereafter we restrict ourselves to the following class of CFSMs.

Definition 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 15.2. It is a simple observation that the CFSM in Example 15.1 is deterministic according to Definition 2. ◇

Remark 17 *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 3 (Systems). Given 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 $\mathbf{w} = (w_{\mathsf{A}\,\mathsf{B}})_{\mathsf{A}\,\mathsf{B}\in\mathcal{C}}$ with $w_{\mathsf{A}\,\mathsf{B}} \in \mathcal{M}^*$; component $\mathbf{q}$ is the *control state* and $q_{\mathsf{A}} \in Q_{\mathsf{A}}$ is the *localstate state* of machine M_{A}. The *initial configuration* of $\mathbf{S}$ is $s_0 = \langle \mathbf{q_0} \; ; \; \varepsilon \rangle$ with $\mathbf{q_0} = (q_{0\mathsf{A}})_{\mathsf{A}\in\mathcal{P}}$.

Hereafter, we fix a machine $M_{\mathsf{A}} = (Q_{\mathsf{A}}, q_{0\mathsf{A}}, \mathcal{M}, \rightarrow_{\mathsf{A}})$ for each participant $\mathsf{A} \in \mathcal{P}$ and let $\mathbf{S} = (M_{\mathsf{A}})_{\mathsf{A}\in\mathcal{P}}$ be the corresponding communicating system.

Definition 4 (Reachablestate states and configurations). A configuration $s' = \langle \mathbf{q}' \; ; \; \mathbf{w}' \rangle$ is *reachable* from another configuration $s = \langle \mathbf{q} \; ; \; \mathbf{w} \rangle$ iff either of the following conditions holds:

1. $\mathbf{q}[\mathsf{S}] \xrightarrow{\mathsf{S}\,\mathsf{R}!\mathsf{m}}_{\mathsf{S}} \mathbf{q}'[\mathsf{S}]$ and
   a. $\mathbf{q}'[\mathsf{A}] = \mathbf{q}[\mathsf{A}]$ for all $\mathsf{A} \neq \mathsf{S}$, and
   b. $\mathbf{w}'[\mathsf{S}\,\mathsf{R}] = \mathbf{w}[\mathsf{S}\,\mathsf{R}] \cdot \mathsf{m}$ and $\mathbf{w}'[\mathsf{A}\,\mathsf{B}] = \mathbf{w}[\mathsf{A}\,\mathsf{B}]$ for all $\mathsf{A}\,\mathsf{B} \neq \mathsf{S}\,\mathsf{R}$; or
2. $\mathbf{q}[r] \xrightarrow{\mathsf{S}\,\mathsf{R}?\mathsf{m}}_{\mathsf{R}} \mathbf{q}'[r]$ and
   a. $\mathbf{q}'[\mathsf{A}] = \mathbf{q}[\mathsf{A}]$ for all $\mathsf{A} \neq \mathsf{R}$, and
   b. $\mathbf{w}[\mathsf{S}\,\mathsf{R}] = \mathsf{m} \cdot \mathbf{w}'[\mathsf{S}\,\mathsf{R}]$ and $\mathbf{w}'[\mathsf{A}\,\mathsf{B}] = \mathbf{w}[\mathsf{A}\,\mathsf{B}]$ for all $\mathsf{A}\,\mathsf{B} \neq \mathsf{S}\,\mathsf{R}$.
3. $\mathbf{q}[\mathsf{B}] \xrightarrow{\varepsilon}_{\mathsf{B}} \mathbf{q}'[\mathsf{B}]$ and
   a. $\mathbf{q}'[\mathsf{A}] = \mathbf{q}[\mathsf{A}]$ for all $\mathsf{A} \neq \mathsf{B}$, and
   b. $\mathbf{w} = \mathbf{w}'$.

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

Condition (1b) in Definition 4 puts m on channel $\mathsf{S}\,\mathsf{R}$, while (2b) gets m from channel $\mathsf{S}\,\mathsf{R}$. Finally, condition (3b) establishes that internal computations do not modify any communication buffer.

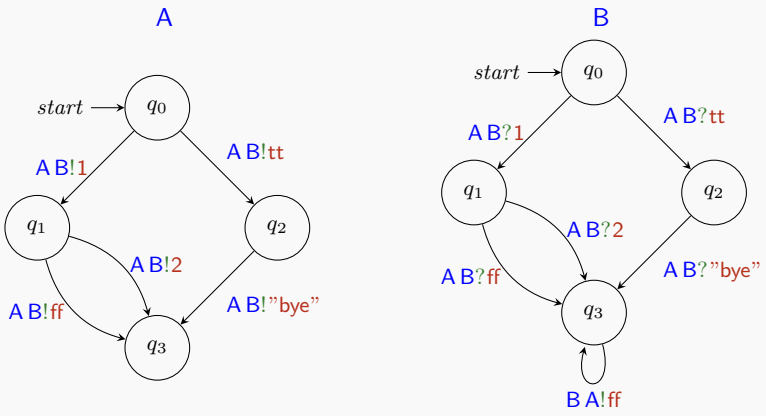
Definition 5 (Reachability). Let $s_1 \overset{\ell_1 \cdots \ell_m}{\Longrightarrow} s_{m+1}$ hold iff, for some configurations $s_2, \ldots, s_m$ we have that $s_1 \overset{\ell_1}{\Longrightarrow} s_2 \cdots s_m \overset{\ell_m}{\Longrightarrow} s_{m+1}$. The set of *reachable configurations of* $\mathbf{S}$ is

$$\mathsf{RC}(\mathbf{S}) = \left\{ s \;\middle|\; \text{there are } \ell_1, \ldots, \ell_m \text{ s.t. } s_0 \xRightarrow{\ell_1 \cdots \ell_m} s \right\}$$

Given $k > 0$, a *k-bounded execution* of a system $\mathbf{S}$ is a sequence of transitions such that the channels of the configurations of the sequence contain at most k messages. The *k-reachability set of* $\mathbf{S}$ is the largest subset $\mathsf{RC}_k(\mathbf{S})$ of $\mathsf{RC}(\mathbf{S})$ within which each configuration s can be reached by a k-bounded executions from s_0.

Note that, for every integer k, the set $\mathsf{RC}_k(\mathbf{S})$ is finite and computable.

Exercise 32 *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 33 *The communicating system of Exercise 32 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 communicating system.

Definition 6 (Well-behaved 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 $\mathbf{w}[\mathsf{A}\,\mathsf{B}] = \varepsilon$ for all $\mathsf{A}\,\mathsf{B} \in \mathcal{C}$, there is $\mathsf{R} \in \mathcal{P}$ such that $\mathbf{q}[\mathsf{R}] \xrightarrow{\mathsf{S}\,\mathsf{R}?\mathsf{m}}_{\mathsf{R}} \mathbf{q}'[\mathsf{R}]$, and for every $\mathsf{A} \in \mathcal{P}$, $\mathbf{q}[\mathsf{A}]$ is a receiving stateor final state, 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 stateor receiving state.

orphan message configuration if all localstate state in $\mathbf{q}$ are final state and there is $\mathsf{A}\,\mathsf{B} \in \mathcal{C}$ such that $\mathbf{w}[\mathsf{A}\,\mathsf{B}] \neq \varepsilon$, i.e., there is at least a non-empty buffer and each machine is in a final state.

unspecified reception configuration if there exists $\mathsf{R} \in \mathcal{P}$ such that $\mathbf{q}[\mathsf{R}]$ is a receiving state, there is $\mathsf{S}\,\mathsf{R} \in \mathcal{C}$ such that $|\mathbf{w}[\mathsf{S}\,\mathsf{R}]| > 0$, and $\mathbf{q}[\mathsf{R}] \xrightarrow{\mathsf{S}\,\mathsf{R}?\mathsf{m}}_{\mathsf{R}} \mathbf{q}'[\mathsf{R}]$ implies that $|\mathbf{w}[\mathsf{S}\,\mathsf{R}]| > 0$ and $\mathbf{w}[\mathsf{S}\,\mathsf{R}] \notin \mathsf{m}\mathcal{M}^*$, 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 [7, Def. 12].

15.2 Projecting g-choreographies to CFSM

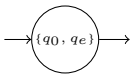
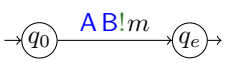
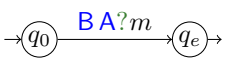
Given two CFSMs $M = (Q, q_0, \rightarrow)$ and $M' = (Q', q_0, \rightarrow')$, write $M \sqcup M'$ for the machine $(Q \cup Q', q_0, \rightarrow \cup \rightarrow')$; observe that M and M' have the same initial state. Also, $M \cap M'$ denotes $Q \cap Q'$. The product of M and M' is defined as usual 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 [18]) 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 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:

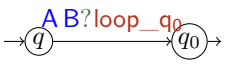
$$\mathsf{G} \downarrow_{\mathsf{A}}^{q_0,q_e} = \begin{cases} \text{  } & \text{if } \mathsf{G} = \mathsf{B}{\to}\mathsf{C}\colon \mathsf{m},\ \mathsf{A} \neq \mathsf{B}, \text{ and } \mathsf{A} \neq \mathsf{C} \\ \text{  } & \text{if } \mathsf{G} = \mathsf{A}{\to}\mathsf{B}\colon \mathsf{m} \\ \text{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

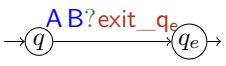
where in the last two cases we define

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

$$\mathsf{G}_{\mathtt{ls}} = \big(\mathsf{A}\to\mathsf{B}_1 : \mathsf{loop\_q_0} \mid \cdots \mid \mathsf{A}\to\mathsf{B}_\mathsf{H} : \mathsf{loop\_q_0}\big) \downarrow_{\mathsf{A}}^{q,q_0}$$

$$\mathsf{G}_{\mathtt{es}} = \big(\mathsf{A}\to\mathsf{B}_1 : \mathsf{exit\_q_e} \mid \cdots \mid \mathsf{A}\to\mathsf{B}_\mathsf{H} : \mathsf{exit\_q_e}\big) \downarrow_{\mathsf{A}}^{q,q_e}$$

$\mathsf{G}_{\mathtt{lr}} =$ 

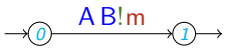
and $\mathsf{G}_{\mathtt{er}} =$ 

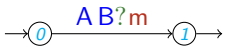
with $\{\mathsf{B}_1, \ldots, \mathsf{B}_h\}$ the set of participants of G' different from A and q being the only state shared between $\mathsf{G}_{\mathtt{ls}}$, $\mathsf{G}_{\mathtt{es}}$, and $\mathsf{G}' \downarrow_{\mathsf{A}}^{q,q_e} \sqcup \mathsf{G}_{\mathtt{es}} \sqcup \mathsf{G}_{\mathtt{es}}$ (resp. $\mathsf{G}_{\mathtt{lr}}$, $\mathsf{G}_{\mathtt{er}}$, and $\mathsf{G}' \downarrow_{\mathsf{A}}^{q,q_e} \sqcup \mathsf{G}_{\mathtt{lr}} \sqcup \mathsf{G}_{\mathtt{lr}}$). 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 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.

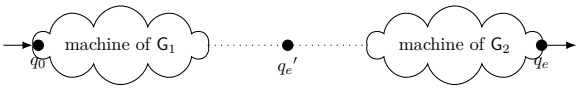
Example 15.3. The projections of $\mathsf{A}\to\mathsf{B} : \mathsf{m}$ *with respect* A *and* B *and the states* 0 *and* 1 *are*



and 

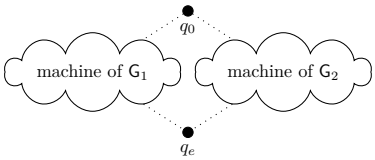
respectively. ◇

The fourth case of Definition 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 machine for G_1 so that its final state is used to as the initial state of the machine built for G_2 (represented by the dotted lines). 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:



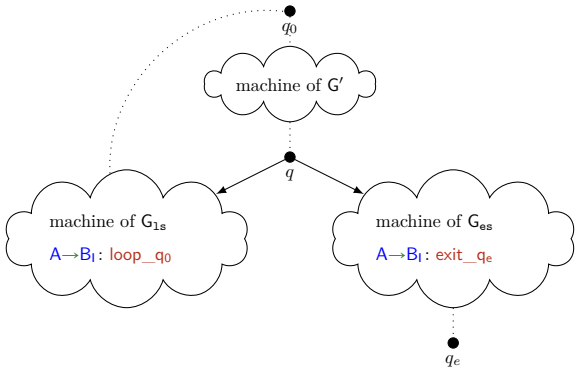
Namely, the resulting machine just follows either the executions of the machine of G_1 or the ones of the machine of G_2 just because the initial state is in common.

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 34 *Give the machines* $\mathsf{G}\downarrow_{\mathsf{A}}^{0,1}$ *and* $\mathsf{G}\downarrow_{\mathsf{B}}^{0,1}$ *for the g-choreography* $\mathsf{G} = \mathsf{A}\to\mathsf{B}\colon \mathsf{m} \mid \mathsf{A}\to\mathsf{B}\colon \mathsf{m}'$.

Exercise 35 *Let* G *be the g-choreography of Exercise 34 and* C *be a participant different from* A *and* B. *Give the CFSM* $\mathsf{G}\downarrow_{\mathsf{C}}^{(0,0),(1,1)}$.

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:



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. 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.
4. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.
5. D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, 1983.
6. 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.
7. G. Cécé and A. Finkel. Verification of programs with half-duplex communication. *I&C*, 202(2):166–190, 2005.
8. 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.
9. P. Deniélou and N. Yoshida. Multiparty Session Types Meet Communicating Automata. In *ESOP*, pages 194–213, 2012.
10. R. Dijkman and M. Dumas. Service-oriented design: A multi-viewpoint approach. *International Journal of Cooperative Information Systems*, 13(4):337–368, 2004.
11. E. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
12. R. W. Floyd. Assigning meaning to programs. In *Proc. Symp. in Applied Mathematics*, volume 19, 1967.
13. 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.

14. R. Guanciale and E. Tuosto. Semantics of global views of choreographies. *Journal of Logic and Algebraic Methods in Programming*, 2017. Revised and extended version of [13]. Accepted for publication. To appear; version with proof available at `http://www.cs.le.ac.uk/people/et52/jlamp-with-proofs.pdf`.
15. D. Harel and P. Thiagarajan. Message sequence charts. Available at `http://www.comp.nus.edu.sg/~thiagu/public_papers/surveymsc.pdf`.
16. T. Hoare. An axiomatic basis of computer programming. *CACM*, 12, 1969.
17. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. *Journal of the ACM*, 63(1):9:1–9:67, 2016. Extended version of a paper presented at POPL08.
18. J. E. Hopcroft. *Introduction to Automata Theory, Languages, and Computation*. Pearson Addison Wesley, 3rd edition, 2007.
19. 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.
20. 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.
21. J. Lange and A. Scalas. Choreography synthesis as contract agreement. In *Proceedings 6th Interaction and Concurrency Experience, ICE 2013*, pages 52–67, 2013.
22. J. Lange, E. Tuosto, and N. Yoshida. From Communicating Machines to Graphical Choreographies. In *POPL15*, pages 221–232, 2015.
23. B. Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992.
24. Z. Micskei and H. Waeselynck. Uml 2.0 sequence diagrams' semantics. `http://home.mit.bme.hu/~micskeiz/sdreport/uml-sd-semantics.pdf`.
25. 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.
26. 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.
27. Object Management Group. Business Process Model and Notation. `http://www.bpmn.org`.
28. M. Papazoglou. *Web Services and SOA: Principles and Technology*. Pearson-Prentice Hall, 2012.
29. C. Peltz. Web services orchestration and choreography. *IEEE Computer*, 36(10):46–52, 2003.

30. 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.
31. S. R. Talbot. Orchestration and choreography: Standards, tools and technologies for distributed workflows. `http://www.nettab.org/2005/docs/NETTAB2005_Ross-TalbotOral.pdf`.
32. R. Taylor, N. Medvidovic, and E. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, Feb. 2009.
33. 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 facebook 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,

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

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?

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

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:

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

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 $nameOf : Persons \to Strings$ 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 $ownerOf : carOwner \to soldCars$ 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:

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

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

Action, 52, 80
  Communication, 47, 49–55, 69–71, 77, 78, 96, 102
  Enabled, 52, 54, 55
  Input, 51, 52, 55, 108
  Output, 51, 52, 54, 82, 108
  Receiving, 53, 70, 102, 103
  Sending, 53, 70, 103
Actor, 16, 99

Branching
  Branching, 56, 57, 60, 62, 67, 83, 94, 103
  Branching Pair, 94

Channel, 47–49, 52–57, 60, 62, 70, 102, 104, 105
Choice, 58–60, 63, 80, 81, 83, 92, 93, 95, 96
  External Choice, 58–60
  Internal Choice, 58–60
Choreography, 19–21, 43–46, 48, 51, 55–63, 65, 66, 69, 78, 95, 99, 101, 103, 107
  g-choreography, 65–67, 69, 76, 78, 79, 81, 83–86, 91, 92, 94, 106–109
Communicating Finite-State Machine, 101–103
  CFSM, 101, 103–109
Communicating System, 95, 103–106
Contract, 16–20, 43, 44
  Design-By-Contract, 17
Control Point, 66, 70

Deadlock, 18, 60–63, 106
Dual, 71, 73, 74

Erlang, 15, 16, 26, 30, 32, 46, 99
Event, 70–77, 80
  Input, 70, 71, 80
  Output, 70, 71

Happens-Before Relation, 72–81, 83, 84, 87, 92

Labels, 101

Orchestration, 19–21
Orchestrator, 21

Participant, 18–21, 44–49, 51–56, 58–60, 62, 63, 65, 66, 70, 77, 79–83, 92–95, 99, 101, 103, 104, 107, 108, 110
  Active, 80–84, 92–96
  Passive, 80–84, 91–96

Reflectivity, 93, 94
Reflects, 94

Software Architecture, 3, 25–27, 29, 31–33, 37
  Architectural Style, 35
  Cohesion, 3, 34, 35
  Component, 27–29, 31, 34–36, 38–40
  Configuration, 26, 32–34, 54, 55, 61
  Connector, 27–33, 36–39
  Coupling, 3, 34, 39
State, 16, 17, 59, 62, 102, 103, 107, 108
  Final State, 103, 106, 109, 110
  Initial State, 102, 106, 108–110
  Local State, 49, 104, 106
  Mixed State, 103
  Reachable State, 104
  Receiving State, 103, 106
  Sending State, 103
Subject, 70, 71, 77, 93

Trace, 50–53
Execution
  Bounded, 105
Transition System, 102

View, 19–21, 43–46, 65, 99, 101
  global, 19, 20, 43–46, 65, 84, 85, 99, 101
  local, 19–21, 43–45, 49, 50, 57, 99, 101

Well-Branchedness, 80, 81, 83, 91–93, 95, 96
Well-Forkedness, 80
Well-Sequencedness, 79