Dottorato di Ricerca in Informatica
Università di Bologna, Padova
INF/01 INFORMATICA

Formalizing languages for Service Oriented Computing

Claudio Guidi

March 2007

Coordinatore:
Prof. Özalp Babaoğlu

Tutore:
Prof. Roberto Gorrieri

Campioni del mondo!
Campioni del mondo!
Campioni del mondo!
(Nando Martellini, 1982)

"L'ultima volta che raccolsi una sfida
mi ripromisi di non farlo mai piú.
Non sono ancora riuscito a
mantenere quella promessa."
(Anonimo)

Abstract

Service Oriented Computing is a new programming paradigm for addressing distributed system design issues. Services are autonomous computational entities which can be dynamically discovered and composed in order to form more complex systems able to achieve different kinds of task. E-government, e-business and e-science are some examples of the IT areas where Service Oriented Computing will be exploited in the next years. At present, the most credited Service Oriented Computing technology is that of Web Services, whose specifications are enriched day by day by industrial consortia without following a precise and rigorous approach. This PhD thesis aims, on the one hand, at modelling Service Oriented Computing in a formal way in order to precisely define the main concepts it is based upon and, on the other hand, at defining a new approach, called *bipolar approach*, for addressing system design issues by synergically exploiting choreography and orchestration languages related by means of a mathematical relation called *conformance*. Choreography allows us to describe systems of services from a global view point whereas orchestration supplies a means for addressing such an issue from a local perspective. In this work we present SOCK, a process algebra based language inspired by the Web Service orchestration language WS-BPEL which catches the essentials of Service Oriented Computing. From the definition of SOCK we will able to define a general model for dealing with Service Oriented Computing where services and systems of services are related to the design of finite state automata and process algebra concurrent systems, respectively. Furthermore, we introduce a formal language for dealing with choreography. Such a language is equipped with a formal semantics and it forms, together with a subset of the SOCK calculus, the bipolar framework. Finally, we present JOLIE which is a Java implentation of a subset of the SOCK calculus and it is part of the bipolar framework we intend to promote.

Acknowledgements

This Ph.D. thesis is the final result of three years of graduate studies which represent for me a beautiful life experience and a very interesting professional growing. During this period there was the difficult moments and the beautiful ones and it will take a lot of time to me for appreciating and understanding all the things I lived.

First of all, I would like to thank my friend and colleague Roberto Lucchi who shared with me difficult moments and enthusiastic scientific discussions. This work would not have been possible without him.

A special thanks goes to my supervisor Prof. Roberto Gorrieri who gave me very valuable comments and suggestions with his experience in coordination models. I also thank him for the travel opportunities and the possibility he gave me to participate within the EU integrated Project SENSORIA (contract n. 016004) which partially funded this work of thesis.

I am grateful to other people in Bologna who gaves me scientific and human support as Gianluigi Zavattaro, Nadia Busi and Mario Bravetti.

I want to thank other students and colleagues that I met during these years who shared with me moods and some appearently insignificant philosophical dicussions: Manuel Mazzara, Laura Bocchi, Stefano Cacciaguerra, Matteo Roffilli, Giulio Manzonetto, Andrea Pescetti, Cinzia Di Giusto, Fabrizio Montesi and all the others that, for the sake of brevity, I cannot mention here.

I would like to mention also the reviewers, Martin Wirsing and Kohei Honda whose observations have been very helpful during the final revision of the document.

Last but not least, I am grateful to my girlfriend. Even if we met during the last months of this Ph.D. period, it seems to me to have shared with her all the time I passed on this work.

Contents

List of Figures

Chapter 1

Introduction

In the last decades, Internet has spread its connections all over the world introducing a new means for communicating. The world economy and the social behaviours of the majority of the world population have been strongly affected by it and new technologies have been developed in order to address the issues raised by the growth of the network. On the one hand, physical and hardware aspects have been studied for incrementing communication speed and bandwidth whereas, on the other hand, computer scientists have dealt with interoperable and distributed applications complexities. Indeed, although there exist different protocol layers which allow for the abstraction from physical locations, software applications over the Internet are intrinsecally distributed because data are distributed and computational functionalities are distributed. Service Oriented Computing, SOC for short, is a new programming paradigm which addresses such an issue whose main concept it is based upon is the design and the composition of services. A service is an autonomous interoperable platform-independent computational entity which can be dynamically discovered and composed in order to obtain different systems which achieve different tasks. Services can be accessed by public interfaces which are standardized and stored within service registers that aim at being queried by other applications for retrieving, at run-time, a specific service for a specific task. Services can be reused and replaced depending on the execution context of the specific distributed application and they can be exploited by different application systems at the same time. E-government, e-business and e-science are some examples of the IT areas where Service Oriented Computing will be exploited in the next years. Nowadays, big industries and consortia like Microsoft, IBM, W3C, OASIS only to mention a few, are putting several efforts and moneys for developing tools which deal with SOC applications. Service Oriented Computing indeed, aims at being the right solution for different kind of problems both at the level of intranet company information system, where application scalability and flexibility are important issues to deal with, and at the level of business protocol designing among different companies which want to make business by interoperating their own application over the Internet [DMK$^+$, CHT, Koc]. In these years some frameworks like Corba [OMG], Java RMI [Suna] and Web Services [W3Cb] have been proposed in order to deal with such a kind of issues. Corba and Java RMI extend the object-oriented paradigm to network applications by supplying a framework where objects can be created and accessed remotely, whereas Web Services is the most credited technology which deals with Service Oriented Computing paradigm. The Web Services are a standardized XML-based technology [W3Ca] defined by means of several specification documents developed by different organizations, consortia and industries. One of the most important goal of Web Services, is the interoperability achievement. Such a kind of technology indeed, was born for addressing the necessity to supply a shared application framework layer on which different subjects, such as enteprises, universities, private citizens, etc., can built their own applications and make them easily interoperable over the Internet [Coh]. There are three specifications that are commonly considered the cornerstone of the Web Services technology: WSDL [Word], SOAP [Wora] and UDDI [Oasb]. The WSDL specification deals with a language which allows for the description of a Web Service interface, the SOAP specification defines a protocol for message exchanges among Web Services and the UDDI one deals with the dynamic discovery of a Web Service. Although Service Oriented Computing raises a lot of interests in the computer science and business communities, at the present, there not exists any kind of shared formal definition for SOC. This is probably due to the fact that the issues addressed by SOC, such as interoperability, dynamic discovery, composition, etc., were born before developing and deeply studying such a kind of approach. The rapidly growing of the Internet indeed, has forced industries to immediately tackle its dynamics and complexities. As a first response, the new developed technologies, such as Web Services, have been developed without following a formal approach so that specifications are often ambiguous and redundant. Summarizing, Service Oriented Computing was not born by following a rigorous developing method but it is the consequence of an immediate response to some real problems. This fact implies that the main concepts SOC paradigm is based upon can be extracted only from practical experiences and case studies as in [AKR$^+$05, CNM06, BCNR06], only to mention a few, technology documentations and informal documents released by industrial consortia like in [OASa, W3Cc]. Although the present technologies provide powerful means for dealing with SOC application design, the fact that SOC is not precisely defined in terms of formal definitions is becoming, day by day, a strong limit for its development. Features like dynamic discovery and composition indeed, need a common understanding on the basic mechanisms SOC is based upon in order to be achieved by different designers by exploiting different tools. Nowadays, it is possible to observe a common interest of the industrial world and the academic one to investigate formal models for describing Service Oriented Computing approach [CFNS05, WCG$^+$06, FLB06]. To this end, conferences and workshops are organized for sharing both industrial and academic investigations such as [DL06, BKZ05, DFS06] and, recently, the European Union has started an integrated project, called SENSORIA, in order to develop both theoretical foundations for SOC and tools for designing SOC applications. In this context, the present thesis must be considered. Here, the Service Oriented Computing is approached as a new programming paradigm and in the remainder of this work we will focus on the formalization of the SOC basic mechanisms which deal with service design and composition upon which, we are developing a new concrete language for dealing with SOC applications.

1.1 Can Service Oriented Computing be considered as a new programming paradigm?

Here, we consider Web Services as the most credited representative for SOC and, in the following, we will use the term service by implicitly considering that defined by Web Services technology. Namely, a service is a loosely coupled application whose interface description is standardized and public. In its interface a service exhibits all of its access points, called *operations*, on which it is possible to interact with by sending a request message and, when defined, receiving a response. Each operation defines exactly the data structure of the exchanged message and the protocol to use for communicating it.

A *service is not an object* because it is loosely coupled that is, there are not explicit links between the invoker and the service. In an object-oriented model indeed, when an object is created, a reference is relased to the object owner. The reference allows for access to all the data and methods exhibited by that object. The state of the object is strictly related to its reference and it is mantained until its destruction. Each time the object owner wants to interact with its object, it has to supply the reference in order to be enabled to access to it. In the same way, a service can have a state related to a specific invoker but the mechanism for accessing it does not exploit a reference, rather the so-called correlation data. Correlation data are a set of incoming data which allows the service for correlating a specific invocation to a specific service state. For example, let us consider the case of a travel agency which requests the name and the surname of a client in order to retrieve his booking information. Every time the client wants to access his data, he has to supply his name and his surname. The main difference between an object-oriented model and a service-oriented model can be found in the fact that, in the former case, the class designer is not aware of the reference mechanism because it is managed by the object framework, whereas, in the latter case, the designer must be aware of the correlation data and he has to decide which data must be exploited for correlation.

A *service is not a remote procedure call nor a function* because services can be composed in a so-called *callback* configuration. A callback is a service invocation which needs a reply that is not performed on the same operation where the invocation is performed but it needs that the invoker exhibits a specific operation for receiving the response. With regard to this, let us consider Fig. 1.1 where operations are represented by the black vertical lines and symbol • represents a computational step. In *i)* is shown a service invocation which resembles a remote procedure call, service A invokes service B and waits for its reply. When invoked, service B executes its code and then, by means of the same operation, sends the response to service A. On the contrary, in case *ii)* a callback composition is represented. Service A invokes service B and then continues to execute its code. When finished, service B will send the response to an operation of A by performing an explicit invocation of service A. Finally, the example *iii)* represents a more complex case where there is not a callback but there is a particular composition where three services are involved. Service A sends a requests to service B which, at the end of its execution, invokes a service C that will sends the response to the service A.

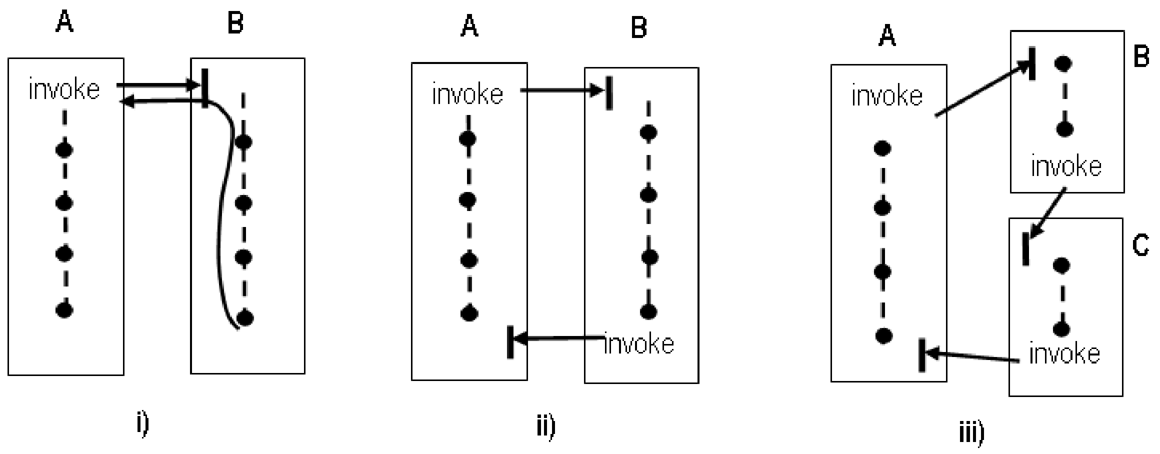

Figure 1.1: Services callback composition.

A *service is not only an application able to send and receive messages* as in the message passing paradigm. A service is not limited to the communication patterns *send* and *receive* where the former means that an access point is provided for receiving a message and the latter means that a message is sent to another application. As far as Service Oriented Computing is considered, in accordance with [BDtH, BB05], other communication patterns are provided such as the *Request-Response* and the *Solicit-Response* one where the former deals with the reception of a message and the sending of the reply whereas the latter deals with the sending of a message and the reception of the reply from the invoked application. Furthermore, public interfaces, which can be discovered at run-time by allowing for service composition within a system, and the correlation data mechanism, can be considered SOC features that are not necessary supported by the message passing paradigm.

In light of these observations, a service can be considered as the new concept of a new programming paradigm, the service-oriented one, that can be exploited for designing internet distributed applications.

1.2 Service design and service composition

Service design and service composition are two different but complementary aspects of Service Oriented Compunting. Here we exploit the terms *service design* for denoting the design of the behaviour of a service and we use the term *service composition* for representing the design process which allows for the creation of a services system by starting from simple services.

1.2.1 Service design

Service design deals with the design of the service behaviour that is the computational procedures it exploits for supplying its functionalities to the invoker. Since services have to exhibit a standardized interface (namely, a WSDL document for a Web Service), service design can in general be approached by using different programming languages. The only aspect that has to be taken into account is the interface which has to be public and standardized. The usual languages like Java, C++, C, etc. does not supply specific primitives for service oriented communication that explicitly models operations, and, at the same time, they do not support specific constructs for data correlation. This fact raises difficulties when the design of the so-called orchestrators is considered. An orchestrator is a particular kind of service which is able to coordinate other services by performing different invocations. In Fig. 1.2 service B represents an orchestrator service which receives a request from service A and, before sending the response, it invokes in sequence service C and service D. As far as Web Services are concerned, specific languages for designing

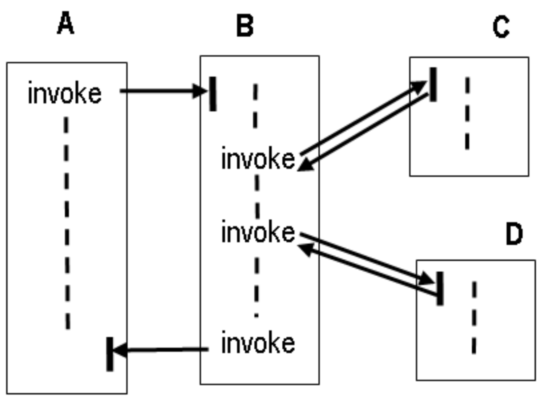

Figure 1.2: An orchestrator service

orchestrators have been developed by international consortia: WSCI[1] [Con] developed by the W3C, and WS-BPEL [OASc] developed by OASIS. Such a kind of languages have specific constructs for performing invocations on Web Services operations and they are equipped of workflow constructs like sequence, parallel and choice; furthermore they support data correlation.

1.2.2 Service composition

Service composition deals with the fact that more than one service can be exploited together in order to achieve a specific task. Such a kind of issue raises some difficulties related to the fact that, usually, services can be retrieved at run-time and they can be replaced during the execution of a system and, moreover, their behaviours must be coherent with the message protocol defined for the overall system. At the state of the art, as far as service composition is concerned, the approach is twofold. On the one hand, the languages for designing orchestrators, *orchestration languages* for short, are exploited for composing services. By means of orchestrators indeed, it is possible to invoke and coordinate the services involved into a system in order to achieve a specific task. On the other hand, the so-called *choreography languages* allows for the description of a ser-

[1]WSCI does not allow for the design of an executable version of an orchestrator but it supplies a means for giving an abstract definition of it.

vices system in a top view manner where it is possible to define the rules which govern a system by designing the interactions among the different involved participants. As far as Web Services are considered, the most credited choreography language is WS-CDL [Worc] of the W3C. Considering the terminology introduced by Honda et al. in [MCY], we say that orchestration languages allows for the composition of services by means of a *local view point* which is that of the orchestrators involved into a system, whereas the choreography ones supply a means for describing a system from a *global view point*. In or-

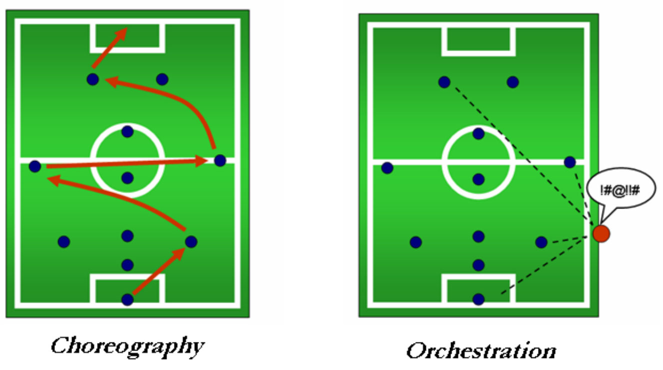

Figure 1.3: Soccer example

der to explain the different approaches of orchestration and choreography let us consider the case of a soccer team where players can be intended as services. In this context, the game strategies are the choregraphy whereas the coach well represents an orchestrator. Complex systems could have more than one orchestrators, for instance the goalkeeper could be the orchestrator of the defense line whereas the center field player could be the orchestrator of the forward line. It is worth noting that the orchestrators must follow the strategies represented within the choreography and the other players will follow the orchestrator signals in order to score the goal.

1.3 Aims of the thesis

The aims of this thesis are twofold. On the one hand, it aims at supplying a reasoning about the main concepts the Service Oriented Computing paradigm is based upon by

following a formal approach, that we believe it is the only way which allow for a precise definition of SOC mechanisms. By means of formal models indeed, it is possible to define the peculiarities of services and services systems without ambiguity that allow for an easy reasoning about the characteristics of SOC paradigm. On the other hand, the thesis aims at supplying a new set of concrete languages for dealing with service design and composition that are based upon a formal model. These languages allow a human designer to deal with services system design by following a new approach, called *bipolar approach*, where two languages are exploited together for achieving the system design. In light of these purposes, the present thesis could be considered as a sort of bridge between a pure theoretical approach and a software engineering one. In this context, the main contribution of this work must be intended. In particular, in this thesis, a lot of work has been done in order to pave the way for a complete formal framework which deals with all the aspects of the Service Oriented Computing paradigm, and some significant steps have been done toward such a visionary result. Furthermore, an open source project has been started in order to develop a new orchestration language for SOC [Ope].
The starting point from which Service Oriented Computing paradigm has been investigated in this thesis is the Web Services technology that we have also exploited for continously tracing a comparison between our work and the industry trend. Web Services specifications are enriched, day by day, by industrial consortia which are investing a lot of money on their development. They are generally defined by exploiting XML language without following a formal approach and they are often verbose and ambiguous. In particular, we have focused on composition languages, such as WS-BPEL and WS-CDL, that are not equipped with a formal semantics, in order to understand the informal mechanisms they are based upon. Web Services composition languages indeed, directly deals with service design and composition and they provide an exhaustive background on which it is possible to extract a formal representation of SOC basic concepts such as communication patterns, mobility mechanisms and composition issues.
The roadmap of our work can be divided into two main steps that correspond to the two parts of this thesis:

- From the informal specifications to a general model for Service Oriented Computing

- Toward a new set of concrete languages: the bipolar approach

Within the former step, three main contributions can be distinguished: *i)* development of SOCK which is a calculus for Service Oriented Computing, *ii)* analysis of the mobility mechanisms in Service Oriented Computing and *iii)* a general model for Service Oriented Computing.

i) As far as SOCK is concerned, we have developed it by analyzing the most important Web Services specifications which deal with the service design and composition such as WSDL, WS-BPEL and WS-CDL. SOCK deals with the basic mechanisms which characterize the SOC paradigm as communication mechanisms, data correlation, workflow operators and services system composition. Communication mechanisms and data correlation are at the basis of data exchanging in SOC, whereas workflow operators and services system composition represent the main concepts service design and composition are based upon. The main contribution of SOCK is twofold. On the one hand it allows us to distinguish and formalize some fundamental concepts of Service Oriented Computing such as the design of a service behaviour, its deployment in an executing enviroment and the composition of services within a system. For each of these topics, by means of SOCK, we are able to precisely define the basic mechanisims they are characterized by and it will be possible to reason about them in a formal way. On the other hand, we promote SOCK as a full concrete and formalized SOC language for dealing with all the aspects of service design and composition. SOCK is three-layered structured and it is composed of the *service behaviour*, the *service engine* and the *services system*. The service behaviour deals with the design of the behaviour of the service, the service engine deals with the actual deployment of a service behaviour within a machinery ables to execute it and the services system deals with the composition of service engines within a system.

ii) Starting from SOCK we analyze the different kinds of mobility within Service Oriented Computing. We distinguish among the *internal state mobility*, the *location* *mobility*, the *interface mobility* and the *behaviour mobility*. The internal state mobility models the data exchange between the states of two services by means of a communication message. The location mobility models the possibility to pass, by means of a message exchange, service locations. The interface mobility allows for the representation of the communication of information related to the interface of the service. Finally, the behaviour mobility represents a sort of code mobility where the exchange information represent a piece of service behaviour to execute by the receiver. This investigation is useful to the end of the system design issues because different kind of mobility mechanisms need different language primitives which straightforwardly affect the composition of a system.

iii) The general model we propose for Service Oriented Computing follows the three-layered structure proposed for SOCK by modelling the three concepts of *service behaviour*, *service engine* and *services system* by means of formal machineries which abstract away, as much as possible, from the language details. The service behaviour deals with the representation of the behaviour of a service by means of a finite state automaton where both computational and communication capabilities are exploited, the service engine deals with the formalization of a machinery which is able to execute a service behaviour infinitely often and, finally, the services system deals with the representation of a system composed by more than one service engine by means of a language inspired by a process algebra such as CCS [Mil89] and CSP [Hoa85].

Within the latter step we introduce the *bipolar approach* by presenting the formal framework it is based upon that is composed of two calculi, the choreography and the orchestration one, and a conformance notion between them. Finally, we present an implementation of the orchestration language called JOLIE. The choreography language which is called C_L, has been developed by analyzing the WS-CDL specifications and it allows for the management of a system from a global view point. The main concepts the choreography language is based upon, are those of *role* and *interaction*. The former represents a system participant in an abstract way without focusing on its implementation details but focusing only on the operations it provides, whereas the latter expresses a message exchange between two roles and it can be composed by exploiting workflow constructs as sequence, parallel and choice. On the contrary, the orchestration language, which is a subpart of SOCK, it is inspired by WS-BPEL and it allows for the representatiion of a services system from a local view point where it is possible to design the behaviour of the services by composing communication and computational activities. Both languages are equipped with a formal semantics which are related by a notion, called *conformance*, that is a sort of bisimulation between their labelled transition systems. The conformance notion exploits the so-called *joining function* in order to map the services on the orchestration side with the roles on the choreography one and it tests if all the interactions described within the choreography are performed by the services within the orchestration system. The choreography language, the orchestration language and the conformance notion are the elements of the *bipolar framework* which we exploit for addressing the system design issue by considering the so-called *bipolar approach*. The idea the bipolar approach is based upon is that *a difficult thing in orchestration is an easy thing in choreography and vice versa*. In order to give the intuition we can trace a comparison with signal analysis. A signal can be processed in the time domain or in the frequency one and the Fourier transform allows for the change from one domain to the other one. It is well known that some things are easy in the frequency domain (e.g. filter design) and other things are easy in the time domain (e.g. signal sampling). In the same way, orchestration and choreography languages supply different domains for representing composed systems whereas the conformance relation plays the role of the Fourier transform. Such a kind of framework could be exploited as it follows: a first coarse system can be designed as a choreography from which it is possible to extract a conformant orchestrated system skeleton that, subsequently, can be enhanced by adding other services or by enriching the behaviour of the existing ones. Afterward, it is possible to rebuild a conformant choreography from the previous system and then adjust or enhance it for introducing more details; then, from the new choreography it will be possible to come back to the orchestrated system and so on. We can describe the bipolar framework by means of Fig 1.4 where the relation between the orchestration domain and the choreography one is given by the conformance notion and two different algorithms, the Extracting Choreography and the Extracting an Orchestration, allow for the generation of a choreography starting from an orchestrated system and the generation of an orchestrated system starting from a choreography respectively. At the present, we are focusing on the orchestration and choreography domains devel-

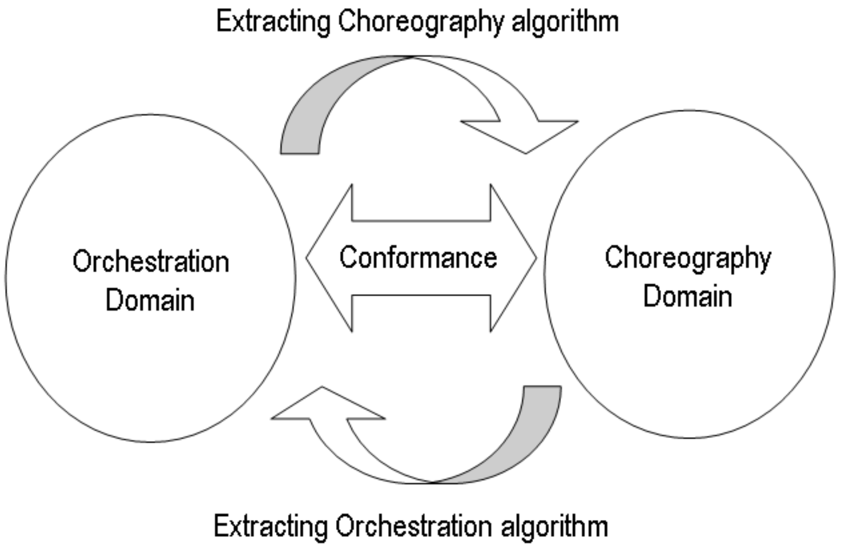
Extracting Choreography algorithm
Orchestration Domain
Conformance
Choreography Domain
Extracting Orchestration algorithm

Figure 1.4: The bipolar framework

opment and on the conformance relation between them. As far as the two algorithms are concerned, we have started to analyze them even if, so far, we cannot present results related to them.

Finally, at the end of the second step, we have defined and developed a new orchestration language based on SOCK which aims at supplying an easy tool for designing orchestrator services. The language is called JOLIE (Java Orchestration Language Interpreter Engine) and its syntax resembles that of C which is a more intuitive language w.r.t. the XML based ones. JOLIE is an open source project and its code is published in [Ope]. Here, we intend to promote JOLIE as a good candidate for becoming a powerful orchestration language for Service Oriented Computing. Joint to the fact that JOLIE provides a very easy and intuitive syntax, there is the most important fact that it is fully based upon SOCK thus it is equipped of a formal semantics which allows us to reason about it in a formal way. Recalling the general aims of this thesis, JOLIE could be considered as the bridge between the theoretical approach and the software engineering one, on the one hand indeed, it is precisely defined in a formal way and, on the other hand, it is a concrete language for designing orchestrators.

1.4 Outline

The outline of the thesis follows:

- In Chapter 2 we recall the main features of Web Service technology specifications with a particular regard to the specification which deal with service design and composition: WSDL, WS-BPEL and WS-CDL. Furthermore, we provide a brief introduction to process algebra and we discuss some general assumptions adopted within the thesis.
- In Chapter 3 we present the SOCK calculus.
- In Chapter 4 we discuss the mobility mechanism in SOC.
- In Chapter 5 we present a general model for Service Oriented Computing based upon the definitions of service behaviour, service engine, services system.
- In Chapter 6 we present the choreography language.
- In Chapter 7 we present the orchestration language.
- In Chapter 8 we present the conformance notion.
- In Chapter 9 we discuss two examples in order to show how the bipolar approach can be exploited for addressing system design issues.
- In Chapter 10 we present JOLIE.
- In Chapter 11 conclusions and future works are reported.

1.5 Related Publications

The main contribution on which some of the chapters of this thesis are based, have already been published during the Ph.D. studies.

- As far as chapter 3 is concerned, the paper *SOCK: a calculus for service oriented computing* [GLG+06] presents the syntax and the semantics of the SOCK calculus and it has been accepted at the fourth international conference on Service Oriented Computing of 2006 (ICSOC '06)
- The paper *Mobility mechanisms in Service Oriented Computing* [GL06] deals with the discussion about the mobility mechanisms in Service Oriented Computing and it has been published in the proceedings of the eighth IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'06).
- As far as chapter 6 is concerned, the choreography language has been firstly introduced within the paper [BGG+05a].
- The bipolar framework has been introduced in the papers *Choreography and Orchestration: a synergic approach for system design* [BGG+05b] and *Choreography and Orchestration conformance for system design* [BGG+06] published in the proceedings of the third international conference on Service Oriented Computing of 2005 (ICSOC'05) and in the proceedings of the eighth International Conference on Coordination Models and Languages (COORDINATION'06), respectively.
- The language JOLIE has been presented in the paper [MGLZ].

Chapter 2

Background

In this chapter we provide the background which is necessary to understand the remainder of the thesis. We introduce Web Service technology with a particular regard to those specifications which are related to service design and composition. Furthermore, we recall process algebra theory and we comment some necessary assumptions we made in order to develop such a work of thesis.

2.1 Web Services

Web Services are characterized by a pletora of XML [W3Ca] based specifications which deal with different aspects of such a kind of technology. Some distinctions can be done in order to categorized the specifications into different layers. Here, we distinguish three main specification layers:

- Base-level specifications
- Quality of Service specifications
- Composition specification

It is worth noting that other specifications there exist but, for the sake of brevity, here we discuss only the most important ones which allows us to give a comprehensive overview of the technology.

2.1.1 Overview

2.1.1.1 Base-Level specifications

In this category there are three main specifications which characterize Web Services: WSDL, SOAP, UDDI.

- *WSDL: Web Service Description Language* [Word]. This specification deals with the description language which allows for the standard definition of a Web Service interface. It is a fundamental specification which fixes the basic communication primitives, the *operations*, exploited by a Web Services for exchanging messages. There are four kind of operations: *One-Way*, *Request-Response*, *Notification* and *Solicit-Response*. In a One-Way operation a message is received, in a Request-Response operation a message is received and a response is sent to the invoker, in a Notification a message is sent and, finally, in a Solicit-response a message is sent to another Web Service and a response is received. Since WSDL specification is strictly related to service design, we will deeply discuss it in the following.

- *SOAP: Simple Object Access Protocol* [Wora]. This specification describes the basic format of an exchanged message between two Web Services. A message is an XML document composed by two different parts: the *header* and the *body*. The former part is an optional part which can contain special control information which characterize the communication such as, e.g., security references for retrieving cryptographic keys or reliability tags for guaranteeng message delivery, whereas the latter part contains the information to be communicated. Since the SOAP specification does not directly deal with service design and composition, for the sake of this thesis, we do not present in a more detailed way such a kind of specification. The reader who is interested in this topic may consult [Wora].

- *UDDI: Universal Description Discovery and Integration* [Oasb]. This specification deals with the programming interface exhibited by a discovery registry which is a particular kind of service that allows for the retrieving of Web Service depending on their functionalities. The UDDI specification introduces the concept of dynamic discovery of a Web Service. At run-time a Web Service can be potentially discovered by performig a query on the discovery registry. This thesis does not explicitly deal with dynamic discovery even if the topic is indirectly treated when mobility mechanisms in SOC will be discussed. Moreover, from a service design perspective, in SOCK service location communication is permitted and there are primitives which model operation invocation that are a sufficient means for roughly modelling dynamic discovery. For this reason, the UDDI specifications will not be commented in detail.

2.1.1.2 QoS specifications

This group of specifications are based upon the basic ones and they deal with other characteristics which can be implemented in order to allow the service to supply extra functionalities such as, e.g., security or transactional aspects. It is out of the scope of this thesis to supply an exhaustive list of such a kind of specifications and, in the following, we list only the most important ones.

- *Specifications which deal with security aspects*: security is achieved by extending the SOAP specifications. The header part of a message indeed, can be enriched by introducing nonce and references to cryptographic keys. WS-Security [OASe] deals with the security aspects of a message exchange between two dialoguers. WS-Trust [IBMb] and WS-SecureConversation [IBMa] deal with the security aspects in a domain where security credentials must be distributed and trusted by participants.

- *Specifications which deal with transactional aspects*: transactions are fundamental for designing e-business applications. WS-Coordination [Mica] defines a Web Services framework where different participants can interact by exploiting specific transaction protocols which are defined within WS-Transactions [Micb] specifications.

- *Specifications which deal with message reliability*: In general, the HTTP protocol, which usually underlies the SOAP one, is not sufficient for guaranteeing the message reliability. The WS-Reliability specification [OASd] deals with such a kind of issues.

2.1.1.3 Composition specification

This group of specifications deal with the composition issue which allows for definition of systems where several services can interact each other in order to fulfill a specific task.

- *WS-BPEL* [OASc]. It is an Oasis specification where an XML-based language for dealing with Web Services orchestration is defined. WS-BPEL is the evolution of BPEL4WS [CGK$^+$] which was a first attempt, developed by Microsoft, IBM and other software industries, to define an orchestration language which merges together some features of WSFL (Web Service Flow Language) by IBM [Ley] and XLANG by Microsoft [Tha]. Such a kind of specification will be discussed deeply in the following.

- *WS-CDL* [Worc]. It is a W3C specification which defines an XML-based language for dealing with Web Services choreography. The development of such a language is followed by both industry and academic experts. WS-CDL will be discussed in detail in the following.

- *WS-Addressing* [W3Cd]. This specification deals with the definition of all the necessary information which represent a conversation end-point and it indirectly allows for the end-point reference exchange. Such a feature, from a system design point of view, allows for the design of Web Services which receives at run-time the address of a dialoguer by allowing a dynamically composition of services during their execution. The end-point mobility topic will be discussed in Section 4 where we analyze mobility mechanisms in service oriented computing. In such a section the end-point mobility will be related to the location one. Here, we do not comment deeply WS-Addressing specification because we are not interested in modelling all the communication details of Web Services message exchange.

2.1.2 WSDL

The WSDL specification deals with the definition of a Web Service interface by introducing an XML-based language which allows for the description of the access points of the service, called *operations*. Each Web Service must public its own WSDL document in order to allow other invokers to access its operations. In the following we present the structure of a WSDL document:

```
<wsdl:definitions name="nmtoken"? targetNamespace="uri"?>
   <import namespace="uri" location="uri"/>*
   <wsdl:documentation .... /> ?
   <wsdl:types> ?
       <wsdl:documentation .... />?
       <xsd:schema .... />*
       <-- extensibility element --> *
   </wsdl:types>

   <wsdl:message name="nmtoken"> *
       <wsdl:documentation .... />?
       <part name="nmtoken" element="qname"? type="qname"?/> *
   </wsdl:message>

   <wsdl:portType name="nmtoken">*
       <wsdl:documentation .... />?
       <wsdl:operation name="nmtoken">*
          <wsdl:documentation .... /> ?
          <wsdl:input name="nmtoken"? message="qname">?
              <wsdl:documentation .... /> ?
          </wsdl:input>
          <wsdl:output name="nmtoken"? message="qname">?
              <wsdl:documentation .... /> ?
          </wsdl:output>
          <wsdl:fault name="nmtoken" message="qname"> *
              <wsdl:documentation .... /> ?
          </wsdl:fault>
       </wsdl:operation>
   </wsdl:portType>

   <wsdl:binding name="nmtoken" type="qname">*
       <wsdl:documentation .... />?
```

```
        <-- extensibility element --> *
        <wsdl:operation name="nmtoken">*
           <wsdl:documentation .... /> ?
           <-- extensibility element --> *
           <wsdl:input> ?
               <wsdl:documentation .... /> ?
               <-- extensibility element -->
           </wsdl:input>
           <wsdl:output> ?
               <wsdl:documentation .... /> ?
               <-- extensibility element --> *
           </wsdl:output>
           <wsdl:fault name="nmtoken"> *
               <wsdl:documentation .... /> ?
               <-- extensibility element --> *
           </wsdl:fault>
        </wsdl:operation>
    </wsdl:binding>

    <wsdl:service name="nmtoken"> *
        <wsdl:documentation .... />?
        <wsdl:port name="nmtoken" binding="qname"> *
           <wsdl:documentation .... /> ?
           <-- extensibility element -->
        </wsdl:port>
        <-- extensibility element -->
    </wsdl:service>
    <-- extensibility element --> *
</wsdl:definitions>
```

- Tag <*definitions*> is the main tag of a WSDL document. It includes all the other tags of the document.

- Tag <*import*> allows for the inclusion of other WSDL documents. WSDL documents indeed, can be obtained by composing other documents.

- Tag <*types*> allows for the definition of data types by means of the definition of XML-Schemas. Such a kind of types can be exploited for defining the content of the exchanged messages.

- Tag <*message*> allows for the definition of the exchanged messages. Each message exchanged by the Web Service indeed, must be declared by means of this tag and it is formed by the so-called *part*. Each part has a specific content whose type can be defined by exploiting the XML-Schema declared in the previous tag.

- Tag <*portType*> describes an abstract collection of operations which will be deployed at the same location under the same protocol. The operation describes the access points exhibited by the Web Service. There are four kinds of operations:

  – *One-Way*: the Web Service receives a message

  – *Request-Response*: the Web Service receives a message and sends a response to the invoker

  – *Notification*: the Web Service sends a message to another service.

  – *Solicit-Response*. the Web Service sends a requets message to another service and waits for its response.

  Usually, the Notification and the Solicit-Response are not declared within a WSDL document. For each operation the input and the output messages are defined by referring to the ones declared within the tags <*message*>. Moreover, it is possible to define a fault message in the case an application error occurs.

- Tag <*binding*> allows for the binding of a specific protocol to each portType. For each message of each portType indeed, it is possible to define the message exchange protocol to follow. Usually, SOAP over HTTP is exploited but there are no restrictions about the protocol to use and, moreover, WSDL can be extended in order to introduce new protocols.

- Tag <*service*> allows for the declaration of the Web Services *ports* that are the real access points of the Web Service. Each port has its own portType, its binding and it is deployed to a specific address.

By specification, WSDL can be extended depending on the application context, some other tags, indeed, can be defined and added in order to enrich the interface description of a Web Service. This is the case, for example, of the WS-BPEL specification that introduces the <*partnerLinkType*> tag which will be discussed in detail in Section 2.1.3.1.

2.1.3 WS-BPEL

It is an Oasis specification where an XML-based language for dealing with Web Services orchestration is defined. WS-BPEL is the evolution of BPEL4WS [CGK$^+$] which was a first attempt, developed by Microsoft, IBM and other software industries, to define an orchestration language which merges together some features of WSFL (Web Service Flow Language) by IBM [Ley] and XLANG by Microsoft [Tha]. A WS-BPEL process is called *business process* an it describes the behaviour of an orchestrator by means of workflow constructs and communication primitives. A business process can actually be executed by the so-called *orchestrator engines*. An orchestrator engine is an execution environment which takes in input a WS-BPEL specification and then animates it. In the following, we present the main characteristics of the WS-BPEL language, the reader who is interested in WS-BPEL details may consult ([OASc]). Furthermore, we present two business process examples extracted from the specifications that we exploit in the following of this thesis for tracing some comparisons with the developed calculi, and we briefly introduce some WS-BPEL engines provided by different producers.

2.1.3.1 The language

A WS-BPEL business process is formed by two main tags: <*definitions*> and <*process*>. The former represents the WSDL document of the business process which contains all the portTypes exhibited by it and all the portTypes exhibited by the other services it will interact with. The latter contains all the activities that have to be executed by the business process. In the following we discuss the two main tags by highlighting the most important features.

<**definitions**> Tag <*definitions*> contains the WSDL document related to the WS-BPEL business process and its document structure follows that defined within the WSDL specification. The WSDL of a WS-BPEL process is extended with somke other tags in order to deal with specific aspects of the business process. These tags are: <*partnerLinkType*>, <*property*> and <*propertyAlias*>.

The tag <*partnerLinkType*> is exploited for describing all the peer-to-peer relationships where the business process is involved. In general, we call a *partner* a service which interacts with the business process. A business process can have several partners and, for each of them, can have several interactions. In particular, the tag <*partnerLinkType*> allows for the description of a two dialoguers relationship where each partnerLinkType is characterized by two roles that participate within the relationship and, for each of them, a portType where the message exchanges are performed is declared. In the following, we present a partnerLinkType example extracted from the specifications:

```
<plnk:partnerLinkType name="invoicingLT">
    <plnk:role name="invoiceService" portType="pos:computePricePT"/>
    <plnk:role name="invoiceRequester" portType="pos:invoiceCallbackPT"/>
</plnk:partnerLinkType>
```

Here, a partnerLinkType, named *invoicingLT*, is defined with two roles: *invoiceService* and *invoiceRequester*. The former role is joined with the portType *computerPricePT* whereas the latter is joined with the portType *invoiceCallbackPT*[1]. In Fig. 2.1 we abstractly represent the partnerLinkType of the example where the dotted double arrow represents the relationship between the two roles. Such a kind of relationship is defined on the two portTypes represented as a black rectangle. Intuitively, the partnerLinkType defines an abstract relationship between two ports of two different dialoguers. Since the partnerLinkType definition does not allow for the joining of specific services to the roles, each business process must declare the role it plays within each partnerLinkType it is involved in. As we will see in the next section, such a declaration is performed within the tag <*partnerLink*> contained within the tag <*process*>. It is worth noting that, in a case where one of the two dialoguers is *a priori* unknown, a partnerLinkType may con-

[1]For the sake of brevity we do not report the portType definitions.

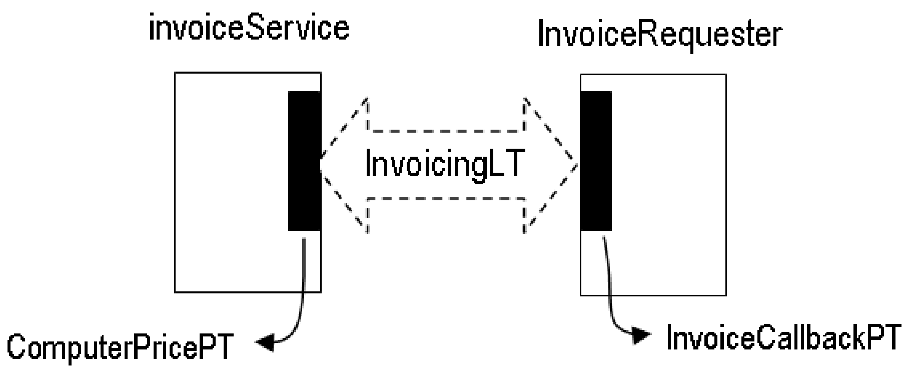

Figure 2.1: WS-BPEL partnerLinkType construct

tain only one role. This is the case of a business process, for example, which supplies a service by means of a single Request-Response operation. Such a kind of service indeed, does not need to know the portType of the invoker because its Request-Response operation is sufficient to successfully complete the interaction with it. On the contrary, it is fundamental to declare the portType of both the participants when different messages are exchanged during the execution of the service (e.g. a callback configuration between two services).

The tags <*property*> and <*propertyAlias*> allow for the management of subparts of message data structures which will be used for correlating the incoming messages[2]. The former tag allows for the definition of a unique name, a *property*, for an XML Schema type whereas the latter tag allows for the association of a subpart of a message with a property. When a propertyAlias is defined on a subpart of a message, within the business process it is possible to refer to that subpart by using the corresponding property. Let us consider the following example where the part *identification* of the message *taxpayerInfoMsg* is joined to the property *taxpayerNumber* by means of the propertyAlias tag.

```
<wsdl:definitions  ...>
 ...
 <wsdl:message name="taxpayerInfoMsg">
   <wsdl:part name="identification" element="txtyp:taxPayerInfoElem" />
 </wsdl:message>
```

[2]The correlation mechanism will be presented in the following.

```
 <vprop:property name="taxpayerNumber" type="txtyp:SSN" />
  ...
  <vprop:propertyAlias propertyName="tns:taxpayerNumber"
  messageType="txmsg:taxpayerInfoMsg"
  part="identification">
   ...
 </vprop:propertyAlias>

</wsdl:definitions>
```

It is worth noting, that the type of the message part must be coherent with the type defined within the property.

<**process**> The tag <*process*> allows for the definition of the activities executed by the business process. The structure of a process tag is represented in the following where, for the sake of brevity, some details are omitted:

```
<process name="NCName" ...>
 ...
<partnerLinks>?
        <partnerLink name="NCName"
        partnerLinkType="QName"
        myRole="NCName"?
        partnerRole="NCName"? ...>+
        </partnerLink>
</partnerLinks>

 ...

<variables>
 <variable name="BPELVariableName" ...>
 </variable>
</variables>
```

```
<correlationSets>?
 <correlationSet name="NCName" properties="QName-list" />+
</correlationSets>

<faultHandlers>?
  ...
</faultHandlers>

<eventHandlers>?
   ...
</eventHandlers>

 activity
```

In the following we comment each tag contained within the process one.

<**partnerLinks**> For each partnerLinkType where the business process is involved, it has to be declared which role is enroled by the business process and which by its partner. The *<partnerLinks>* tag allows for such a kind of declaration by means of the attributes *myRole* and *partnerRole* where the former specifies the role of the business process and the latter that of the partner. If we consider the partnerLinkType example presented in the previous section, the related partnerLink for an orchestrator which enroles the *invoiceRequester* role can be defined as follows:

```
<partnerLinks>
  <partnerLink name="invoicing"
    partnerLinkType="lns:invoicingLT"
    myRole="invoiceRequester"
    partnerRole="invoiceService" />
</partnerLinks>
```

Each partner within a partnerLink is joined with an endpoint reference which is the actual means for identifying an orchestrator and interacting with it. The endpoint reference association is differently managed depending on the engine which animates the specification.

<**variables**> The tag <*variables*> allows for the definition of the variables exploited within the business process. Variables will be exploited for storing the received messages and for manipulating data. In the following we present an example of variable declaration where the attribute *messageType* refers to a message defined within the WSDL and specifies that the type of the variable must be the same of that of the declared message:

```
<variables>
 <variable name="PO" messageType="lns:POMessage"/>
 <variable name="Invoice" messageType="lns:InvMessage"/>
 <variable name="shippingRequest" messageType="lns:shippingRequestMessage"/>
 <variable name="shippingInfo" messageType="lns:shippingInfoMessage"/>
 <variable name="shippingSchedule" messageType="lns:scheduleMessage"/>
</variables>
```

<**correlationSets**> Different intances of a business process can be executed concurrently on the same engine. Each instance executes the same business process but with a different set of data. Usually, a business process instance is initiated by a message reception and it is identified by a particular set of the received data. The data which allows for the identification of each instance are defined within the so-called *correlation set*. Let us consider, for example, the case of a business process which starts its execution by receiving a message that contains the nickname of a user. Moreover, let us assume that the nickname is declared within the correlation set of the business process and that the nicknames univocally identify a user. If such a kind of business process is concurrently invoked by two users which have different nicknames as, for example, Micky Mouse and Homer Simpson, two instances will be created and each instance will be identify by the corresponding nickname. The tag <*correlationSets*> allows for the definition of all the data which are used for identifying a specific instance of the business process. In the following we present an example of correlation sets declaration:

```
<correlationSets xmlns:cor="http://example.com/supplyCorrelation">
   <correlationSet name="PurchaseOrder"
    properties="cor:customerID␣cor:orderNumber" />
   <correlationSet name="Invoice"
    properties="cor:vendorID␣cor:invoiceNumber" />
</correlationSets>
```

It is worth noting that, in order to define correlation sets, WS-BPEL exploits the property and the propertyAlias constructs described in the previous section. In the example above, two correlation sets are defined: the former is named *PurchaseOrder* and it is joined to the properties *cor:customerID* and *cor:orderNumber* whereas the latter is named *Invoice* and it is joined with the properties *cor:vendorID* and *cor:invoiceNumber*.

<faultHandlers> and <eventHandlers> The tags *<faultHandlers>* and *<eventHandlers>* allow for the definition of the handlers which manages faults and events. Faults can be logically generated during the process execution or received within a message exchange whereas some event reactions can be programmed when a message reception or an alarm occur. Since faults and events are out of the topic of this thesis, here we do not present such a kind of constructs in detail.

activity Each business process has an initial activity and a terminating one. In the following we list all the activities and we present some usage examples for the ones which are relevant to the end of the understanding of this thesis.

- Activities which deal with communication
  - *receive*: it allows the business process to receive a message on an One-Way or Request-Response operation. Within a receive it is possible to define a correlation set for identifying the right instance to which route the incoming message. In the following we present an example of the receive activity:

    ```
    <receive partnerLink="purchasing"
     portType="lns:purchaseOrderPT"
     operation="sendPurchaseOrder"
     variable="PO"
     createInstance="yes">
    </receive>
    ```

    It is worth noting that the partnerLink, the portType and the operation on which the message reception is performed are declared. Moreover, the variable (*PO*) where the message content will be stored is defined. The attribute *createInstance* set to *yes* means that a new business process instance is created when the message is received.

– *reply*: it allows the business process to send a reply message in a Request-Response message exchange. The reply activities must be logically executed after a receive activity declared on the same operation. In the following an example of a reply activity is reported:

```
<reply partnerLink="purchasing"
portType="lns:purchaseOrderPT"
operation="sendPurchaseOrder"
variable="Invoice">
</reply>
```

In this case, the attribute *variable* identifies the variable from which the message content will be taken.

– *invoke*: it allows the business process to invoke a One-Way or a Request-Response operation of another business process. In the case of a Request-Response invocation the invoke activity will be blocked until the response message will be received. In the following a Request-Response invocation is presented.

```
<invoke partnerLink="shipping"
portType="lns:shippingPT"
operation="requestShipping"
inputVariable="shippingRequest"
outputVariable="shippingInfo">
</invoke>
```

It is worth noting that the attributes *inputVariable* and *outputVariable* specify the variables where the incoming message will be stored and from which will be taken the outcoming one respectively.

In general, within a communication primitive a correlation set may be defined in order to route a received message to the right instance depending on the contained values. In the following, we present an example where a receive activity is correlated with the correlation set *auctionIdentification*:

```
<receive name="acceptSellerInformation"
partnerLink="seller"
```

```
portType="as:sellerPT"
operation="submit"
variable="sellerData"
createInstance="yes">
 <correlations>
  <correlation set="auctionIdentification"/>
 </correlations>
</receive>
```

- Activities which deal with computational aspects:
  - *assign*: it allows the business process to assign a value or an expression result to a variable furthermore, the assign activity can also be used to assign an endpoint reference to a partnerLink. In the following we present an assign example where the tag <*copy*> specifies that some values must be copied from a source specified within the tag <*from*> to the target defined within the tag <*to*>. In this example, the source is an endpoint reference whereas the target is a partnerLink.

```
<assign>
 <copy>
  <from>
   <literal>
    <sref:service-ref>
     <addr:EndpointReference>
      <addr:Address>
       http://example.com/auction/RegistrationService/
      </addr:Address>
      <addr:ServiceName>
       as:RegistrationService
      </addr:ServiceName>
     </addr:EndpointReference>
    </sref:service-ref>
   </literal>
  </from>
  <to partnerLink="auctionRegistrationService" />
 </copy>
```

```
</assign>
```

- *validate*: it allows the business process to check the validation of the values contained within a variable w.r.t. the joined WSDL data definition.

- Activities which deal with activity composition:

  - *sequence*: it allows for the sequential composition of activities. In the following we present a sequence example where an invoke activity is performed after two assign activities:

```
<sequence>
 <assign>
  <copy>
   <from>$sellerData.endpointReference</from>
   <to partnerLink="seller" />
  </copy>
  <copy>
   <from>
    <literal>Thank you!</literal>
   </from>
   <to>$sellerAnswerData.thankYouText</to>
  </copy>
 </assign>
 <invoke name="respondToSeller"
  partnerLink="seller"
  portType="as:sellerAnswerPT"
  operation="answer"
  inputVariable="sellerAnswerData" />
</sequence>
```

  - *flow*: it allows for the parallel composition of activities. In the following we present an example where a sequence of an invoke and a receive activity are composed in parallel with a sequence of two invoke activities.

```
<flow>
 <sequence>
  <invoke partnerLink="shipping"
```

```
   portType="lns:shippingPT"
   operation="requestShipping"
   inputVariable="shippingRequest"
   outputVariable="shippingInfo">
   </invoke>
   <receive partnerLink="shipping"
   portType="lns:shippingCallbackPT"
   operation="sendSchedule"
   variable="shippingSchedule">
   </receive>
  </sequence>
  <sequence>
   <invoke partnerLink="invoicing"
   portType="lns:computePricePT"
   operation="initiatePriceCalculation"
   inputVariable="PO">
   </invoke>
   <invoke partnerLink="invoicing"
   portType="lns:computePricePT"
   operation="sendShippingPrice"
   inputVariable="shippingInfo">
   </invoke>
  </sequence>
</flow>
```

It is worth noting that the activities programmed to be executed in parallel can be synchronized by means of internal synchronization signals expressed by means of the tag <*links*>. The tags <*source*> and <*target*> allows for the specification of the activity which sends the synchronizing signal and the activity which receives it respectively. In the following we present the same example above where the signal *ship-to-invoice* is introduced. The source activity is the first invoke activity of the first sequence whereas the target is the last invoke activity of the second sequence.

```
<flow>
  <links>
   <link name="ship-to-invoice" />
  </links>
  <sequence>
   <invoke partnerLink="shipping"
   portType="lns:shippingPT"
   operation="requestShipping"
   inputVariable="shippingRequest"
   outputVariable="shippingInfo">
    <sources>
     <source linkName="ship-to-invoice" />
    </sources>
   </invoke>
   <receive partnerLink="shipping"
   portType="lns:shippingCallbackPT"
   operation="sendSchedule" variable="shippingSchedule">
   </receive>
  </sequence>
  <sequence>
   <invoke partnerLink="invoicing"
   portType="lns:computePricePT"
   operation="initiatePriceCalculation"
   inputVariable="PO">
   </invoke>
   <invoke partnerLink="invoicing"
   portType="lns:computePricePT"
   operation="sendShippingPrice"
   inputVariable="shippingInfo">
    <targets>
     <target linkName="ship-to-invoice" />
    </targets>
   </invoke>
  </sequence>
</flow>
```

It is important to highlight the fact that the tags <*source*> and <*target*> allow also for the specification of an inner guard (a boolean expression) which can block or not the sending and the receiving of the synchronizing signal.

– *pick*: it allows for the representation of non-deterministic choice among a set of events. Only the selected event will be executed whereas the other will be discarded. The events that can be programmed within a pick activity are the reception of a message, which resembles a receive activity, or an internal time alarm. In the following, we present an example where two message receptions and an alarm are programmed within a pick activity:

```
<pick>
 <onMessage partnerLink="buyer"
  portType="orderEntry"
  operation="inputLineItem"
  variable="lineItem">
 </onMessage>
 <onMessage partnerLink="buyer"
  portType="orderEntry"
  operation="orderComplete"
  variable="completionDetail">
 </onMessage>
 <onAlarm>
 <!-- set an alarm to go off
  3 days and 10 hours after the last order line -->
 <for>'P3DT10H'</for>
 </onAlarm>
</pick>
```

– *if*: it allows for the selection of exactly one activity among a collection of choices. It represents the usual *if then else* construct. An example where the *if* activity is used follows:

```
<if>
 <condition>
  bpel:getVariableProperty('stockResult','inventory:level') > 100
 </condition>
 <flow>
 <!-- perform fulfillment work -->
 </flow>
 <elseif>
  <condition>
```

```
   bpel:getVariableProperty('stockResult','inventory:level') >= 0
   </condition>
   <!-- perform elseif activities -->
  </elseif>
  <else>
   <!-- perform else activities -->
  </else>
</if>
```

– *while*: it allows for the expression of repeated activities depending on a condition. The condition is evaluated before the execution of the inner activities. A usage example follows:

```
<while>
 <condition>$orderDetails > 100</condition>
 <scope>...</scope>
</while>
```

– *repeatUntil*: as the while activity it allows for the expression of repeated activities. The condition is evaluated after the firs execution of the inner activities.

– *forEach*: it allows for the repetition of some activities. The inner activities are repeated a specified number of times.

- Activities which deal with activity scoping

  – *scope*: it allows for the definition of a nested activity scope where different partnerLinks, messageExchanges, variables, correlationSets, faultHandlers, compensationHandler, terminationHandler can be defined.

- Activities which deal with faults

  – *exit*: it allows for the immediate termination a business process instance.

  – *throw*: it allows for the generation of a fault within the business process

  – *rethorw*: it allows for the re-generation of a fault within the fault handler it is processing it.

  - *compensate*: it allows the business process to start a compensation activity within all the inner scopes.
  - *compensateScope*: it allows the business process to start a compensation activity on a specific inner scope.

- Others:
  - *wait*: it allows the business process to wait for a specified amount of time.
  - *empty*: no activities are performed when this activity is executed.
  - *extensionActivity*: it allows for the extension of WS-BPEL activiites with new kind of activities.

2.1.3.2 Abstract process

A WS-BPEL business process can be programmed as a so-called *abstract process*. An abstract process cannot be executed by an engine but it is exploited for representing a sort of behavioural skeleton of the business process. Such a kind of processes are usually exploited for supplying a view of a business process where only the observable actions are shown. Some *opaque* constructs are introduced for replacing executable statements with a non-observable ones. In particular it is possible to have an opaque construct for:

- an *activity*: an activity can be replaced with an opaque one in order to not specify its behaviour.
- an *expression*: an expression can be replaced with an opaque value. Such a kind of feature implicitly inroduces non-determinism within choice constructs where conditions are evaluated.
- an *attribute*: tag attributes can be replaced with an opaque value.

In the following we present an abstract process example where all the three kind of opaque constructs are used:

```
<process ...>
 <partnerLinks>
  <partnerLink name="homeInfoVerifier"
  partnerLinkType="##opaque"
  myRole="##opaque"
  partnerRole="##opaque">
  </partnerLink>
 </partnerLinks>
 <variables>
  <variable name="commonRequestVar" element="##opaque" />
 </variables>
 <sequence>
  <opaqueActivity template:createInstance="yes">
   <documentation>
   Pick an appraisal request from one of 3 customer referral channels.
   </documentation>
  </opaqueActivity>
  <assign>
   <documentation>
   Transform one of these 3 appraisal request into our own format.
   </documentation>
   <from opaque="yes" />
   <to variable="commonRequestVar" />
  </assign>
  <opaqueActivity>
   <documentation>
   Extract customer and housing info from our appraisal
   request into a message understood by our home info
   verification partner.
   </documentation>
  </opaqueActivity>
  <invoke partnerLink="homeInfoVerifier"
  operation="##opaque"
  inputVariable="##opaque"/>
 </sequence>
```

```
</process>
```

2.1.3.3 Two examples

In the following we present two examples reported in the WS-BPEL specification. The former describes a *Purchase order service* whereas the latter describes an *Auction service* process which allows for the management of an auction between a seller and a buyer.

Purchase order Service This example models a purchase order service which starts with a Request-Response operation *sendPurchaseOrder*. Between the request message and the response one it coordinates three activities in parallel: one deals with the price calculation process, one deals with the shipping activity and one deals with the production scheduling activity. It is worth noting that links among parallel threads are exploited for synchronizing the activities. No correlation sets are used within this example. In Fig. 2.2 we present the graphical representation of the service, given within the specifications, where rectangles represent activities, the smooth rectangle represents a parallel composition of the wrapped activities, dotted arrows represent a sequence of activities and arrows represent a synchronization between two activities.

In the following we report the XML code of the Purchase order service as it is presented within the specifications.

```
<wsdl:definitions
targetNamespace="http://manufacturing.org/wsdl/purchase"
xmlns:sns="http://manufacturing.org/xsd/purchase"
xmlns:pos="http://manufacturing.org/wsdl/purchase"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <wsdl:types>
  <xsd:schema>
   <xsd:import namespace="http://manufacturing.org/xsd/purchase"
   schemaLocation="http://manufacturing.org/xsd/purchase.xsd" />
  </xsd:schema>
```

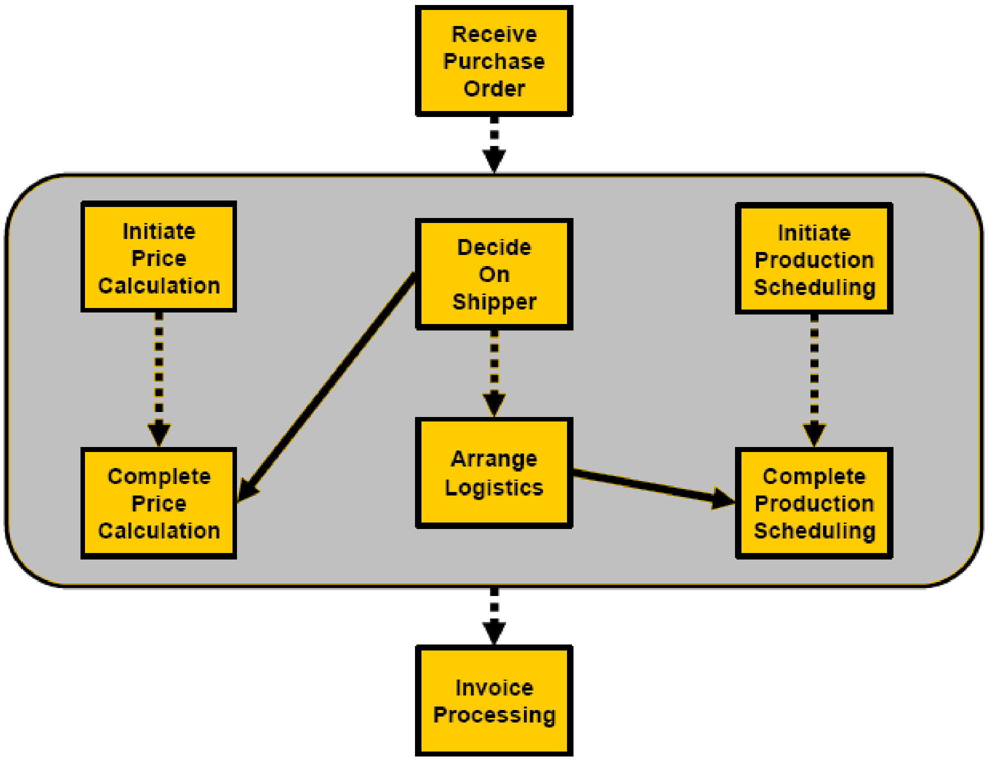
Receive Purchase Order
Initiate Price Calculation
Decide On Shipper
Initiate Production Scheduling
Complete Price Calculation
Arrange Logistics
Complete Production Scheduling
Invoice Processing

Figure 2.2: Purchase order service

```
</wsdl:types>
<wsdl:message name="POMessage">
   <wsdl:part name="customerInfo" type="sns:customerInfoType" />
   <wsdl:part name="purchaseOrder" type="sns:purchaseOrderType" />
</wsdl:message>
<wsdl:message name="InvMessage">
   <wsdl:part name="IVC" type="sns:InvoiceType" />
</wsdl:message>
<wsdl:message name="orderFaultType">
  <wsdl:part name="problemInfo" element= "sns:OrderFault␣" />
</wsdl:message>
<wsdl:message name="shippingRequestMessage">
  <wsdl:part name="customerInfo" element="sns:customerInfo" />
</wsdl:message>
<wsdl:message name="shippingInfoMessage">
  <wsdl:part name="shippingInfo" element="sns:shippingInfo" />
```

```
</wsdl:message>
<wsdl:message name="scheduleMessage">
 <wsdl:part name="schedule" element="sns:scheduleInfo" />
</wsdl:message>
<!-- portTypes supported by the purchase order process -->
<wsdl:portType name="purchaseOrderPT">
 <wsdl:operation name="sendPurchaseOrder">
  <wsdl:input message="pos:POMessage" />
  <wsdl:output message="pos:InvMessage" />
 <wsdl:fault name="cannotCompleteOrder"
 message="pos:orderFaultType" />
</wsdl:operation>
</wsdl:portType>
<wsdl:portType name="invoiceCallbackPT">
  <wsdl:operation name="sendInvoice">
   <wsdl:input message="pos:InvMessage" />
  </wsdl:operation>
</wsdl:portType>
<wsdl:portType name="shippingCallbackPT">
 <wsdl:operation name="sendSchedule">
  <wsdl:input message="pos:scheduleMessage" />
 </wsdl:operation>
</wsdl:portType>
<!-- portType supported by the invoice services -->
<wsdl:portType name="computePricePT">
 <wsdl:operation name="initiatePriceCalculation">
  <wsdl:input message="pos:POMessage" />
 </wsdl:operation>
 <wsdl:operation name="sendShippingPrice">
  <wsdl:input message="pos:shippingInfoMessage" />
 </wsdl:operation>
</wsdl:portType>
<!-- portType supported by the shipping service -->
<wsdl:portType name="shippingPT">
 <wsdl:operation name="requestShipping">
  <wsdl:input message="pos:shippingRequestMessage" />
```

```
    <wsdl:output message="pos:shippingInfoMessage" />
    <wsdl:fault name="cannotCompleteOrder"
    message="pos:orderFaultType" />
  </wsdl:operation>
 </wsdl:portType>
 <!-- portType supported by the production scheduling process -->
 <wsdl:portType name="schedulingPT">
  <wsdl:operation name="requestProductionScheduling">
    <wsdl:input message="pos:POMessage" />
  </wsdl:operation>
  <wsdl:operation name="sendShipingSchedule">
    <wsdl:input message="pos:scheduleMessage" />
  </wsdl:operation>
 </wsdl:portType>
 <plnk:partnerLinkType name="purchasingLT">
  <plnk:role name="purchaseService"
  portType="pos:purchaseOrderPT" />
 </plnk:partnerLinkType>
 <plnk:partnerLinkType name="invoicingLT">
  <plnk:role name="invoiceService"
  portType="pos:computePricePT" />
  <plnk:role name="invoiceRequester"
  portType="pos:invoiceCallbackPT" />
 </plnk:partnerLinkType>
 <plnk:partnerLinkType name="shippingLT">
  <plnk:role name="shippingService"
  portType="pos:shippingPT" />
 <plnk:role name="shippingRequester"
 portType="pos:shippingCallbackPT" />
 </plnk:partnerLinkType>
 <plnk:partnerLinkType name="schedulingLT">
  <plnk:role name="schedulingService"
  portType="pos:schedulingPT" />
 </plnk:partnerLinkType>
</wsdl:definitions>
```

```
<process name="purchaseOrderProcess"
targetNamespace="http://example.com/ws-bp/purchase"
xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
xmlns:lns="http://manufacturing.org/wsdl/purchase">
 <documentation xml:lang="EN">
 A simple example of a WS-BPEL process for handling a purchase
 order.
 </documentation>
 <partnerLinks>
  <partnerLink name="purchasing"
  partnerLinkType="lns:purchasingLT"
  myRole="purchaseService" />
  <partnerLink name="invoicing" partnerLinkType="lns:invoicingLT"
  myRole="invoiceRequester" partnerRole="invoiceService" />
  <partnerLink name="shipping" partnerLinkType="lns:shippingLT"
  myRole="shippingRequester" partnerRole="shippingService" />
  <partnerLink name="scheduling"
  partnerLinkType="lns:schedulingLT"
  partnerRole="schedulingService" />
 </partnerLinks>
 <variables>
  <variable name="PO" messageType="lns:POMessage" />
  <variable name="Invoice" messageType="lns:InvMessage" />
  <variable name="shippingRequest"
  messageType="lns:shippingRequestMessage" />
  <variable name="shippingInfo"
  messageType="lns:shippingInfoMessage" />
  <variable name="shippingSchedule"
  messageType="lns:scheduleMessage" />
 </variables>
 <faultHandlers>
  <catch faultName="lns:cannotCompleteOrder"
  faultVariable="POFault"
  faultMessageType="lns:orderFaultType">
   <reply partnerLink="purchasing"
   portType="lns:purchaseOrderPT"
```

```
    operation="sendPurchaseOrder" variable="POFault"
    faultName="cannotCompleteOrder" />
  </catch>
</faultHandlers>
<sequence>
  <receive partnerLink="purchasing" portType="lns:purchaseOrderPT"
  operation="sendPurchaseOrder" variable="PO"
  createInstance="yes">
   <documentation>Receive Purchase Order</documentation>
  </receive>
<flow>
  <documentation>
  A parallel flow to handle shipping, invoicing and
  scheduling
  </documentation>
  <links>
   <link name="ship-to-invoice" />
   <link name="ship-to-scheduling" />
  </links>
  <sequence>
   <assign>
    <copy>
     <from>$PO.customerInfo</from>
     <to>$shippingRequest.customerInfo</to>
    </copy>
   </assign>
   <invoke partnerLink="shipping" portType="lns:shippingPT"
    operation="requestShipping"
    inputVariable="shippingRequest"
    outputVariable="shippingInfo">
    <documentation>Decide On Shipper</documentation>
    <sources>
     <source linkName="ship-to-invoice" />
    </sources>
   </invoke>
   <receive partnerLink="shipping"
```

```
    portType="lns:shippingCallbackPT"
    operation="sendSchedule" variable="shippingSchedule">
   <documentation>Arrange Logistics</documentation>
    <sources>
     <source linkName="ship-to-scheduling" />
    </sources>
   </receive>
  </sequence>
  <sequence>
   <invoke partnerLink="invoicing"
    portType="lns:computePricePT"
    operation="initiatePriceCalculation"
    inputVariable="PO">
    <documentation>
     Initial Price Calculation
    </documentation>
   </invoke>
   <invoke partnerLink="invoicing"
    portType="lns:computePricePT"
    operation="sendShippingPrice"
    inputVariable="shippingInfo">
    <documentation>
     Complete Price Calculation
    </documentation>
    <targets>
     <target linkName="ship-to-invoice" />
    </targets>
   </invoke>
   <receive partnerLink="invoicing"
    portType="lns:invoiceCallbackPT"
    operation="sendInvoice" variable="Invoice" />
  </sequence>
  <sequence>
   <invoke partnerLink="scheduling"
    portType="lns:schedulingPT"
    operation="requestProductionScheduling"
    inputVariable="PO">
     <documentation>
      Initiate Production Scheduling
     </documentation>
    </invoke>
    <invoke partnerLink="scheduling"
    portType="lns:schedulingPT"
    operation="sendShippingSchedule"
    inputVariable="shippingSchedule">
     <documentation>
     Complete Production Scheduling
     </documentation>
     <targets>
      <target linkName="ship-to-scheduling" />
     </targets>
    </invoke>
   </sequence>
  </flow>
  <reply partnerLink="purchasing" portType="lns:purchaseOrderPT"
  operation="sendPurchaseOrder" variable="Invoice">
   <documentation>Invoice Processing</documentation>
  </reply>
  </sequence>
</process>
```

Auction Service This example models an auction service which waits for a seller and a buyer message for managing an auction identified by a specific ID. Both the seller and the buyer has to specify such an ID in order to be joined within the Auction service. A correlation set is used for identifying the instances within the process. In particular, the auction ID is exploited for denoting an instance and a property *auctionId* is defined in order to retrieve such a value within the different incoming messages. In Fig. 2.3 we report the graphical representation of the service which starts with a parallel composition between two receives, one from the buyer and one from the seller, and then continues with a sequence of activities. The process ends with the parallel composition of two invoke activities that it exploits for notifying both the buyer and the seller that the auction is succesfully terminated. Within the activities composed in sequence, the Auction service performs some internal computations which roughly model the management of the data related to the auction and then sends such information to an Auction Registration service by invoking it. Finally, it waits for a response from the Auction Registration service. For the sake of brevity, here we do not model the Auction Registration service because we are interested only to show how the correlation mechanism is concretely used in WS-BPEL. In order to understand the behaviour of such a service it is sufficient to know that it stores the received data and then it sends a response to the invoker.

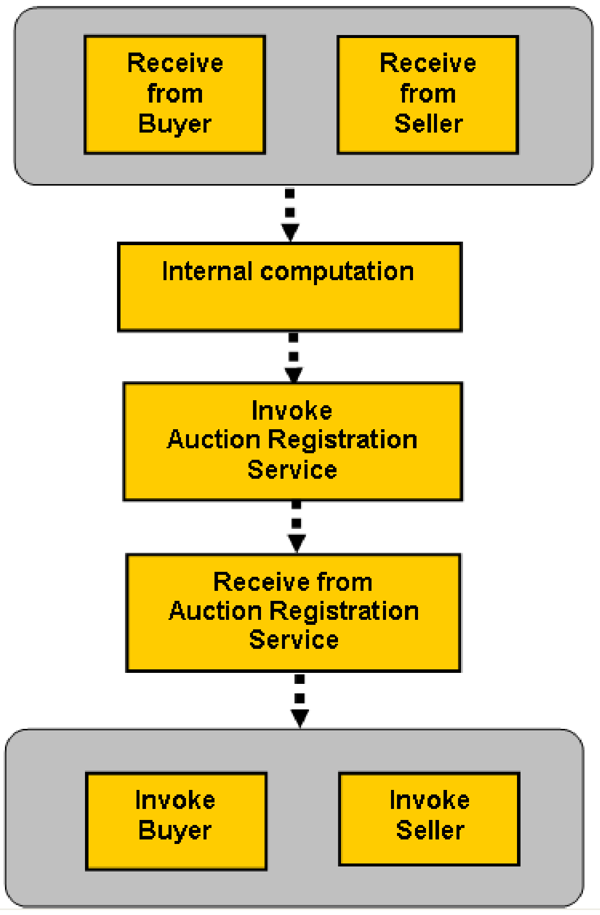

Figure 2.3: Auction service

In the following we report the WS-BPEL code of the process. It is worth noting that the two initial receive activities, which wait for a message from the buyer and the seller, initiate the correlation set. Since it is impossible to predict which receive activity will be executed firstly (that from the buyer or that form the seller) only the first received message initiates a new instance by setting a new correlation set whereas the second one will be routed to the right instance by correlating it. Such a feature is expressed within WS-BPEL, by exploiting the attribute *initiate="join"* within the tag *correlation* of the two received activities.

```
<wsdl:definitions
 targetNamespace="http://example.com/auction/wsdl/auctionService/"
 xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
 xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
 xmlns:tns="http://example.com/auction/wsdl/auctionService/"
 xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<!-- Messages for communication with the seller -->
<wsdl:message name="sellerData">
  <wsdl:part name="creditCardNumber" type="xsd:string" />
  <wsdl:part name="shippingCosts" type="xsd:integer" />
  <wsdl:part name="auctionId" type="xsd:integer" />
  <wsdl:part name="endpointReference" type="bpel:ServiceRefType" />
</wsdl:message>
<wsdl:message name="sellerAnswerData">
  <wsdl:part name="thankYouText" type="xsd:string" />
</wsdl:message>
<!-- Messages for communication with the buyer -->
<wsdl:message name="buyerData">
  <wsdl:part name="creditCardNumber" type="xsd:string" />
  <wsdl:part name="phoneNumber" type="xsd:string" />
  <wsdl:part name="ID" type="xsd:integer" />
  <wsdl:part name="endpointReference" type="bpel:ServiceRefType" />
</wsdl:message>
<wsdl:message name="buyerAnswerData">
  <wsdl:part name="thankYouText" type="xsd:string" />
</wsdl:message>
<!-- Messages for communication with the auction registration service -->
<wsdl:message name="auctionData">
```

```
 <wsdl:part name="auctionId" type="xsd:integer" />
 <wsdl:part name="amount" type="xsd:integer" />
 <wsdl:part name="auctionHouseEndpointReference"
 type="bpel:ServiceRefType" />
</wsdl:message>
<wsdl:message name="auctionAnswerData">
 <wsdl:part name="registrationId" type="xsd:integer" />
 <wsdl:part name="auctionId" type="xsd:integer" />
</wsdl:message>
<!-- PortTypes for interacting with the seller -->
<wsdl:portType name="sellerPT">
 <wsdl:operation name="submit">
  <wsdl:input message="tns:sellerData" />
  </wsdl:operation>
 </wsdl:portType>
 <wsdl:portType name="sellerAnswerPT">
  <wsdl:operation name="answer">
   <wsdl:input message="tns:sellerAnswerData" />
  </wsdl:operation>
 </wsdl:portType>
 <!-- PortTypes for interacting with the buyer -->
 <wsdl:portType name="buyerPT">
  <wsdl:operation name="submit">
   <wsdl:input message="tns:buyerData" />
  </wsdl:operation>
 </wsdl:portType>
 <wsdl:portType name="buyerAnswerPT">
 <wsdl:operation name="answer">
  <wsdl:input message="tns:buyerAnswerData" />
  </wsdl:operation>
  </wsdl:portType>
<!-- PortTypes for interacting with the
auction registration service -->
<wsdl:portType name="auctionRegistrationPT">
 <wsdl:operation name="process">
  <wsdl:input message="tns:auctionData" />
 </wsdl:operation>
</wsdl:portType>
<wsdl:portType name="auctionRegistrationAnswerPT">
 <wsdl:operation name="answer">
  <wsdl:input message="tns:auctionAnswerData" />
 </wsdl:operation>
</wsdl:portType>
<!-- Context type used for locating business process
via auction Id -->
<vprop:property name="auctionId" type="xsd:integer" />
<vprop:propertyAlias propertyName="tns:auctionId"
messageType="tns:sellerData" part="auctionId" />
<vprop:propertyAlias propertyName="tns:auctionId"
messageType="tns:buyerData" part="ID" />
<vprop:propertyAlias propertyName="tns:auctionId"
messageType="tns:auctionData" part="auctionId" />
<vprop:propertyAlias propertyName="tns:auctionId"
messageType="tns:auctionAnswerData" part="auctionId" />
<!-- PartnerLinkType for seller/auctionHouse -->
<plnk:partnerLinkType name="sellerAuctionHouseLT">
 <plnk:role name="auctionHouse" portType="tns:sellerPT" />
 <plnk:role name="seller" portType="tns:sellerAnswerPT" />
</plnk:partnerLinkType>
<!-- PartnerLinkType for buyer/auctionHouse -->
<plnk:partnerLinkType name="buyerAuctionHouseLT">
 <plnk:role name="auctionHouse" portType="tns:buyerPT" />
 <plnk:role name="buyer" portType="tns:buyerAnswerPT" />
</plnk:partnerLinkType>
<!-- Partner link type for auction house/auction
registration service -->
<plnk:partnerLinkType
name="auctionHouseAuctionRegistrationServiceLT">
 <plnk:role name="auctionRegistrationService"
 portType="tns:auctionRegistrationPT" />
 <plnk:role name="auctionHouse"
 portType="tns:auctionRegistrationAnswerPT" />
```

```
</plnk:partnerLinkType>
</wsdl:definitions>

<process name="auctionService"
targetNamespace="http://example.com/auction"
xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
xmlns:sref="␣http://docs.oasis-open.org/wsbpel/2.0/serviceref"
xmlns:addr="http://example.com/addressing"
xmlns:as="http://example.com/auction/wsdl/auctionService/">
<import importType="http://schemas.xmlsoap.org/wsdl/"
location="auctionServiceInterface.wsdl"
namespace="http://example.com/auction/wsdl/auctionService/" />
 <partnerLinks>
  <partnerLink name="seller"
  partnerLinkType="as:sellerAuctionHouseLT"
  myRole="auctionHouse"
  partnerRole="seller" />
 <partnerLink name="buyer"
 partnerLinkType="as:buyerAuctionHouseLT"
 myRole="auctionHouse"
 partnerRole="buyer" />
 <partnerLink name="auctionRegistrationService"
 partnerLinkType="as:auctionHouseAuctionRegistrationServiceLT"
 myRole="auctionHouse"
 partnerRole="auctionRegistrationService" />
 </partnerLinks>
 <variables>
  <variable name="sellerData"
  messageType="as:sellerData" />
  <variable name="sellerAnswerData"
  messageType="as:sellerAnswerData" />
  <variable name="buyerData"
  messageType="as:buyerData" />
  <variable name="buyerAnswerData"
  messageType="as:buyerAnswerData" />
  <variable name="auctionData"
  messageType="as:auctionData" />
  <variable name="auctionAnswerData"
  messageType="as:auctionAnswerData" />
 </variables>
 <correlationSets>
  <correlationSet name="auctionIdentification"
  properties="as:auctionId" />
 </correlationSets>
 <sequence>
 <!-- Process buyer and seller request concurrently
 Either one can create a process instance -->
 <flow>
 <!-- Process seller request -->
 <receive name="acceptSellerInformation"
 partnerLink="seller"
 portType="as:sellerPT"
 operation="submit"
 variable="sellerData"
 createInstance="yes">
 <correlations>
  <correlation set="auctionIdentification"
  initiate="join" />
  </correlations>
 </receive>
 <!-- Process buyer request -->
 <receive name="acceptBuyerInformation"
 partnerLink="buyer"
 portType="as:buyerPT"
 operation="submit"
 variable="buyerData"
 createInstance="yes">
  <correlations>
    <correlation set="auctionIdentification"
     initiate="join" />
  </correlations>
 </receive>
```

```
</flow>
<!-- Invoke auction registration service by setting the target
endpoint reference and setting my own endpoint reference
for call back and receiving the answer Correlation of
request and answer is via auction Id -->
<assign>
 <copy>
  <from>
   <literal>
    <sref:service-ref>
     <addr:EndpointReference>
      <addr:Address>
      http://example.com/auction/
      RegistrationService/
      </addr:Address>
      <addr:ServiceName>
      as:RegistrationService
      </addr:ServiceName>
     </addr:EndpointReference>
    </sref:service-ref>
   </literal>
  </from>
  <to partnerLink="auctionRegistrationService" />
 </copy>
 <copy>
  <from partnerLink="auctionRegistrationService"
  endpointReference="myRole" />
  <to>$auctionData.auctionHouseEndpointReference</to>
 </copy>
 <copy>
  <from>$sellerData.auctionId</from>
  <to>$auctionData.auctionId</to>
 </copy>
 <copy>
  <from>1</from>
  <to>$auctionData.amount</to>
  </copy>
 </assign>
 <invoke name="registerAuctionResults"
 partnerLink="auctionRegistrationService"
 portType="as:auctionRegistrationPT"
 operation="process"
 wsbpel-specification-draft-01
 inputVariable="auctionData" />
 <receive name="receiveAuctionRegistrationInformation"
 partnerLink="auctionRegistrationService"
 portType="as:auctionRegistrationAnswerPT"
 operation="answer"
 variable="auctionAnswerData">
  <correlations>
   <correlation set="auctionIdentification" />
  </correlations>
 </receive>
<!-- Send responses back to seller and buyer -->
 <flow>
<!-- Process seller response by setting the seller to
the endpoint reference provided by the seller
and invoking the response -->
  <sequence>
   <assign>
    <copy>
     <from>$sellerData.endpointReference</from>
     <to partnerLink="seller" />
    </copy>
    <copy>
     <from>
      <literal>Thank you!</literal>
     </from>
     <to>$sellerAnswerData.thankYouText</to>
    </copy>
   </assign>
   <invoke name="respondToSeller"
```

```
    partnerLink="seller"
    portType="as:sellerAnswerPT"
    operation="answer"
    inputVariable="sellerAnswerData" />
   </sequence>
<!-- Process buyer response by setting the buyer to
the endpoint reference provided by the buyer
and invoking the response -->
   <sequence>
    <assign>
     <copy>
      <from>$buyerData.endpointReference</from>
      <to partnerLink="buyer" />
     </copy>
     <copy>
      <from>
       <literal>Thank you!</literal>
      </from>
      <to>$buyerAnswerData.thankYouText</to>
     </copy>
     </assign>
     <invoke name="respondToBuyer"
     partnerLink="buyer"
     portType="as:buyerAnswerPT"
     operation="answer"
     inputVariable="buyerAnswerData" />
    </sequence>
   </flow>
  </sequence>
</process>
```

2.1.3.4 WS-BPEL engines

WS-BPEL engines architecture is not standardized and the implementations that exist are the result of a free interpretation of the WS-BPEL specification made by the different producers. A detailed description of these products and a comparison of the different features they supply, is out of the scope of this thesis. It is not clear, for example, if WS-BPEL orchestrators designed with different tools are compatible each other. Since in the next sections we provide a model for service engines, that are machinery able to execute services, here we are interested, for the sake of completeness, to list the most credited ones and, in particular, we would to focus on their architecture. The documentation about this topic lacks in details and it is often not available. For this reason, in the following we comment a rough architecture extracted from the web pages of the Active BPEL project [act] in order to catch the basic characteristics an engine is based upon.

- Open source projects:
  - *Active BPEL* [act]. It implements an engine ables to animate WS-BPEL processes which follows the specifications. It is written in Java and it is also equipped with a visual support for designing BPEL processes by following a graphical approach.
  - *Apache ODE* [Apaa]. It is a Java open source project which implements WS-BPEL specifications. It does not supply a visual support for designing processes.
- Commercial products:
  - *Oracle BPEL Process Manager* [ora]. It is a proprietary product developed by Oracle which supplies an execution environment for native WS-BPEL code and a visual tool for designing processes in a graphical way.
  - *IBM WebSphere* [web]. It is a complete infrastructure software for integrating web applications. It also supplies a WS-BPEL enigine for executing Web Services orchestrators.

In [act] a very short description of the Active BPEL architecture is provided. Here, we try to summarize the basic concepts it is based upon[3]:

[3]The reader who is interested in details about this topic may consult [act].

- An active BPEL process is composed by different activities which correspond to those defined within the WS-BPEL specifications.

- Activities can be *Basic activities* (e.g. receive, reply, invoke) or *Structured activities* (e.g. sequence, flow, pick)

- Activities are joined by links and they have their own state which describe their current behaviour (e.g. inactive, ready-to-execute, executing)

- Each activity is defined within a scope which contains the values of the variables, fault handlers, event handlers, compensation handlers, etc. Each WS-BPEL process has a global scope as defined within the specifications.

- An activity is a *start activity* if it initiates a new active BPEL process. When a start activity is triggered a new process is created.

- The executing receive activities are queued in order to wait for an incoming message.

- The receive queue contains also the incoming messages received from other services that did not match at the moment of the reception.[4] hese messages are queued until a timeout period passes.

- The engine dispatches incoming messages to the correct process instance.

- If there is correlation data, the engine tries to find the correct instance that matches the correlation data. On the contrary, if there is no correlation data and the request matches a start activity, a new process instance is created. In Fig. 2.4 we report the request dispatch flowchart provided by the web pages documentation of the project which is essentially self-descriptive.

[4]Such a feature is at the basis of the asynchronous communication which charactersize Web Services. We discuss this topic in the next section.

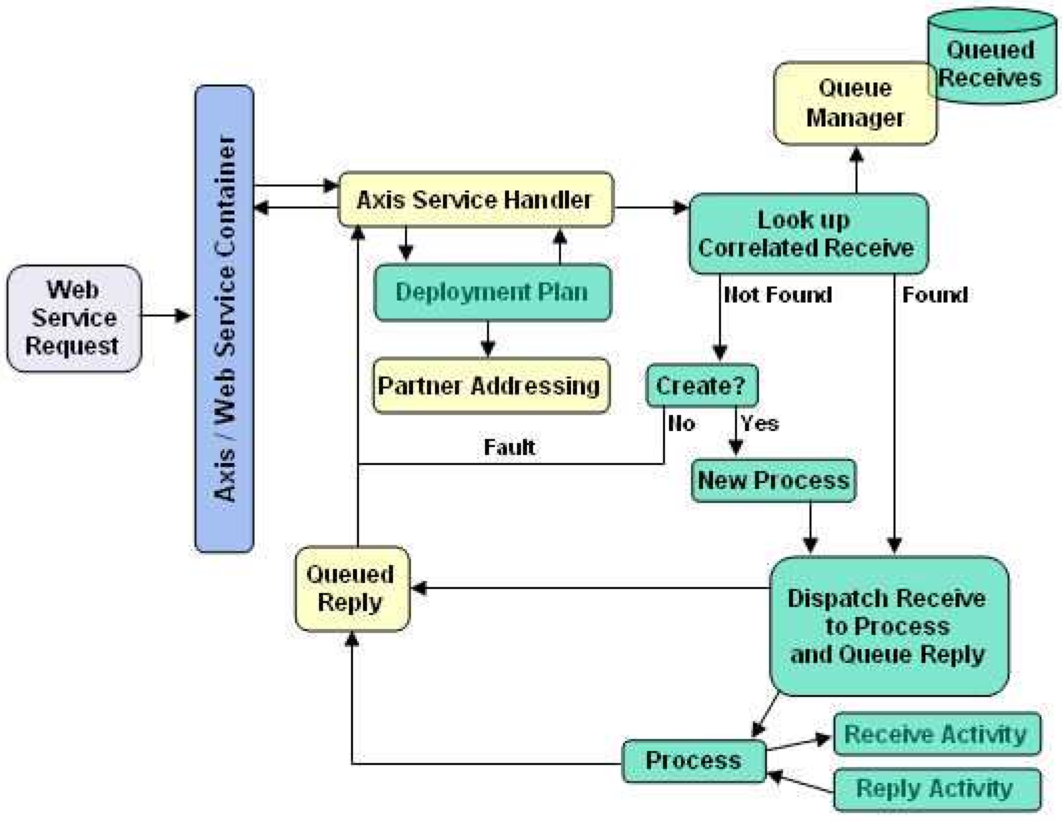

Figure 2.4: Request Dispatch Flowchart of the Active Bpel engine

2.1.4 WS-CDL

WS-CDL is a W3C specification where an XML-based language which deals with Web Service choreography description is defined. A choreography, differently from a WS-BPEL business process, cannot be executed. Thus, there are no engines able to animate a WS-CDL specification. WS-CDL indeed aims at providing a means which allows for the description of a services system from a global view point, focusing on the interactions among the involved participants. In the following we present the main characteristics of the language in order to give a good starting point for a WS-CDL comprehension. Otherwise, this section does not aim at being an exhaustive presentation of the language. The reader who is interested to focus on WS-CDL details may consult the specification [Worc, Worb]. The structure of a WS-CDL document follows:

```
<package
name="NCName"
author="xsd:string"?
```

```
version="xsd:string"?
targetNamespace="uri"
xmlns="http://www.w3.org/2005/10/cdl">

 <informationType/>*
  ...
 <roleType/>*
 <relationshipType/>*
 <participantType/>*
 <channelType/>*

 Choreography−Notation*
</package>
```

where a *package* allows for the specification of WS-CDL type definitions as *informationType*, *roleType*, *relationshipType*, *participantType* and *channelType*. Moreover, a package contains the description of the choreographies which refer to the specified types.

2.1.4.1 WS-CDL types

In general, WS-CDL types allows for the abstract representation of the participants involved in a choreoghraphy where each participant can enrole more roles with more than one behaviour. Furthermore, channel types define the nature of the means on which interactions between participants can be performed.

roleType A role type allows for the enumeration of all the behaviours exhibited by a role where a role abstractly represents a specific dialoguer within a system whereas a behaviour represents a specific task within a role. In general, a behaviour is joined to a WSDL document which represents the interface of the behaviour. In the following we present an example where a *BuyerRole* with a behaviour *BuyerBehaviour* is defined.

```
<roleType name="BuyerRole">
 <description type="documentation">
 Role for Buyer
 </description>
 <behavior name="BuyerBehavior" interface="BuyerBehaviorInterface">
```

```
   <description type="documentation">
   Behavior for Buyer Role
   </description>
  </behavior>
</roleType>
```

It is worth noting that tags *description* allows for the specification of human readable descriptions.

relationshipType A relationship type describes a conversational relation between two behaviours of two different roles. Intuitively we can consider a relationshipType as an abstract communication link between two roles which allows for the accomplishment of a specific task. In the following we present an example where a relationshipType between a Buyer and a Seller is defined:

```
<relationshipType name="Buyer2Seller">
  <description type="documentation">
  Buyer Seller Relationship
  </description>
  <roleType typeRef="tns:BuyerRole"/>
  <roleType typeRef="tns:SellerRole"/>
</relationshipType>
```

participantType In WS-CDL a participant abstractly represents an entity, usually a service, which enroles some of the declared roleTypes. A participant can enrole more than one role, this is the case, for example, of a service that interacts with different services. With regard to this, let us consider the case of a market service which interacts both with a client and a supplier. With respect to the client, the market will play the role of a seller whereas, with respect to the supplier, it will play the role of a customer. In the following we present a possible participant definition of such a kind of market service.

```
<participantType name="Market">
  <description type="documentation">
  Seller Participant
  </description>
  <roleType typeRef="tns:SellerRole"/>
  <roleType typeRef="tns:CustomerRole"/>
```

```
</participantType>
```

channelType A channelType describes where and how information between participantTypes can be exchanged. A channel is the abstract representation of the means on which a message exchange can be performed. A channelType is associated to a roleType and a behaviour. Within a channelType it is also possible to define the *identity* information which allows for the definition of the correlation data contained within a message exchange. In the following example a channelType within the role *SellerRole* is defined. The information *id* is exploited for identifying the current conversation session of that channel.

```
<channelType name="Buyer2SellerChannel">
 <description type="documentation">
 Buyer to Seller Channel Type
 </description>
 <roleType typeRef="tns:SellerRole"/>
  ...
 <identity type="primary">
   <token name="tns:id"/>
 </identity>
</channelType>
```

It is worth noting that the tag *<token>* represents a way for referencing a piece of data within a message or a variable.

informationType The informationType allows for the definition of the type of variables and messages. In general the informationType joins a name to a type defined within a WSDL document or an XML Schema type. In the following we present an informationType definition where we suppose that the type *pns:purchaseOrderMessage* has been previously defined within a WSDL Message type.

```
<informationType name="purchaseOrder" type="pns:purchaseOrderMessage"/>
```

2.1.4.2 WS-CDL choreography definitions

A choreography is defined within the tag <*choreography*>. In order to correctly define a choreography it is necessary to declare the relationshipTypes involved, the used variables and the interactions to perform. Moreover, it is possible to define two blocks: the *exceptionBlock* and the *finalizerBlock*. The former deals with the interactions to perform when an exception occur whereas the latter allows for the specification of the finalizer activities for a choreography. The description of these two blocks is out of the scope of this work, the reader who is ineterested in this topic may consult [Worc].

```
<choreography  name="NCName"  ...>
 <relationship  type="QName" />+
 variableDefinitions?
 activities
 <exceptionBlock  name="NCName" />
 <finalizerBlock  name="NCName" />
</choreography>
```

varibleDefinition The *variableDefinition* tag allows for the definition of the variables used within a choreography. In general, each variable is declared to belong to a specific roleType and an informationType or a channelType can be joined to it depending on the fact that the variable describes an information or a channel. In the following we present the definition of a channel variable and an information one:

```
<variableDefinitions>
 <variable name="Buyer2SellerC"
 channelType="tns:Buyer2SellerChannel"
 roleTypes="tns:BuyerRole␣tns:SellerRole">
  <description  type="documentation">
  Channel Variable
  </description>
 </variable>
 <variable name="quoteRequest"
  informationType="tns:QuoteRequestType"
  roleTypes="tns:BuyerRole␣tns:SellerRole">
   <description  type="documentation">
   Request Message
   </description>
 </variable>
</variableDefinitions>
```

activities Here we distinguish the activities into two different categories: basic activities and structured activities.

- basic activities:
  - *interaction*: the interaction is the basic construct of WS-CDL and it describes a message exchange between two roles. An interaction is always performed on a channel and it is finished when the message exchange completes successfully. Some elements and attributes of the tag interaction deserve to be commented. The elements <*participate*> defines the participants involved in the interactions by specifying the attributes *relationshipType*, *fromRole* and *toRole*. The element <*exchange*> defines the variables exchanged by the sender and the receiver. It contains two sub-elements <*send*> and <*receive*> where the former defines the information sent by the sender whereas the latter defines the information received by the receiver. The attribute *action* of the exchange tag defines the direction of the interaction and has two possible values: *request* or *respond*. In the former case the interaction is performed from the role fromRole, which is the sender, to the role toRole which is the receiver. In the latter case the toRole is the sender and the fromRole is the receiver. One or two exchange tags can be defined within the tag interaction. If there are two exchange elements one must have action equal to request and the other must assume the value respond. The former defines the information exchange during the request interaction and the latter during the response one. The element <*record*> allows for the specification of some variables which can be updated when the interaction is performed. As far as the attributes of the interaction tag are concerned, *channelVariable* and *operation* define the WS-CDL channel and the WSDL operation on which the interaction is performed whereas the attribute *align* defines if the interaction must be aligned or not w.r.t. the updating of the variables specified within the record element. If the attribute align is set to true both the dialoguers should be assured of the variable changement availability. In the following we present an example of an interaction where there are two exchange tags defined one for the request message exchange and the other for the response one.

```
<interaction name="createPO"
 channelVariable="tns:retailer-channel"
 operation="handlePurchaseOrder" >
  <participate relationshipType="tns:ConsumerRetailerRelationship"
  fromRoleTypeRef="tns:Consumer" toRoleTypeRef="tns:Retailer"/>
  <exchange name="request"
  informationType="tns:purchaseOrderType" action="request">
   <send variable="cdl:getVariable('tns:purchaseOrder','','')" />
   <receive variable="cdl:getVariable('tns:purchaseOrder','','')"
    recordReference="record-the-channel-info" />
  </exchange>
  <exchange name="response"
  informationType="purchaseOrderAckType" action="respond">
   <send variable="..." />
   <receive variable="..." />
  </exchange>
</interaction>
```

– *assign*: the assign activity allows for the creation or the changement of a variable value within a roleType. The syntax is intuitive. In the following we present an assignment example where the value of the variable *CustomerAddres*, located within the message *PurchaseOrderMsg* and retrieved by using the function *getVariable*, is stored within the variable *CustomerAddres*.

```
<assign roleType="tns:Retailer">
  <copy name="copyAddressInfo">
   <source variable="cdl:getVariable('PurchaseOrderMsg'
,'','/PO/CustomerAddress')" />
   <target variable="cdl:getVariable('CustomerAddress','','')" />
  </copy>
</assign>
```

– *silent action*: the silent actions allows for the specification of a non-observable action which is performed within a roleType. In the following we present an example of a silent action performed within the role *Buyer*:

```
<silentAction roleType="Buyer" />
```

- structured activities:
  - *workunit*: the workunit allows for the collection of some activities which has to be performed when some constraints are satisfied. The attributes *guard* and *repeat* allows for the specification of boolean expressions where the former is a condition to satisfy for performing the workunit content whereas the latter is a condition to satisfy for repeating the activities contained within the workunit. In the following we present an example where an interaction is defines within a guarded workunit.

```
<workunit
name="drawdown"
guard="cdl:getVariable('Chosen','','','Broker')='A'" >
 <interaction name="drawdownInteraction"
  channelVariable="tns:CreditRequestor"
  operation="drawDown">
   ...
 </interaction>
</workunit>
```

  - *sequence*: the sequence allows for sequentially composing activities. In the following we present an example where two interactions are defined to be performed in sequence:

```
<sequence>
 <interaction name="inter1">
   ...
 </interaction>
 <interaction name="inter2">
   ...
 </interaction>
</sequence>
```

  - *parallel*: the parallel activity allows for the parallel composition of activities. An example follows:

```
<parallel>
 <interaction name="inter1">
   ...
 </interaction>
 <interaction name="inter2">
   ...
 </interaction>
</parallel>
```

– *choice*: the choice construct allows for the specification of a choice among the activities specified within its body where only one of them can be performed. It is worth noting that it is possible to specify workunits inside a choice construct in order to express a choice with guard conditions. Only the workunits which satisfy their guards can be involved within the choice race. In the following, we present an example where a choice between two interactions is defined:

```
<choice>
 <interaction name="processGoodCredit"
  channelVariable="goodCredit−channel"
  operation="doCredit">
  ...
 </interaction>
 <interaction name="processBadCredit"
  channelVariable="badCredit−channel"
  operation="doBadCredit">
  ...
 </interaction>
<choice>
```

2.1.4.3 A WS-CDL example

In the following we present an example presented in [Worb] where there are two roles involved, the *BuyerRole* and the *SellerRole*. The BuyerRole starts for an bartering by sending a request on the operation *getQuote*. The SellerRole sends a response to this request by commnunicating the *quoteResponse*. Depending on the BuyerRole (there are no definitions about the internal choice of the BuyerRole), it can accept the quote and then sending an *orderRequest* or, contrarly, it does not accept it and it asks for an updating of the quote on the operation *updateQuote*. The bartering is repeated until the variable *BarteringDone* is set to false. Such a variable change its value when the BuyerRole requests for an *orderRequest*.

```
<?xml version="1.0" encoding="UTF-8"?>
<package name="IntermediateExample" ...>
   <description type="documentation">
   Intermediate Example: Simple Bartering Process
   </description>

   <informationType name="QuoteRequestType"
   type="primer:QuoteRequestMsg">
       <description type="documentation">
       Quote Request Message
       </description>
   </informationType>
   <informationType name="QuoteResponseType"
   type="primer:QuoteResponseMsg">
       <description type="documentation">
       Quote Response Message
       </description>
   </informationType>
   <informationType name="QuoteResponseFaultType"
   type="primer:QuoteResponseFaultMsg">
       <description type="documentation">
       Quote Response Fault Message
       </description>
   </informationType>
   <informationType name="IdentityType"
   type="xsd:string">
       <description type="documentation">
       Identity Attribute
       </description>
   </informationType>
   <informationType name="URI" type="xsd:uri">
       <description type="documentation">
       Reference Token For Channels
```

```
        </description>
    </informationType>
    <informationType name="OrderRequestType"
    type="primer:OrderRequestMessage">
        <description type="documentation">
        Order Request Message
        </description>
    </informationType>
    <informationType name="OrderResponseType"
    type="primer:OrderResponseMessage">
        <description type="documentation">
        Order Response Message
        </description>
    </informationType>
    <informationType name="Boolean"
    type="xsd:boolean">
        <description type="documentation">
        Boolean type for use in loop control
        </description>
    </informationType>

    <token name="id" informationType="tns:IdentityType">
        <description type="documentation">
        Identity token
        </description>
    </token>
    <token name="URI" informationType="tns:URI">
        <description type="documentation">
        Reference Token for Channels
        </description>
    </token>

    <tokenLocator tokenName="tns:id" i
    nformationType="tns:QuoteRequestType" query="/quote/@id">
        <description type="documentation">
        Identity for QuoteRequestType
        </description>
    </tokenLocator>
    <tokenLocator tokenName="tns:id"
    informationType="tns:QuoteResponseType" query="/quote/@key">
        <description type="documentation">
        Identity for QuoteResponseType
        </description>
    </tokenLocator>
    <tokenLocator tokenName="tns:id"
    informationType="tns:QuoteResponseFaultType"
        query="/quote/@key">
        <description type="documentation">
        Identity for QuoteResponseFaultType
        </description>
    </tokenLocator>
    <tokenLocator tokenName="tns:id"
    informationType="tns:OrderRequestType"
    query="/order/@orderId">
        <description type="documentation">
        Identity for OrderRequestType
        </description>
    </tokenLocator>
    <tokenLocator tokenName="tns:id"
    informationType="tns:OrderResponseType" query="/order/@orderId">
        <description type="documentation">
        Id for OrderResponseType
        </description>
    </tokenLocator>

    <roleType name="BuyerRole">
        <description type="documentation">
        Role for Buyer
        </description>
        <behavior name="BuyerBehavior"
        interface="BuyerBehaviorInterface">
            <description type="documentation">
            Behavior for Buyer Role
            </description>
        </behavior>
    </roleType>
    <roleType name="SellerRole">
        <description type="documentation">
        Role for Seller
        </description>
        <behavior name="SellerBehavior"
        interface="SellerBehaviorInterface">
            <description type="documentation">
            Behavior for Seller
            </description>
        </behavior>
    </roleType>

    <relationshipType name="Buyer2Seller">
        <description type="documentation">
        Buyer Seller Relationship
        </description>
        <roleType typeRef="tns:BuyerRole"/>
        <roleType typeRef="tns:SellerRole"/>
    </relationshipType>

    <participantType name="Seller">
        <description type="documentation">
        Seller Participant
        </description>
        <roleType typeRef="tns:SellerRole"/>
    </participantType>
    <participantType name="Buyer">
        <description type="documentation">
        Buyer Participant
        </description>
        <roleType typeRef="tns:BuyerRole"/>
    </participantType>
```

```
<channelType name="Buyer2SellerChannel">
    <description type="documentation">
    Buyer to Seller Channel Type
    </description>
    <roleType typeRef="tns:SellerRole"/>
    <reference>
        <token name="tns:URI"/>
    </reference>
    <identity type="primary">
        <token name="tns:id"/>
    </identity>
</channelType>


<choreography name="IntermediateChoreography" root="true">
    <description type="documentation">
    The Choreography for the degenerate use case
    </description>

    <relationship type="tns:Buyer2Seller"/>

    <variableDefinitions>
        <variable name="Buyer2SellerC"
        channelType="tns:Buyer2SellerChannel"
        roleTypes="tns:BuyerRole␣tns:SellerRole">
            <description type="documentation">
            Channel Variable
            </description>
        </variable>
        <variable name="quoteRequest"
        informationType="tns:QuoteRequestType"
        roleTypes="tns:BuyerRole␣tns:SellerRole">
            <description type="documentation">
            Request Message
            </description>
        </variable>
        <variable name="quoteResponse"
        informationType="tns:QuoteResponseType"
        roleTypes="tns:BuyerRole␣tns:SellerRole">
            <description type="documentation">
            Response Message
            </description>
        </variable>
        <variable name="faultResponse"
        informationType="tns:QuoteResponseFaultType"
        roleTypes="tns:BuyerRole␣tns:SellerRole">
            <description type="documentation">
            Fault Message
            </description>
        </variable>
        <variable name="orderRequest"
        informationType="tns:OrderRequestType"
        roleTypes="tns:BuyerRole␣tns:SellerRole">
            <description type="documentation">
            Order Request Message
            </description>
        </variable>
        <variable name="barteringDone"
        informationType="tns:Boolean" roleTypes="tns:SellerRole">
            <description type="documentation">
            Variable used to control the loop exit from
            </description>
        </variable>
    </variableDefinitions>

    <sequence>
        <assign roleType="tns:SellerRole">
            <description type="documentation">
            Initialise Loop Variable
            </description>
            <copy name="setBarteringDone">
                <description type="documentation">
                Set barteringDone to false
                </description>
                <source expression="false()"/>
                <target
                variable="cdl:getVariable('barteringDone','','')"/>
            </copy>
        </assign>
        <interaction name="QuoteElicitation" operation="getQuote"
            channelVariable="tns:Buyer2SellerC">
            <description type="documentation">
            Elicit a quote from the seller
            </description>
            <participate relationshipType="tns:Buyer2Seller"
            fromRoleTypeRef="tns:BuyerRole"
            toRoleTypeRef="tns:SellerRole"/>
            <exchange name="QuoteRequest"
            informationType="tns:QuoteRequestType"
            action="request">
             <description type="documentation">
             Quote Request Message Exchange
             </description>
              <send
              variable="cdl:getVariable('quoteRequest','','')"/>
              <receive
              variable="cdl:getVariable('quoteRequest','','')"/>
            </exchange>
            <exchange name="QuoteResponse"
            informationType="tns:QuoteResponseType"
            action="respond">
                <description type="documentation">
                Quote Response Message Exchange
                </description>
                <send
                variable="cdl:getVariable('quoteResponse','','')"/>
                <receive
                variable="cdl:getVariable('quoteResponse','','')"/>
            </exchange>
            <exchange name="QuoteResponseFault"
             informationType="tns:QuoteResponseFaultType"
             action="respond" faultName="InvalidProductFault">
              <send
               variable="cdl:getVariable('faultResponse','','')"
               causeException="TerminalFailure"/>
              <receive
               variable="cdl:getVariable('faultResponse','','')"
               causeException="TerminalFailure"/>
            </exchange>
        </interaction>
        <workunit name="WhileBarteringIsNotFinished"
            guard="cdl:getVariable("barteringDone","","")=true()"
            repeat="true()">
            <description type="documentation">
            While barteringDone is false
            </description>
            <choice>
                <sequence>
                    <description type="documentation">
                    Accept the quote and place the order
                    </description>
                    <interaction name="QuoteAccept"
                    operation="order"
                    channelVariable="tns:Buyer2SellerC">
                        <description type="documentation">
                        The Buyer accepts the quote and orders
                        the goods based on the last price
                        </description>
                        <participate relationshipType="tns:Buyer2Seller"
                        fromRoleTypeRef="tns:BuyerRole"
                        toRoleTypeRef="tns:SellerRole"/>
                        <exchange name="OrderRequest"
                        informationType="tns:OrderRequestType"
                        action="request">
                         <send
                          variable="cdl:getVariable('orderRequest'...)"/>
                         <receive
                          variable="cdl:getVariable('orderRequest'...)"/>
                        </exchange>
                    </interaction>
                    <assign roleType="tns:SellerRole">
                      <description type="documentation">
                      Break out of the loop
                      </description>
                       <copy name="setBarteringDone">
                        description type="documentation">
                        Set barteringDone to true
                        </description>
                        <source expression="true()"/>
                        <target
                        variable="cdl:getVariable('barteringDone'...)"/>
                       </copy>
                    </assign>
                </sequence>
                <sequence>
                    <description type="documentation">
                    Reject the quote and ask for a new quote
                    </description>
                    <interaction name="QuoteReelicitation"
                    operation="updateQuote"
                    channelVariable="tns:Buyer2SellerC">
                     <description type="documentation">
                     Barter based on previous quote
                     </description>
                     <participate relationshipType="tns:Buyer2Seller"
                     fromRoleTypeRef="tns:BuyerRole"
                     toRoleTypeRef="tns:SellerRole"/>
                      <exchange name="QuoteRequest"
                      informationType="tns:QuoteRequestType"
```

```
                                    action="request">
                                    <description type="documentation">
                                    Quote re-request based on amended quoteRequest
                                    </description>
                                     <send
                                     variable="cdl:getVariable('quoteRequest'...)"/>
                                     <receive
                                     variable="cdl:getVariable('quoteRequest'...)"/>
                                    </exchange>
                                    <exchange name="QuoteResponse"
                                    informationType="tns:QuoteResponseType"
                                    action="respond">
                                      <send
                                      variable="cdl:getVariable('quoteResponse'...)"/>
                                      <receive
                                      variable="cdl:getVariable('quoteResponse'...)"/>
                                    </exchange>
                                 </interaction>
                            </sequence>
                      </choice>
                </workunit>

          </sequence>
    </choreography>
</package>
```

2.2 Process Algebras

Process algebras [Mil89, Hoa85, BW90] are algebraic languages which support compositional description of concurrent systems and the formal verification of their properties. The basic elements of any process algebra are its actions, which represent activities carried out by the system being modeled, and its operators which are used to compose algebraic descriptions. In this section, we recall the theory of the process algebras by introducing CCS [Mil89] that will serve as a basis for the development of the process calculi introduced in the rest of the thesis.

2.2.1 CCS syntax

Here we introduce the syntax of CCS. Let $Names$ be the set of channel names ranged over by $a, b, c, \ldots$ and let $CoNames$ be the set of the channel co-names ranged over by $\overline{a}, \overline{b}, \overline{c}, \ldots$ where $\overline{\overline{a}} = a$. Let Act be the set of actions ranged over by α defined as follows:

$$Act ::= Names \cup CoNames \cup \{\tau\}$$

where τ is the action, called *silent action*, which denotes *unobservable* activities. Finally, let $Const$ be a set of constants ranged over by $A, B, \ldots$ Let $\mathcal{P}$ be the set of processes generated by the following syntax where $L \subseteq Act - \{\tau\}$.

$$P ::= \mathbf{0} \mid \alpha.P \mid P/L \mid P + P \mid P \parallel P \mid A$$

- The *null term* "$\mathbf{0}$" is the term that cannot execute any action.
- The *action prefix operator* "$\alpha.\_$" denotes the sequential composition of an action and a term. Term $\alpha.P$ can execute action α and then behaves as term P.
- The *abstraction operator* "$\_/L$" makes actions unobservable. Term P/L behaves as term P except that each executed action α is hidden, i.e. turned into τ, whenever $\alpha \in L$. This operator provides a means to encapsulate or ignore information.
- The *alternative composition operator* "$\_ + \_$" expresses a non-deterministic choice between two terms. Term $P_1 + P_2$ behaves as either term P_1 or term P_2 depending on whether an action of P_1 or an action of P_2 is executed.
- The *parallel composition operator* "$\_ \parallel \_$" expresses the concurrent execution of two terms according to the following synchronization discipline: two actions $\alpha \in Names$ and $\beta \in CoNames$ can synchronize if and only if $\alpha = \overline{\beta}$.
- Let partial function $Def : Const \nrightarrow \mathcal{P}$ be a set of *constant defining equations* of the form $A \triangleq P$. It is worth noting that recursive definitions like "$A \triangleq \alpha.\beta.A$" can also

be denoted by means of a *recursion operator* "$\mathrm{rec}X.\_$", where the meaning of "$\mathrm{rec}X.P$" is the same as that of "$A \triangleq P\{A/X\}$", with "$E\{A/X\}$" being the term obtained from P by replacing A for X. The two constructs are shown to be equivalent in [Mil89].

In order to guarantee the correctness of recursive definitions, we restrict ourselves to terms that are closed and guarded w.r.t. Def, i.e. those terms such that each constant occurring in them has a defining equation and appears in the context of an action prefix operator. This rules out undefined constants, meaningless definitions such as $A \triangleq A$, and infinitely branch terms such as $A \triangleq \alpha.\mathbf{0} \parallel A$ whose executable actions cannot be computed in finite time. Let us denote by "$\_ \equiv \_$" the syntactical equality between terms and by "$\_st\_$" the relation subterm-of.

Definition 2.1 *The term* $P < A := P' >$ *obtained from* $P \in \mathcal{P}$ *by replacing each occurrence of* A *with* P'*, where* $A \stackrel{\triangle}{=} P' \in \mathrm{Def}$*, is defined by induction on the syntactical structure of* P *as follows:*

$$\begin{array}{rcl}
\boldsymbol{0} < A := P' > & \equiv & \boldsymbol{0} \\
(\alpha.P) < A := P' > & \equiv & \alpha.P < A := P' > \\
P/L < A := P' > & \equiv & P < A := P' > /L \\
(P_1 + P_2) < A := P' > & \equiv & P_1 < A := P' > +P_2 < A := P' > \\
(P_1 \parallel P_2) < A := P' > & \equiv & P_1 < A := P' >\parallel P_2 < A := P' > \\
B < A := P' > & \equiv & \begin{cases} P' \ \text{if } B \equiv A \\ B \ \ \text{if } B \not\equiv A \end{cases}
\end{array}$$

Definition 2.2 *The set of terms obtained from* $P \in \mathcal{P}$ *by repeatedly replacing constants by the right hand side terms of their defining equations in* Def *is defined by*

$$\mathrm{Subst}_{\mathrm{Def}}(P) = \bigcup_{n \in \boldsymbol{N}} \mathrm{Subst}^{n}_{\mathrm{Def}}(P)$$

where

$\mathrm{Subst}^{n}_{\mathrm{Def}}(P) =$

$$= \begin{cases} \{E\} & \text{if } n = 0 \\ \{F \in \mathcal{P} \mid F \equiv G < A := P' > \wedge G \in \mathrm{Subst}^{n-1}_{\mathrm{Def}}(P) \wedge A \ \mathrm{st} \ G \wedge A \stackrel{\triangle}{=} P' \in \mathrm{Def}\} & \text{if } n > 0 \end{cases}$$

Definition 2.3 *The set of constants occurring in* $P \in \mathcal{P}$ *w.r.t.* Def *is defined by*

$$\mathrm{Const}_{\mathrm{Def}}(P) = \{A \in \mathrm{Const} \mid \exists F \in \mathrm{Subst}_{\mathrm{Def}}(P).A \ \mathrm{st} \ F\}$$

Definition 2.4 *A term* $P \in \mathcal{P}$ *is closed and guarded w.r.t.* Def *if and only if for all* $A \in \mathrm{Const}_{\mathrm{Def}}(P)$

- A *is equipped in* Def *with defining equations* $A \stackrel{\triangle}{=} P'$*, and*
- *there exists* $F \in \mathrm{Subst}_{\mathrm{Def}}(P')$ *such that, whenever an instance of a constant* B *satisfies* $B \ \mathrm{st} \ F$*, then the same instance satisfies* $B \ \mathrm{st} \ \alpha.G \ \mathrm{st} \ F$.

we denote with $\mathcal{G}$ *the set of terms in* $\mathcal{P}$ *that are closed and guarded w.r.t.* Def.

2.2.2 Rooted Labelled Transition Systems

In this section we present the definition of labelled transition system together with some related notions. These mathematical models, which are essentially state transition graphs, are commonly adopted when defining the semantics for a process algebra in the operational style [Plo81].

Definition 2.5 *A rooted labelled transition system (LTS) is a tuple*

$$(S, U, \longrightarrow, s_0)$$

where:

- *S is a set whose elements are called states.*
- *U is a set whose elements are called labels.*
- *$\longrightarrow \subseteq S \times U \times S$ is called transition relation.*
- *$s_0 \in S$ is called the initial state.*

In the graphical representation of a LTS, states are drawn as black dots and transitions are drawn as arrows between pairs of states with the appropriate labels; the initial state is pointed to by an unlabeled arrow.
Below we recall two notions of equivalence defined for LTSs. The former, isomorphism, realtes two LTSs if they are structurally equal. This is formalized by requiring the existence of a label preserving relation which is bijective, i.e. a bijection between the two state spaces sich that any pair of corresponding states have identically labelled transitions toward any pair of corresponding states. The latter equivalence, bisimilarity, is coarser than isomorphism as it relates also LTSs which are not structurally equal provided that they are able to simulate each other. This is formalized by requiring the existence of a label preserving relation between the two state spaces which is not necessarily bijective.

Definition 2.6 *Let $Z_k = (S_k, U, \longrightarrow_k, s_{0k})$, $k \in \{1, 2\}$, be two LTSs.*

- *Z_1 is isomorphic to Z_2 if and only if there exists a bijection $\beta : S_1 \longrightarrow S_2$ such that:*

- $\beta(s_{01}) = s_{02}$
- *for all* $s, s' \in S_1$ *and* $u \in U$

$$s \xrightarrow{u}_1 s' \Longleftrightarrow \beta(s) \xrightarrow{u}_2 \beta(s')$$

- Z_1 *is bisimilar to* Z_2 *if and only if there exists a relation* $\mathcal{R} \subseteq S_1 \times S_2$ *such that:*
  - $(s_{01}, s_{02}) \in \mathcal{R}$
  - *for all* $(s_1, s_2) \in \mathcal{R}$ *and* $u \in U$
    * *whenever* $s_1 \xrightarrow{u}_1 s_1'$*, then* $s_2 \xrightarrow{u}_2 s_2'$ *and* $(s_1', s_2') \in \mathcal{R}$
    * *whenever* $s_2 \xrightarrow{u}_2 s_2'$*, then* $s_1 \xrightarrow{u}_1 s_1'$ *and* $(s_1', s_2') \in \mathcal{R}$

2.2.3 Operational semantics

The semantics of the presented process algebra is defined following the structured operational approach [Plo81]. This means that inference rules are given for each operator which defines an abstract interpreter for the language. More precisely, the semantics is defined through the transition relation $\longrightarrow$ which is the least subset of $\mathcal{G} \times \mathrm{Act}\mathcal{G}$ satisfying the inference rules is Table 2.1. Such rules, which formalize the meaning of each operator informally presented in the previous Section, yield LTSs where states are in correspondance with terms and transitions are labelled with actions. Given a term P, the outgoing transitions of the state corresponding to P are generated by proceeding by induction on the syntactical structure of P applying at each step the appropriate semantics rule until an action prefix operator is encountered or no rule can be used. This can be done in finite time because of the restriction to closed and guarded terms.
We recall that the abstraction operator and the parallel composition operator are called static operators because they appear also in the derivative term of the consequence of the related semantics rules. By contrary, the acton prefix operator and and the alternative composition operator are classified as dynamic operators. We say that a term is sequential if every occurrence of static operators is within the scope of an action prefix operator. It is worth noting that the LTS underlying $P \in \mathcal{G}$ is finite if all of the subterms of terms in $\mathrm{Subst}_{\mathrm{Def}}(P)$ whose outermost operator is static contains no recursive constants.

Definition 2.7 *The operational interleaving semantics of* $P \in \mathcal{G}$ *is the LTS*

$$[P] = (S_P, \mathrm{Act}, \rightarrow_P, P)$$

where:

- S_P *is the least subset of* $\mathcal{G}$ *such that:*
  - $P \in S_P$.
  - *if* $S_1 \in S_P$ *and* $P_1 \xrightarrow{\alpha} P_2$*, then* $P_2 \in S_P$.
- $\rightarrow_P$ *is the restriction of* $\rightarrow$ *to* $S_P \times \mathrm{Act} \times S_P$.

2.2.4 Bisimulation equivalence

In this section we define a notion of equivalence for the presented process algebra and we provide its equational and logical characterizations. The purpose of such an equivalence is to relate those therms representing systems which, though structurally equivalent, behave the same from the point of view of an external observer. Following [Mil89], the equivalence is given in the bisimulation style.

Definition 2.8 *A relation* $\mathcal{R} \subseteq \mathcal{G} \times \mathcal{G}$ *is a strong bisimulation if and only if, whenever* $(P_1, P_2) \in \mathcal{R}$*, then for all* $\alpha \in \mathrm{Act}$*:*

- *whenever* $P_1 \xrightarrow{\alpha} P_1'$*, then* $P_2 \xrightarrow{\alpha} P_2'$ *and* $(P_1', P_2') \in \mathcal{R}$
- *whenever* $P_2 \xrightarrow{\alpha} P_2'$*, then* $P_1 \xrightarrow{\alpha} P_1'$ *and* $(P_1', P_2') \in \mathcal{R}$

The union of all the strong bisimulations can be shown to be an equivalence relation which coincides with the largest strong bisimulation.

2.3 General assumptions

This section is devoted to discuss some general assumptions this work of thesis is based upon. In particular, we discuss asynchronous communications and faults and errors management.

(Action prefix)

$$\alpha.P \xrightarrow{\alpha} P$$

(Abstraction 1)

$$\frac{P \xrightarrow{\alpha} P'}{P/L \xrightarrow{\alpha} P'/L} \quad \text{if} \alpha \notin L$$

(Abstraction 2)

$$\frac{P \xrightarrow{\alpha} P'}{P/L \xrightarrow{\tau} P'/L} \quad \text{if} \alpha \in L$$

(Alternative 1)

$$\frac{P_1 \xrightarrow{\alpha} P'}{P_1 + P_2 \xrightarrow{\alpha} P'}$$

(Alternative 2)

$$\frac{P_2 \xrightarrow{\alpha} P'}{P_1 + P_2 \xrightarrow{\alpha} P'}$$

(Parallel 1)

$$\frac{P_1 \xrightarrow{\alpha} P_1'}{P_1 \parallel P_2 \xrightarrow{\alpha} P_1' \parallel P_2}$$

(Parallel 2)

$$\frac{P_2 \xrightarrow{\alpha} P_2'}{P_1 \parallel P_2 \xrightarrow{\alpha} P_1 \parallel P_2'}$$

(Synchro)

$$\frac{P_1 \xrightarrow{a} P_1' \quad P_2 \xrightarrow{\overline{a}} P_2'}{P_1 \parallel P_2 \xrightarrow{\tau} P_1 \parallel P_2'}$$

(Def)

$$\frac{P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'} \quad \text{if} A \triangleq P$$

Table 2.1: Operational semantics for CCS

2.3.1 Asynchronous communications

Service Oriented Computing message exchanges are intrinsecally asynchronous that is messages are not necessarily received when they are sent. The reason is twofold:

- In general, message exchanges are managed by a middleware infrastructure which underlies Service Oriented applications that does not guarantee that messages are always delivered when they are sent.
- It cannot be assumed that messages are processed by the applications when they are delivered to their middleware infrastructure but they can be queued before they are consumed. This is the case, for example, of an active BPEL engine which receives a message without a corresponding queued receive activity that is waiting for it.

In Fig. 2.5 we summarize a message exchange architecture between two applications where two engines and/or two middleware infrastructures underlie the applications and dispatch messages to them.

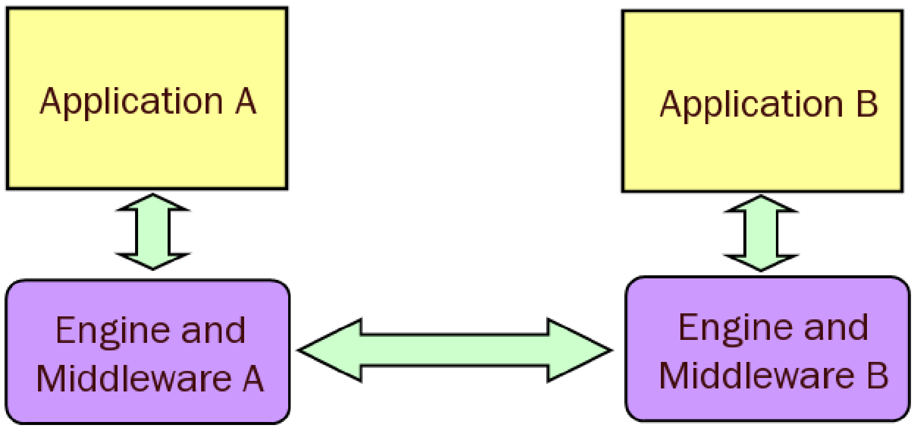

Figure 2.5: Message exchange protocol stack

In general when a message is sent from an application A to an application B, firstly the message is delivered to the engine/middleware infrastructure of application A, then it is communicated to the engine/middleware infrastructure of the application B and finally, it is delivered to the application B. Asynchronous communication deals with the fact that a message can be sent by the sender that can continue its activities without knowing if it is received by the receiver. The receiver can consume the message in a different moment. In the following, we abstract away from asynchronous communication when we will deal with SOCK. This is due to the fact that, with SOCK, we want to focus on design and composition aspects of Service Oriented Computing without focusing on any kind of verification issue. This choice will allow us to present a lighter semantics for SOCK in order to reason about the main concepts it is based upon. On the contrary, as far as the bipolar framework in concerned, we model asynchronous communication in the orchestration calculus, which is a subpart of SOCK, because we are interested to verify the conformance relation between a choreography and an orchestrated system. In this case, asynchronous communication plays a fundamental role and it can alter the conformance test which is the cornerstone of the bipolar framework. It is worth noting that, intuitively, it is possible to enhance the semantics rules of SOCK in order to consider asynchronous communication by following the same approach we will adopt in the orchestration calculus. In this sense, we consider asynchronous communcation exhaustively modelled.

2.3.2 Error and Faults

For the sake of this thesis, we do not deal with errors and faults because we intend to focus only on the main concepts Service Oriented Computing is based upon. In SOC different errors and faults can be considered:

- *Computational errors*: they deal with internal application errors raised by computational primitives such as, e.g. division by zero. We assume that no computational errors occur.
- *Network errors*: they deal with the fact that a message cannot be delivered because the network is not available. Here we assume that messages are always delivered.
- *Application faults*: they deal with logic faults that can be programmed at the level of the application. Such a kind of faults, in Service Oriented Computing, are usually

related to compensation mechanisms for restoring the system during a long running transaction. Here, we do not provide any specific primitive for dealing with such a kind of issues.

2.3.3 Starting application

In this section we introduce a simple distinction among Service Oriented Computing applications. In particular, we distinguish between *service* and *starting application*. Such a classification does not have a strong impact on the models and languages we develop in this thesis but it simply allows us to reason about services system by differentiating between those applications that *serve* from those that *fire* (i.e. start) a system execution. On the one hand, a service always starts its activities when it is invoked by another application, that is to say that each service is always triggered by an external message sent to it whereas, on the other hand, a starting application is able to start its activities by sending a message to another service. Someone can argue that the client/server paradigm can well describe such a classification without introducing new concepts, but there is another observation that has to be taken into account: the key fact of service composition is that *a composition of services is a service*. The client/server paradigm indeed, it is always defined between two dialoguers, whereas here we also consider a system of services. Since a service is always trigerred by an external message, also a system obtained as a composition of services will start its activities by receiving one or more messages from other applications. Summarizing, a starting application is able to initiate the execution of a services system by sending a message that also will start its own activities. In Fig. 2.6 we summarize such a distinction between starting application and service by means of a simple example where a services system is represented by a configuration of dominos cards whereas the starting application is depicted by a red ball able to start the game. The dominos cards space placement indeed, represent the dependencies among the services of the system that can start its execution only if it is fired by an external application (the red ball). In Chapter 3, we exploit SOCK for formalizing both services and starting applications.

2.3.4 Bipolar approach with no sessions

Sessions are an interesting Service Oriented Computing topic. Several Web Services specifications deal with service sessions (called also instances) and the session identification mechanism (the correlation set) is a particular characteristic of the SOC paradigm. Here, we deal with sessions within the SOCK calculus. In particular, in SOCK, we will define a session as an execution path of a service behaviour where the service behaviour is a means for describing all the possible execution paths of a service. On the contrary, as far as the bipolar approach is considered, at the state of the art, we do not deal with the session issue. For this reason, we do not consider sessions within both the choreography language and orchestration one. In our future works, we will investigate such a topic also in the bipolar approach setting.

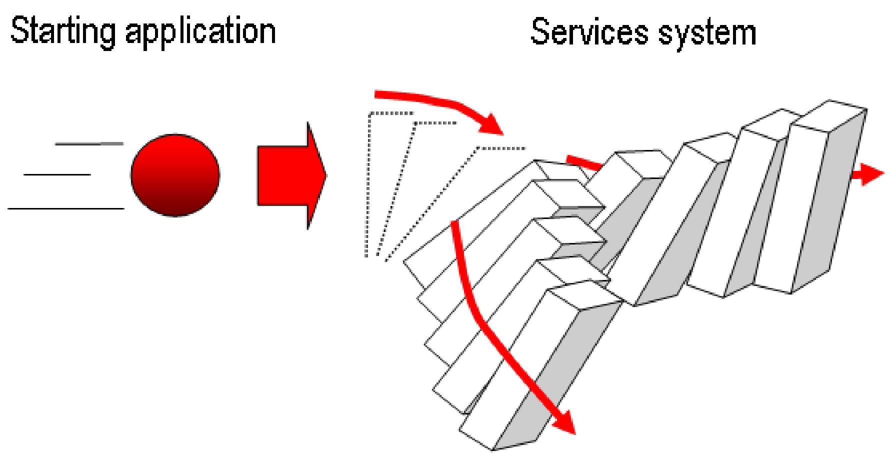

Figure 2.6: Starting application and services system

Part I

From the informal specifications to a general model for Service Oriented Computing

Chapter 3

SOCK: Service Oriented Computing Kernel

In this chapter we present SOCK (Service Oriented Computing Kernel) which is a formal language, inspired by WS-BPEL, equipped with a formal semantics. The main contribution of SOCK is twofold. On the one hand it allows us to distinguish and formalize some fundamental concepts of Service Oriented Computing such as the design of a service behaviour, its deployment in an executing enviroment and the composition of services within a system. For each of these topics, by means of SOCK, we are able to precisely define the basic mechanisims they are characterized by and it will be possible to reason about them in a formal way. On the other hand, we intend to promote SOCK as a full concrete and formalized SOC language for dealing with all the aspects of service design and composition. To this end, in Chapter 10, we will present an implementation of a subpart of SOCK, called JOLIE, which supplies an easy tool for programming services by means of a simple and intuitive syntax that resembles those of C and Java.

SOCK is three-layered structured and it allows for the distinction of three main concepts that characterize SOC systems: the *service behaviour*, the *service engine* and the *services system*. The service behaviour deals with the design of the behaviour of the service, the service engine deals with the actual deployment of a service behaviour within a machinery ables to execute it and the services system deals with the composition of service engines within a system. Each of them is represented by means of a specific calculus, where the service behaviour one underlies the service engine calculus and the service engine one underlies the services system calculus. The three-level layering allows us to reason about the different aspects of Service Oriented Computing in a separate way. Such a kind of approach is new and it is not provided by any Web Service specification (e.g. WS-BPEL). In particular, the distinction between service behaviour and service engine is not trivial and it allows us to reason about the main characteristics a language for describing the behaviour of a service has to satisfy and the different features a service engine has to provide. At the best fo our knowledge, no formal and informal specification which describe orchestrator engines exist and, in this sense, SOCK can be considered as a first attempt towards a complete standard specification for orchestration engines. Furthermore, by considering also the services system calculus, the SOCK three-level layering approach supplies a complete formal framework for dealing with both the design and composition issues. From the technical point of view, the *service behaviour calculus* supplies all the communication primitives and workflow constructs for describing the internal behaviour of the service. As we discuss later, the semantics of the service behaviour calculus is expressed in terms of a labelled transition system [Kel76] that allows us to introduce the concept of service behaviour *session*. Given a service behaviour indeed, a session is represented by an execution path of its lts. The *service engine calculus* is built on top of the previous one and it is composed of two main parts: the *declaration* and the *execution environment*. The former part allows both for the specification of the service behaviour whose sessions have to be executed and for the specification of the deployment characteristics the service engine has to supply, whereas the latter environment provides a means for actually executing sessions. In particular, the declaration specifies the way the sessions will be executed by programming three orthogonal service engine features: the *execution modality*, the *persistent state* flag and the *correlation sets*. The execution modality deals with the possibility to execute a service in a sequential order or in a concurrent one; the persistent state flag allows to declare if each session (of the service engine) has its own independent state or if the state is shared among all the sessions of the same service engine; the correlation sets allows for the specification of the correlation data which allow us to identify the different sessions that are executing within a service engine. Finally, the services system calculus allows for the composition of different service engines in order to obtain a system where services can interact each other.

The semantics of the calculi are defined in terms of labelled transition systems (lts for short) and they are organized as follows. There are six lts layers:

1. service behaviour lts layer
2. service engine state lts layer
3. service engine correlation lts layer
4. service engine execution modality lts layer
5. service engine location lts layer
6. services system lts layer

The first layer is the lowest one. Each lts layer catches the actions raised by the underlying one and will enable or disable them. If an action is enabled by an lts layer, it will be raised to the overlying one. The service behaviour lts layer describes all the possible execution paths generated by a service behaviour. The service engine state lts layer defines the rule for joining a service behaviour with a service engine local state. The service engine correlation lts layer deals with correlation set mechanism, the service engine execution modality lts layer represents rules for executing sessions concurrently or in a sequential order and the service engine location lts layer deals with the rules for deploying a service engine at a specific location. Finally, the services system lts layer deals with a composed service engine system.

3.1 Service behaviour calculus

This section is devoted to present the service behaviour calculus. Before introducing it, we discuss some important issues such as the external input and output actions and the service locations.

3.1.1 External input and output actions

The external input and output actions deal with those actions that are exploited by the service behaviour for communicating with other services. Communication in service oriented computing is always a peer-to-peer communication. In accordance with the interaction patterns proposed in [BDtH, BB05], the primitives related to such a kind of actions are called *operations* which explicitly model those of WSDL. Each operation is described by a name and an *interaction modality*. There are four kinds of peer-to-peer interaction modalities divided into two groups:

- Operations which supply a service functionality, *Input operations*:
  - *One-Way*: it is devoted to receive a request message.
  - *Request-Response*: it is devoted to receive a request message which implies a response message to the invoker.
- Operations which request a service functionality, *Output operations*:
  - *Notification*: it is devoted to send a request message.
  - *Solicit-Response*: it is devoted to send a request message which requires a response message.

The input operations are published by a service behaviour in order to receive messages on them. The output operation, on the contrary, are exploited for sending messages to the input ones exhibited by the service behaviour to invoke. Here we group the operations into *single message operations* and *double message operations*. The former deal with the One-Way and the Notification operations whereas the latter with the Request-Response and the Solicit-Response ones. Let $\mathcal{O}$ and $\mathcal{O}_R$ be two mutually disjoint sets of operation names where the former represent the single message operation names and the latter the double message ones. Let $InOp = \{(o, ow) \mid o \in \mathcal{O}\} \cup \{(o_r, rr) \mid o_r \in \mathcal{O}_R\}$ be the set containing all the input operations where ow and rr represent One-Way and Request-Response operations, respectively. Let $OutOp = \{(o, n) \mid o \in \mathcal{O}\} \cup \{(o_r, sr) \mid o_r \in \mathcal{O}_R\}$ be the set containing all the output operations where n and sr denote Notification and Solicit-Response operations. Let $Op = InOp \cup OutOp$ be the set of all the possible operations. Informally, we represent a One-Way (o, ow) with the symbol o and a Notification (o, n) with the symbol $\overline{o}$. In the same way, we represent a Request-Response operation (o_r, rr) with the symbol o_r whereas we represent a Solicit-Response operation (o_r, sr) with the symbol $\overline{o_r}$. We say that two operations o and $\overline{o}'$ are *dual* if $o = o'$. Analogously, we say that two operations o_r and $\overline{o_r}'$ are *dual* if $o_r = o_r'$. In Table 3.1 we briefly summarize the operation classification by reporting dual operations within the same row.

Input operations	**Formal and informal representation**	**Output operations**	**Formal and informal representation**
One-Way	$(o, ow) \in InOp$ o	Notification	$(o, n) \in OutOp$ $\overline{o}$
Request-Response	$(o_r, rr) \in InOp$ o_r	Solicit-Response	$(o_r, sr) \in OutOp$ $\overline{o_r}$

Table 3.1: Operation classification

3.1.1.1 Locations

In our framework locations represent the address (let it be a logical or a physical one) where a service is located. In order to perform an output operation it is fundamental to make explicit both the operation name and the location of the receiver in order to achieve a correct message delivery. In the following, locations will appear into the output operation primitives of the service behaviour calculus and, since they deal with the external communication, they will be exploited into the services system calculus for synchronizing external inputs with the corresponding output ones. Formally, let Loc be a finite set of location names ranged over by l.

3.1.2 The syntax

In the following we present the Syntax of the calculus devoted to represent services. Let $Signals$ be a set of signal names exploited for synchronizing processes in parallel within a service behaviour. Let Var be a set of variables ranged over by x, y, z and Val, ranged over by v, be a generic set of values on which it is defined a total order relation. We exploit the notations $\vec{x} = \langle x_0, x_1, ..., x_i \rangle$ and $\vec{v} = \langle v_0, v_1, ..., v_i \rangle$ for representing tuples of variables and values respectively. Let k ranges over $Var \cup$ Loc where $Var \cap$ Loc $= \emptyset$. The syntax follows:

$P, Q, \ldots ::=$	processes	
$\mathbf{0}$		null process
$\overline{\epsilon}$		output
ϵ		input
$x := e$		assignment
$\chi ? P : Q$		if then else
$P; P$		sequence
$P \mid P$		pararallel
$\sum^{+}_{i \in W} \epsilon_i; P_i$		non-det. choice
$\chi \rightleftharpoons P$		iteration

$\overline{\epsilon} \quad ::=$	output	
$\overline{s}$		output signal
$\overline{o}@k(\vec{x})$		Notification
$\overline{o_r}@k(\vec{x}, \vec{y})$		Solicit-Response

ϵ ::=		input
	s	input signal
	$o(\vec{x})$	One-Way
	$o_r(\vec{x}, \vec{y}, P)$	Request-Response

We denote with SC the set of all possible processes ranged over by P and Q. $\mathbf{0}$ is the null process. Outputs can be a signal $\bar{s}$, a notification $\bar{o}@k(\vec{x})$ or a solicit-response $\overline{o_r}@k(\vec{x}, \vec{y})$ where $s \in Signals$, $o \in \mathcal{O}$ and $o_r \in \mathcal{O}_R$, $k \in Var \cup \mathrm{Loc}$ represents the receiver location which can be explicit or represented by a variable[1], $\vec{x}$ is the tuple of the variables which store the information to send and $\vec{y}$ is the tuple of variables where, in the case of the solicit-response, the received information will be stored. Dually, inputs can be an input signal s, a one-way $o(\vec{x})$ or a Request-Response $o_r(\vec{x}, \vec{y}, P)$ where $s \in Signals$, $o \in \mathcal{O}$ and $o_r \in \mathcal{O}_R$, $\vec{x}$ is the tuple of variables where the received information are stored whereas $\vec{y}$ is the tuple of variables which contain the information to send in the case of the Request-Response; finally P is the process that has to be executed between the request and the response. $x := e$ assigns the result of the expression e to the variable x. For the sake of brevity, we do not present the syntax for representing expressions, we assume that they include all the arithmetic operators, values in Val and variables. $\chi?P : Q$ is the *if_then_else* process, where χ is a logic condition on variables whose syntax is:

$$\chi ::= x \leq e \mid e \leq x \mid \neg\chi \mid \chi \wedge \chi$$

It is worth noting that conditions such as $x = v$, $x \neq v$ and $v_1 \leq x \leq v_2$ can be defined as abbreviations; P is executed only if the condition χ is satisfied, otherwise Q is executed. $P; P$, $P \mid P$ represent sequential and parallel composition respectively, whereas $\sum_{i \in W}^{+} \epsilon_i; P_i$ is the non-deterministic choice restricted to be guarded on inputs. Such a restriction is due to the fact that we are not interested to model internal non-determinism in service behaviour. Our calculus indeed aims at supplying a basic language for designing service behaviours where designers have a full control of the internal machinery and the

[1]It is worth noting that, in order to fullfil location mobility, k must be a variable and it has to contain a location. Location mobility will be discussed deeply in Section 4

only non-predictable choices are those driven by the external message reception. Finally, $\chi \rightleftharpoons P$ is the construct to model guarded iterations.

3.1.3 Semantics.

As far as the semantics is concerned, here we consider the extension of SC which includes also the the terms $\bar{o}_r@l(\vec{x}), \bar{o}_r@z(\vec{x})$ and $o_r(\vec{x})$. These terms allows us to limit the semantics rules for the Request-Response message exchange mechanism. In the semantics indeed, we will consider the response message as a One-Way message exchange as well. It is worth noting that the service behaviour calculus does not deal with the actual values of variables and locations but it models all the possible execution paths for all the possible variable values and locations. The semantics follows this idea by means of an infinite set of actions where external inputs, external outputs and assignment actions report all the value substitutions for both variables and locations except the actions $\bar{o}@l(\vec{v}/\vec{x}), \bar{o}_r@l(\vec{v}/\vec{x})$ and $\overline{o_r}@l(\vec{v}/\vec{x}, \vec{y})$ where locations are defined. It is worth noting that each external input actions and the output actions related to the request message[2] in a Request-Response message exchange, report the location where the primitive is performed that is the location where the service behaviour is actually executed. Such a location is not known at the level of service behaviour but it will be joined at the level of the service engine; within the action it is always followed by the symbol : (e.g. $l\colon o(\vec{v}/\vec{x})$) and it will be fundamental, at the level of services system, for synchronizing the communcation among different service engines. Formally, let ω range over $\mathcal{O} \cup \mathcal{O}_R$ and let Act, defined as $Act = In \cup Out \cup Internal$, be the set of actions ranged over by a where:

$In = \{$

$l\colon \omega(\vec{v}/\vec{x}),$	One-Way input
$l'\colon o_r(\vec{v}/\vec{x}, \vec{y}, P)@l$	Request-Response input

$\}$

[2]In the case of the Request-Response, the location of the sender must be included within the action because it will be exploited by the receiver for sending the reply.

$\mathrm{Out} = \{$

$\overline{\omega}@l/z(\vec{v}/\vec{x})$,	One-Way output
$\overline{\omega}@l(\vec{v}/\vec{x})$,	located One-Way output
$l'{:}\overline{o_r}@l/z(\vec{v}/\vec{x},\vec{y})$,	Solicit-Response output
$l'{:}\overline{o_r}@l(\vec{v}/\vec{x},\vec{y})$	located Solicit-Response output

$\}$

$\mathrm{Internal} = \{$

s,	input signal
$\bar{s}$,	output signal
$x := v/e$,	assignment
$\chi?$,	satisfied condition
$\neg\chi?$	not satisfied condition

$\}$

The action *One-Way input* $l{:}\ \omega(\vec{v}/\vec{x})$, represents the reception of a request message on a One-Way operation or the reception of a response message on a Solicit-Response operation; l is the location of the receiver, ω is the operation name on which the message is received and $\vec{x}$ is the vector of variables which are updated with the received values $\vec{v}$. The action *Request-Response input* $l'{:}\ o_r(\vec{v}/\vec{x},\vec{y},P)@l$ represents the reception of request message on a Request-Response operation where l' is the location of the receiver, o_r is the operation name, $\vec{x}$ is the vector of variables which are updated with the received values $\vec{v}$, $\vec{y}$ is the vector of variables from which the response values will be taken, P is the process to execute between the request and the response message and l is the location of the sender. The actions *One-Way output* $\overline{\omega}@l/z(\vec{v}/\vec{x})$ and *located One-Way output* $\overline{\omega}@l(\vec{v}/\vec{x})$ represent the sending of a request message by exploiting a Notification operation or the sending of a response message in a Request-Response operation. The former deals with an action related to a process where the receiver location l is contaned within the variable z whereas the latter is related to a process where the location l is specified as a constant; ω is the operation name on which the message is sent and $\vec{x}$ is the vector of variables from which the sent values $\vec{v}$ are read. The actions *Solicit-Response output* $l'{:}\overline{o_r}@l/z(\vec{v}/\vec{x},\vec{y})$ and *located Solicit-Response output* $l'\colon \overline{o_r}@l(\vec{v}/\vec{x}, \vec{y})$ represent the sending of a request message from a Solicit-Response operation. The former deals with an action related to a process where the receiver location l is contaned within the variable z whereas the latter is related to a process where the location l is specified as a constant; l' represents the location of the sender, $\vec{x}$ is the vector of variables from which the sent values $\vec{v}$ are read and $\vec{y}$ is the vector of variables which will be updated with the value received within the response message. The actions *input signal* s and *output signal* $\overline{s}$ represent the reception and the sending of a signal s which allows for the synchronization among parallel processes, respectively. The action *assigment* $x := v/e$ represent an assignment where the variable x is updated with the value v that is the result of the evaluation of the expression e. Finally, the actions *satisfied condition* $\chi?$ and *not satisfied condition* $\neg\chi?$ represent a positive or a negative evaluation of the condition χ, respectively.

3.1.3.1 The labelled transition system

We define $\rightarrow \subseteq SC \times Act \times SC$ as the least relation which satisfies the axioms and rules of Table 3.2 and closed w.r.t. $\equiv$, where $\equiv$ is the least congruence relation satisfying the axioms at the end of Table 3.2. The rules are divided into axioms and rules for defining composition operators that are quite standard. Rules ONE-WAYOUTLOC and REQ-OUTLOC deals with output operations where the location l is explicit whereas rules ONE-WAYOUT and REQ-OUT deal with output operations where the location is represented by the variable z. Rule REQIN produces a process which executes P and then performs a notification joined with the sender location l. It is worth noting that the actual location will be joined at the level of the services system lts layer where synchronizations among service engines are defined. Rule SYNCHRO defines the synchronization between signals which allows us to exploit them for synchronizing parallel processes of the same service behaviour. Finally, rules ITERATION and NOT ITERATION model iteration in a way which resembles that of imperative programming.

(In) $s \xrightarrow{s} \mathbf{0}$

(Out) $\bar{s} \xrightarrow{\bar{s}} \mathbf{0}$

(One-WayOut) $\bar{\omega}@z(\vec{x}) \xrightarrow{\bar{\omega}@l/z(\vec{v}/\vec{x})} \mathbf{0}$

(One-WayOutLoc) $\bar{\omega}@l(\vec{x}) \xrightarrow{\bar{\omega}@l(\vec{v}/\vec{x})} \mathbf{0}$

(One-WayIn) $\omega(\vec{x}) \xrightarrow{l:\omega(\vec{v}/\vec{x})} \mathbf{0}$

(Assign) $x := e \xrightarrow{x:=v/e} \mathbf{0}$

(Req-Out) $\overline{o_r}@z(\vec{x},\vec{y}) \xrightarrow{l':\overline{o_r}@l/z(\vec{v}/\vec{x},\vec{y})} o_r(\vec{y})$

(Req-OutLoc) $\overline{o_r}@l(\vec{x},\vec{y}) \xrightarrow{l':\overline{o_r}@l(\vec{v}/\vec{x},\vec{y})} o_r(\vec{y})$

(Req-In) $o_r(\vec{x},\vec{y},P) \xrightarrow{l':o_r\,(\vec{v}/\vec{x},\vec{y},P)@l} P; \bar{o}_r@l(\vec{y})$

(If then) $\chi?P : Q \xrightarrow{\chi?} P$

(Else) $\chi?P : Q \xrightarrow{\neg\chi?} Q$

(Iteration) $\chi \rightleftharpoons P \xrightarrow{\chi?} P; \chi \rightleftharpoons P$

(Not Iteration) $\chi \rightleftharpoons P \xrightarrow{\neg\chi?} \mathbf{0}$

(Synchro)
$$\frac{P \xrightarrow{s} P',\ Q \xrightarrow{\bar{s}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

(Sequence)
$$\frac{P \xrightarrow{a} P'}{P;Q \xrightarrow{a} P';Q}$$

(Parallel)
$$\frac{P \xrightarrow{a} P'}{P \mid Q \xrightarrow{a} P' \mid Q}$$

(Choice)
$$\frac{\epsilon_i \xrightarrow{a} \mathbf{0} \quad i \in I}{\sum_{i\in I}^{+} \epsilon_i; P_i \xrightarrow{a} P_i}$$

Structural Congruence

$$P \mid Q \equiv Q \mid P \qquad P \mid \mathbf{0} \equiv \mathbf{0} \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R \qquad \mathbf{0}; P \equiv P$$

Table 3.2: Rules for service behaviour lts layer

3.1.4 Example

In the following we present a simple example which describes the service behaviour ($\mathsf{P}_{\mathsf{REG}}$) of a register service.

$\mathsf{P}_{\mathsf{REG}} := getData(id, serData, serData := \mathtt{read(}id\mathtt{)})$
$+getIdByQuery(query, id, id := \mathtt{search(}query\mathtt{)})$

The service supplies two Request-Response operations on which it is possible to retrieve some kind of data. In particular, $geData$ receives an id and return the data contained within the variable $SerData$ whose calculation is modelled by means of a function $\mathtt{read(}id\mathtt{)}$ whereas $getIdByQuery$ receives a query (here we abstract away from the query formalization) and it returns an id which depends on the query. The function $\mathtt{search(}query\mathtt{)}$ abstractly models some kind of search for retrieving the right id.

3.1.5 Service and starting application

As we will see in the following, a services system is always obtained as a composition of service engines which execute service behaviours. It is worth noting that each service engine always starts its actitivities by means of an exetrenal input operation, in other words the execution of each service engine is always enabled by an external invocation. From such a consideration, it trivially descends that also the system, obtained as a composition of service engines, will start its activities by receiving a message from an external application. Here, we distinguish between a service behaviour which describes a service and a service behaviours which describes a *starting application*. The former will always starts with an external input operation whereas the latter starts with all the processes except the input operations. In the following, we will refer to the service behaviour of a service by exploiting the terms *service behaviour* whereas we will use the term *starting application service behaviour* for denoting that which describes a starting application. The well-formedness definition for the service behaviour follows.

Definition 3.1 ***(Service behaviour well-formedness)*** *Let $P \in SC$ be a service behaviour calculus process. Let Ψ the set of all the possible external input operation terms. We say that P is a well-formed service behaviour process if:*

$$\exists a_1, ..., a_n \in \Psi, \exists Q_1, ..., Q_n \in SC, P = a_1; Q_1 + ... + a_n; Q_n$$

We denote the set of all the well-formed service behaviour processes with the symbol X_{SC}.

The condition above states that a service behaviour process is well-formed if it is formed by a set of processes, composed by means of an alternative choice, which start with an input operation. By defintion it follows that $X_{SC} \subseteq SC$. In the following we define a session of a service behaviour as a trace which ends in a null process.

Definition 3.2 *Let $P \in X_{SC}$, we say that the trace a^* of P is a session iff:*

$$P \xrightarrow{a^*} \boldsymbol{0}$$

In the following we present the well-formedness definition for a starting application.

Definition 3.3 ***(Starting application service behaviour well-formedness)*** *Let $P \in SC$ be a service behaviour calculus process and let A be the set of all its traces. We say that P is a well-formed starting application service behaviour process if for each trace of A the first external operation occurence is always an output external operation. We denote the set of all the well-formed starting application service behaviour processes with the symbol X_{STA}.*

The condition above states that, differently from a service, the first external communication for a starting application is always an output towards another service.

3.2 Service engine calculus

This section is devoted to present the service engine calculus. Before presenting its syntax, we introduce some basic concepts such as *state*, *correlation sets* and *service declaration*. In a service engine indeed, all the executed sessions of a service behaviour are joined by a state and a correlation set. Furthermore, a service engine always executes sessions by following the specifications defined within the service declaration.

3.2.1 State

A state is represented by a function $\mathcal{S} : Var \rightarrow Val \cup \{\bot\}$ from variables to the set $Val \cup \{\bot\}$ ranged over by w. Val, ranged over by v, is a generic set of values on which it is defined a total order relation[3]. $\mathcal{S}(x)$ represents the value of variable x in the state $\mathcal{S}$ ($\mathcal{S}(x) = \bot$ means that x is undefined) while $\mathcal{S}[v/x]$ denotes the state $\mathcal{S}$ where x holds value v (we use $\mathcal{S}[\vec{v}/\vec{x}]$ when dealing with tuples of variables), formally:

$$\mathcal{S}[v/x] = \mathcal{S}' \quad \mathcal{S}'(x') = \begin{cases} v & if\ x' = x \\ \mathcal{S}(x') & otherwise \end{cases}$$

Conditions can be evaluated over states. We exploit the notation $\mathcal{S} \vdash \chi$ for denoting that the state $\mathcal{S}$ satisfies the condition χ. The satisfaction relation for $\vdash$ is defined by the following rules, where e denotes an expression and $e \hookrightarrow_{\mathcal{S}} v$ denotes that, when the state is $\mathcal{S}$, the expression e is evaluated into the value v or, when some variables within the expression are not instantiated, into the symbol $\bot$:

1. $e \hookrightarrow_{\mathcal{S}} v, \mathcal{S}(x) \le v \Rightarrow \mathcal{S} \vdash x \le e$
2. $e \hookrightarrow_{\mathcal{S}} v, v \le \mathcal{S}(x) \Rightarrow \mathcal{S} \vdash e \le x$
3. $\mathcal{S} \vdash \chi' \wedge \mathcal{S} \vdash \chi'' \Rightarrow \mathcal{S} \vdash \chi' \wedge \chi''$
4. $\neg(\mathcal{S} \vdash \chi) \Rightarrow \mathcal{S} \vdash \neg\chi$

3.2.2 Correlation sets

Sessions often require to be distinguished and accessed only by those dialoguers which hold some specific references. In the object oriented paradigm such a reference is the object reference guaranteed by the object oriented framework. In service oriented computing in general, we cannot assume the existence of an underlying framework which guarantees references management. Correlation sets allows us to address such an issue by introducing a specific programming construct. Given a service engine, it is possible to

[3]We extend such an order relation on the set $Val \cup \{\bot\}$ considering $\bot < v,\ \forall v \in Val$.

define a *correlation set* which allows for the identification of the different sessions depending on the incoming message values. In particular, a correlation set is a set of variables called *correlated variables*. Formally, let $CSet = \mathbf{P}(Var)$ be the set of all the correlation sets ranged over by c. In a service engine a session is identified by the values assigned to the correlated variables within the current state. In other words, a session is identified by its own state and a correlation set c implicitly identifies the service engine states by means of the correlated variable store contents. Thus, all the service engine states that share the same store contents for the correlated variables, defined within a correlation set c, can be considered equal from the point of view of the correlation on c. For example, if we consider a correlation set c defined as $c = \{x\}$ and two possible service engine states $\mathcal{S}[1/x, 6/y]$ and $\mathcal{S}'[1/x, 8/y]$ where only the variables x and y are initialized, we can say that $\mathcal{S}$ and $\mathcal{S}'$ are equal from the point of view of the correlation on c because they share the same value for the store content of the correlated variable x. Moreover we can say that $\mathcal{S}$ and $\mathcal{S}'$ are different states from the point of view of the correlation on the correlation set c' defined as $c' = \{y\}$. Moreover, the session identification issue is raised when a message is received on an input operation, that is correlation set mechanism is always triggered by a One-Way input action or a Request-Response input one at the level of service behaviour. Since there should be several sessions that are waiting on the same input operation, the right session to which the message is delivered is only identified by means of the correlated variables values involved within the input action. This fact implies that there could be input operations that receive values only for variables that represent a subset of the correlation set. In this case, it is not possible to distinguish the session states by considering the store contents of all the variables contained within the correlation set but it is necessary to consider only those correlated variables that are specified within the One-Way operation or the Request-Response one. Let us consider, for example, a service engine where the following two sessions are waiting for a message on the One-Way operation a and each session is joined with a different state:

$i)\quad a(\langle x, y\rangle), \mathcal{S}[5/x, 2/y, 1/z]$

$ii)\quad a(\langle x, y\rangle), \mathcal{S}[6/x, 2/y, 0/z]$

The One-Way operation a is waiting for two values that will be stored within the variables x and y respectively. Session i) is joined with a state where variable x is initialized with value 5, variable y with value 2 and variable z with value 1, whereas session ii) is joined with a state where the variable x is initialized with value 6, variable y with value 2 and variable z with value 0. Now, let us consider the following different cases:

a) a correlation set $c = \{x\}$ is defined for the service engine

b) a correlation set $c' = \{y\}$ is defined for the service engine

c) a correlation set $c'' = \{y, z\}$ is defined for the service engine

In case a) it is possible to distinguish the two sessions because they have a different store content for the variable x which is contained within both the correlation set and the receiving variables of the One-Way operation a. In case b) it is not possible to distinguish the two sessions because they are identified by the store content of the variable y which is equal to 2 in both states. Since variable x is not contained within the correlation set it is not relevant to the end of correlation. In case c), although it is possible to distinguish the two sessions by considering the correlation set c'' because it contains the correlated variables y and z where the latter one has different values within the two states, sessions i) and ii) cannot be distinguished when a message is received on the operation a. This is due to the fact that only the correlated variable y is specified within the receiving variables of the operation a and only its store contents can be exploited for distinguishing session states. Since y has a value equal to 2 in both cases, it does not allow for the identification of the session. In the following we present a formal judgement which allows us to state if an incoming message value is correlated to a variable content when a given correlation set is specified. Given a variable x, two values v and w and a correlation set c, where x is the receiving variable name included within a not specified input operation, v is the received value that will update the content of x, w is the store content of x before receiving the value v and c is the correlation set to be considered, we say that v *is correlated to* x *, whose actual state value is* w*, on the correlation set* c, $v/x \vdash_c w$, if:

a) the variable x belongs to c and its actual value is $w = v$

b) the variable x belongs to c and $w = \perp$

c) the variable x does not belong to c

Condition a) states that only a variable which is contained within the correlation set can be considered to the end of correlation. Correlation takes place only if the store content of the correlated variable corresponds to the incoming value. Condition b) states that correlated variables which are not initialised, are not relevant to the end of correlation. Condition c) states that all the variables that do not belong to the correlation set are not relevant to the end of correlation. Formally we exploit the following notation:

$$v/x \vdash_c w \Longleftrightarrow (x \in c \wedge (v = w \vee w = \perp)) \vee x \notin c$$

We extend such a definition to vector of variables and values:

$$\vec{v}/\vec{x} \vdash_c \vec{w} \Longleftrightarrow \forall x_i, v_i/x_i \vdash_c w_i$$

3.2.2.1 Example

In order to clarify the correlation set mechanism let us consider the following simple example where we consider a system with three services involved: a Flight Reservation Service (FRS) which offers the possibility to book flights and Alice and Bob that represent two different service clients that invoke the Flight Reservation Service. Here, we assume that both the clients have already initiated some sessions on the FRS service by sending their personal account IDs (1234 for Alice and 5678 for Bob). In particular, we assume that Alice has started two sessions by requesting for two different flight numbers and Bob has started a session by requesting for a flight number. Here, we consider the case where the FRS service is waiting for a flight confirmation from both the clients by exploiting the same input operation within all the sessions. In the following, we present the formalization of the sessions that are executed within the FRS service engine[4].

$$
\begin{aligned}
\{\mathsf{ID}, \mathsf{fNum}\} \triangleright (&\\
&(\mathsf{confirm}(\langle \mathsf{ID}, \mathsf{fNum}, \mathsf{resp}\rangle); \texttt{booking}(), \mathcal{S}[1234/\mathsf{ID}, \mathsf{H543}/\mathsf{fNum}])\\
&\mid\\
&(\mathsf{confirm}(\langle \mathsf{ID}, \mathsf{fNum}, \mathsf{resp}\rangle); \texttt{booking}(), \mathcal{S}[1234/\mathsf{ID}, \mathsf{H798}/\mathsf{fNum}])\\
&\mid\\
&(\mathsf{confirm}(\langle \mathsf{ID}, \mathsf{fNum}, \mathsf{resp}\rangle); \texttt{booking}(), \mathcal{S}[5678/\mathsf{ID}, \mathsf{F821}/\mathsf{fNum}])\\
&)
\end{aligned}
$$

where variable ID contains the personal account ID of a client and variable fNum contains the flight number. The $\triangleright$ operator means that the correlation set $\{\mathsf{ID}, \mathsf{fNum}\}$ guards the three concurrent sessions represented by the couples:

$$\mathrm{I})(\mathsf{confirm}(\langle \mathsf{ID}, \mathsf{fNum}, \mathsf{resp}\rangle); \texttt{booking}(), \mathcal{S}[1234/\mathsf{ID}, \mathsf{H543}/\mathsf{fNum}])$$

$$\mathrm{II})(\mathsf{confirm}(\langle \mathsf{ID}, \mathsf{fNum}, \mathsf{resp}\rangle); \texttt{booking}(), \mathcal{S}[1234/\mathsf{ID}, \mathsf{H798}/\mathsf{fNum}])$$

$$\mathrm{III})(\mathsf{confirm}(\langle \mathsf{ID}, \mathsf{fNum}, \mathsf{resp}\rangle); \texttt{booking}(), \mathcal{S}[5678/\mathsf{ID}, \mathsf{F821}/\mathsf{fNum}])$$

$\mathsf{confirm}$ is the name of the One-Way operation on which the sessions are waiting for a confirmation from the clients. The operation $\mathsf{confirm}$ receives three values where the former one represents the ID which denotes the client, the second one represents the flight number and the third one represents the actual confirmation response. The function $\texttt{booking}()$ models all the computational steps necessary for booking a flight. It is worth noting that in session I) the state is initialised with the variable ID set on the value 1234, which corresponds to the Alice personal account, and the variable fNum is set on the value H543 which, we assume, is the first flight number selected by Alice. In session II) variable ID is set on the personal account number of Alice whereas variable fNum is set on the value H798, which corresponds to the second flight number selected by Alice. Analogously, in session III) the state is initialised with variable ID set on the value 5678

[4]It is worth noting that, here, we exploit in advance a subpart of the syntax of the service engine calculus which will be explained in the next section.

which corresponds to Bob and variable fNum set on the value $\mathsf{F821}$ which, we assume, is the number of the flight selected by Bob. Now, let us consider the case that Alice invokes the One-Way $\mathsf{confirm}$ by performing a Notification $\overline{\mathsf{confirm}}(\langle 1234, \mathsf{H798}, \mathsf{yes}\rangle)$. In this case the message is correctly delivered to session II) because the store contents of the correlated variables ID and fNum correspond to the incoming values. In the following we show the formal correlation judgements for the three sessions:

$$\mathrm{i})\ \langle 1234, \mathsf{H798}, \mathsf{yes}\rangle \,/\, \langle \mathsf{ID}, \mathsf{fNum}, \mathsf{resp}\rangle \nvdash_{\{\mathrm{ID},\mathrm{fNum}\}} \langle 1234, \mathsf{H543}, \perp\rangle$$

$$\mathrm{ii})\ \langle 1234, \mathsf{H798}, \mathsf{yes}\rangle \,/\, \langle \mathsf{ID}, \mathsf{fNum}, \mathsf{resp}\rangle \vdash_{\{\mathrm{ID},\mathrm{fNum}\}} \langle 1234, \mathsf{H798}, \perp\rangle$$

$$\mathrm{iii})\ \langle 1234, \mathsf{H798}, \mathsf{yes}\rangle \,/\, \langle \mathsf{ID}, \mathsf{fNum}, \mathsf{resp}\rangle \nvdash_{\{\mathrm{ID},\mathrm{fNum}\}} \langle 5678, \mathsf{F821}, \perp\rangle$$

3.2.3 Service declaration

A service engine supplies a support for executing service behaviour sessions and it is composed of a service declaration and an execution environment. The service declaration contains all the necessary information for executing sessions. In particular, the service declaration specifies:

- the service behaviour whose sessions are executed by the service engine.
- if each session has its own state or if there is a common state shared by all the sessions. In the former case the state is renewed each time the execution of a session starts and it expires when the session terminates: we say that the state is *not persistent*. In the latter case, the state is never renewed and the variables hold their values after the termination of the sessions: we say that the state is *persistent*.
- the correlation set which guards the executed sessions
- if the sessions are executed in a sequential order or in a concurrent one (execution modality).
- the location where the service engine is deployed.

The syntax follows:

$U ::= P_\times \mid P_\bullet$
$W ::= c \triangleright U$
$L ::= !W \mid W^*$
$D ::= L@l$

where $P \in X_{SC}$ is a service behaviour, flag $\times$ denotes that P is equipped with a not persistent state and flag $\bullet$ denotes that P is equipped with a persistent one. c is the correlation set which guards the execution of the sessions, $!W$ denotes a concurrent execution of the sessions and W^* denotes the fact that sessions are executed in a sequential order. D is a service declaration and l is the location where the service engine is deployed.

3.2.4 The service engine calculus syntax

Here we present the service engine calculus syntax which exploits the service behaviour calculus and the service declaration syntax. The syntax is obtained as a composition of four different syntactic elements: the *service behaviour state coupled* element, the *execution environment* element, the *executing service engine* element and the *service engine* one. Each syntactic element represents a set of processes characterized by a specific feature of the service engine. In particular, the former deals with processes composed by a service behaviour and a state whereas the second one deals with the execution of sessions guarded by a correlation set, the executing service engine element deals with the session execution modalities (concurrent or sequential) and, finally, the service engine element allows us to define a located service engine by exploiting the previous elements. The syntax of the service behaviour coupled state element follows:

$P_S ::= (P, \mathcal{S})$

P_S represents a service behaviour session actually executed on a service engine ($P \in X_{SC}$) coupled with a state ($\mathcal{S}$). We denote with Y_S the set of all the service behaviour state coupled processes, ranged over by P_S. The syntax of the execution environment element follows:

$H ::= c \triangleright P_{PS}$
$P_{PS} ::= P_S \mid \ \ P_S \mid P_S$

where P_{PS} can be a service behaviour coupled with a state or the parallel composition of them. H is the execution environment which is a composition of service behaviour state coupled processes guarded by a correlation set c and it represents the actual sessions which are running on the service engine. It is worth noting that a service engine is not correlated when $c = \emptyset$. We denote with Y_H the set of all the execution environment processes ranged over by H. The syntax of the executing service engine follows:

$M ::= D[H]$

M is a not located service engine and it is composed of a service declaration D and an execution environment H. It represents an executing service engine without considering any specific location. We denote with Y_M the set of all the executing service engine processes ranged over by M. The syntax of the service engine element follows:

$Y ::= M_l$

where l is the actual location where the service engine is deployed. We denote with Y_Y, ranged over by Y, the set of all the service engine processes.

3.2.4.1 Starting application syntax

Since a starting application never starts with an external input, we do not consider correlation set and execution modality when it is deployed within a service engine. Correlation set and execution modality indeed, are service engine features directly related to

(In)
$$\frac{P \xrightarrow{s} P'}{(P, \mathcal{S}) \xrightarrow{s}_S (P', \mathcal{S})}$$

(Out)
$$\frac{P \xrightarrow{\bar{s}} P'}{(P, \mathcal{S}) \xrightarrow{\bar{s}}_S (P', \mathcal{S})}$$

(Synchro)
$$\frac{P \xrightarrow{\tau} P'}{(P, \mathcal{S}) \xrightarrow{\tau}_S (P', \mathcal{S})}$$

(One-WayOut)
$$\frac{P \xrightarrow{\bar{\omega}@l/z(\vec{v}/\vec{x})} P', \mathcal{S}(z) = l, \mathcal{S}(\vec{x}) = \vec{v}}{(P, \mathcal{S}) \xrightarrow{\bar{\omega}@l(\vec{v})}_S (P', \mathcal{S})}$$

(One-WayOutLoc)
$$\frac{P \xrightarrow{\bar{\omega}@l(\vec{v}/\vec{x})} P', \mathcal{S}(\vec{x}) = \vec{v}}{(P, \mathcal{S}) \xrightarrow{\bar{\omega}@l(\vec{v})}_S (P', \mathcal{S})}$$

(One-WayIn)
$$\frac{P \xrightarrow{l:\omega(\vec{v}/\vec{x})} P'}{(P, \mathcal{S}) \xrightarrow{l:\omega(\vec{v}/\vec{x}) \mapsto \mathcal{S}(\vec{x})}_S (P', \mathcal{S}[\vec{v}/\vec{x}])}$$

(Req-In)
$$\frac{P \xrightarrow{l':o_r\,(\vec{v}/\vec{x}, \vec{y}, P)@l} P'}{(P, \mathcal{S}) \xrightarrow{l':o_r\,(\vec{v}/\vec{x}, \vec{y}, P)@l \mapsto \mathcal{S}(\vec{x})}_S (P', \mathcal{S}[\vec{v}/\vec{x}])}$$

(Req-Out)
$$\frac{P \xrightarrow{l':\overline{o_r}@l/z(\vec{v}/\vec{x}, \vec{y})} P', \mathcal{S}(z) = l, \mathcal{S}(\vec{x}) = \vec{v}}{(P, \mathcal{S}) \xrightarrow{l':\overline{o_r}@l(\vec{v}, \vec{y})}_S (P', \mathcal{S})}$$

(Req-OutLoc)
$$\frac{P \xrightarrow{l':\overline{o_r}@l(\vec{v}/\vec{x}, \vec{y})} P', \mathcal{S}(\vec{x}) = \vec{v}}{(P, \mathcal{S}) \xrightarrow{l':\overline{o_r}@l(\vec{v}, \vec{y})}_S (P', \mathcal{S})}$$

(Assign)
$$\frac{P \xrightarrow{x:=v/e} P', e \hookrightarrow_{\mathcal{S}} v}{(P, \mathcal{S}) \xrightarrow{\tau}_S (P', \mathcal{S}[v/x])}$$

(Satisfaction)
$$\frac{P \xrightarrow{\chi?} P', \chi \vdash \mathcal{S}}{(P, \mathcal{S}) \xrightarrow{\tau}_S (P', \mathcal{S})}$$

(Not Satisfaction)
$$\frac{P \xrightarrow{\neg\chi?} P', \chi \nvdash \mathcal{S}}{(P, \mathcal{S}) \xrightarrow{\tau}_S (P', \mathcal{S})}$$

Table 3.3: Rules for service engine state lts layer

the management of new sessions that can only be initiated by external messages. For this reason, a starting application is always deployed within a service engine equipped with a state only. The syntax which describes a service engine that executes a starting application deals only with the execution environment composed with a couple of a starting application service behaviour and a state:

$Y_{st} ::= [(P, \mathcal{S})]_l$

where $P \in X_{STA}$. Here we do not discuss the semantics for the starting application because it strictly descends from that of the service engine. In the following we will use the term *starting application* for denoting a service engine which executes a starting application service behaviour and the terms *service engine* in the other cases.

3.2.5 Semantics.

The semantics is defined in terms of different label transition system layers presented in Tables 3.3, 3.5, 3.6 and 3.7. Table 3.3 deals with the rules for the service engine state lts layer which defines the semantics for a couple of a service behaviour process and a state, Table 3.5 deals with the rules for service engine correlation lts layer where it is defined the semantics for managing correlation sets, Table 3.6 deals with the service engine execution modality lts layer which defines the semantics for executing sessions in a concurrent or in a sequential way and, finally, Table 3.7 deals with the rules for joining a service engine with a location. Since the semantics is defined in terms of different lts layer where each layer enables the actions for the overlying one, for each layer we define a specific set of actions: Act_S, ranged over by γ, is the set of actions raised by a service behaviour state coupled process, Act_H, ranged over by η, is the set of actions raised by an execution environment process, Act_M, ranged over by ν, is the set of actions raised by an executing service engine process and Act_Y, ranged over by α, is the set of actions raised by a service engine. The action sets are defined as follows:

$$Act_S = \{s, \overline{s}, \tau, \overline{\varpi}@l(\vec{v}), l\!: \omega(\vec{v}/\vec{x}) \mapsto \mathcal{S}(\vec{x}), l'\!: o_r(\vec{v}/\vec{x}, P)@l \mapsto \mathcal{S}(\vec{x}), l'\!: \overline{o_r}@l(\vec{v}, \vec{y})\}$$

$$Act_H = \{s, \overline{s}, \tau, \overline{\varpi}@l(\vec{v}), l{:}\,\omega(\vec{v}), l'{:}o_r(\vec{v}, P)@l, l'{:}\overline{o_r}@l(\vec{v}, \vec{y})\}$$

$$Act_M = \{\tau, \overline{\varpi}@l(\vec{v}), l{:}\,\omega(\vec{v}), l'{:}o_r(\vec{v}, P)@l, l'{:}\overline{o_r}@l(\vec{v}, \vec{y})\}$$

$$Act_Y = Act_M$$

Furthermore, for each lts layer we define a specific relation which allows for the generation of the lts. We define $\rightarrow_S \subseteq Y_S \times Act_S \times Y_S$ as the least relation which satisfies the rules of Table 7.3, $\rightarrow_H \subseteq Y_H \times Act_H \times Y_H$ as the least relation which satisfies the rules of Table 3.5 closed w.r.t. the structural congruence, $\rightarrow_M \subseteq Y_M \times Act_M \times Y_M$ as the least relation which satisfies the rules of Table 3.7 and $\rightarrow_Y \subseteq Y_Y \times Act_Y \times Y_Y$ as the least relation which satisfies the rules of Table 3.7. We formally represent the relationship among the different labelled transition systems by exploiting the following relation:

$$\Delta \subseteq \rightarrow \times \rightarrow_S \times \rightarrow_H \times \rightarrow_M \times \rightarrow_Y$$

where an element $\delta = ((p, a, p'), (p_S, \gamma, p'_S), (h, \eta, h'), (m, \nu, m'), (y, \alpha, y'))$ with $p, p' \in X_{SC}$, $p_S, p'_S \in Y_S$, $h, h' \in Y_H$, $m, m' \in Y_M$ and $y, y' \in Y_Y$, belongs to Δ if the following conditions are satisfied:

- $(p, a, p') \Rightarrow (p_S, \gamma, p'_S)$ w.r.t. the rules of Table 3.3.
- $(p_S, \gamma, p'_S) \Rightarrow (h, \eta, h')$ w.r.t. the rules of Table 3.5
- $(h, \eta, h') \Rightarrow (m, \nu, m')$ w.r.t. the rules of Table 3.6.
- $(m, \nu, m') \Rightarrow (y, \alpha, y')$ w.r.t. the rules of Table 3.7.

The relation Δ implicitly defines a relationship among the actions raised by the different lts layers which is reported In Table 3.4. In general, an action when raised to the overlying lts layer, will be modified in order to forward only the needed information. In particular, as far as Table 3.3 is concerned, an action is enabled if the current state contains variables values which correspond to those reported into the action. An action is disabled, i.e. it is not raised to the overlying lts layer, when the variables values into the state do not correspond to those reported into the action. Indeed, considering Table 3.4, actions a), b) are not altered since they do not deal with the state whereas actions g) are

	Act	Act_S	Act_H	$Act_M = Act_Y$
a)	s	s	s	τ
b)	$\overline{s}$	$\overline{s}$	$\overline{s}$	τ
c)	$\overline{\omega}@l/z(\vec{v}/\vec{x})$ $\overline{\omega}@l(\vec{v}/\vec{x})$	$\overline{\omega}@l(\vec{v})$	$\overline{\omega}@l(\vec{v})$	$\overline{\omega}@l(\vec{v})$
d)	$l\colon \omega(\vec{v}/\vec{x})$	$l\colon \omega(\vec{v}/\vec{x}) \mapsto \mathcal{S}(\vec{x})$	$l\colon \omega(\vec{v})$	$l\colon \omega(\vec{v})$
e)	$l'\colon \overline{o_r}@l/z(\vec{v}/\vec{x}, \vec{y})$ $l'\colon \overline{o_r}@l(\vec{v}/\vec{x}, \vec{y})$	$l'\colon \overline{o_r}@l(\vec{v}, \vec{y})$	$l'\colon \overline{o_r}@l(\vec{v}, \vec{y})$	$l'\colon \overline{o_r}@l(\vec{v}, \vec{y})$
f)	$l'\colon o_r(\vec{v}/\vec{x}, \vec{y}, P)@l$	$l'\colon o_r(\vec{v}/\vec{x}, \vec{y}, P)@l \mapsto \mathcal{S}(\vec{x})$	$l'\colon o_r(\vec{v}, \vec{y}, P)@l$	$l'\colon o_r(\vec{v}, \vec{y}, P)@l$
g)	$\chi?, \neg\chi?, x := v/e, \tau$	τ	τ	τ

Table 3.4: Lts layer action relation

replaced with a τ action because they do not carry any information needed by the overlying layer. Actions c) and e) are related to the output operations and, when enabled, they resolve the values of the variables and locations. Indeed, if we consider rules ONE-WAYOUT and REQ-OUT they contain the conditions $\mathcal{S}(z) = l$ and $\mathcal{S}(\vec{x}) = \vec{v}$ which allows for the verification of the actual values of the variables within the current state. In particular, rules ONE-WAYOUTLOC and REQ-OUTLOC do not resolve locations because they are explicitly represented. Actions d) and f) deal with the input operations (rules ONE-WAYIN, REQ-IN) and, when enabled, they do not resolve the values of the variables but they forward the actual values of the variables involved into the action ($\mapsto \mathcal{S}(\vec{x})$). This is due to the fact that some variables could be correlated and it will be necessary to verify if they satisfy the current correlation set. Such a control will be done in the overlying layer whose rules are reported in Table 3.5 where rules CORRELATEDONE-WAYIN and CORRELATEDREQ-IN deal with the actions d) and f) of Table 3.4. In particular, in rule CORRELATEDONE-WAYIN we assume that P_S has a transition labelled with the action d), that means that a message which contains the values $\vec{v}$ is received on the operation ω trying to update the variables $\vec{x}$ which are set on values $\vec{w}$ within the state. Such a transition can be enabled at the service engine execution modality layer only if the correlation judgement $\vec{v}/\vec{x} \vdash_c \vec{w}$ is satisfied; in other words, the message can be received only if the store contents $\vec{w}$ of the correlated variables specified within $\vec{x}$ correspond to the related incoming values specified within $\vec{v}$. Analogously, rule CORRELATEDREQ-IN models the same behaviour but considering the action f) instead of action d). In rule NOT CORRELATED premises, we assume that P_S performs an action which differs from those treated in rules CORRELATEDONE-WAYIN and CORRELATEDREQ-IN (i.e. input operation actions) and, as a conclusion, we state that all the actions, which do not deal with the input operation ones, can be raised to the overlying layer because they do not deal with the correlation mechanism. Rule PARALLEL allows for the evolving of the parallel composition of P_S processes guarded by a correlation set c. As far as Table 3.6 is concerned, the actions are enabled at the level of the service engine (rule EXECUTION) and session execution modalities, concurrent or sequential with a persistent or a not persistent state, are defined within rules CONCURRENTNOTPERSISTENT, CONCURRENTPER-

SISTENT, SEQUENTIALNOTPERSISTENT and SEQUENTIALPERSISTENT. Rule CONCURRENTNOTPERSISTENT deals with a concurrent execution of the sessions and with a not persistent state. In particular, each session has its own state, that is initially fresh ($\mathcal{S}_\perp$), and it is executed concurrently with the other ones. It is worth noting that condition $\nexists\ \mathcal{S}_i \in P_S\ c \triangleright (P, \mathcal{S}_i) \xrightarrow{\gamma} c \triangleright (P', \mathcal{S}'_i)$[5] states that it is not possible to start a new session with a set of values for the correlated variables that belong to another running session. Rule CONCURRENTPERSISTENT deals with the concurrent execution of sessions which share a common state. Rule SEQUENTIALNOTPERSISTENT deals with the sequential execution of sessions which have their own state. In this case there is always no more than one executed session at a time. The state is not persistent and it is renewed each time a new session is spawned. Rule SEQUENTIALPERSISTENT deals with the sequential execution of the sessions where the state is shared and it does not expire after session termination. It is worth noting that the actions s and $\overline{s}$ are replaced with a τ only at the level of the service engine because internal synchronization are exploited for synchronizing sessions which share the same state. Such an issue is discussed deeply in Section 3.5. Finally, rules of Table 3.7 state that only the actions whose location corresponds to that where the service engine is deployed can be raised.

3.2.6 Example

Here, as a an example of service engine, we consider the register service behaviour presented in the previous section:

$P_{REG} := getData(id, serData, serData := \mathtt{read(}id\mathtt{)})$
$\quad +getIdByQuery(query, id, id := \mathtt{search(}query\mathtt{)})$

$REG := (\emptyset \triangleright P_{REG\bullet})^{*}@REG[\emptyset \triangleright (\mathbf{0}, \mathcal{S}_{REG0})]_{REG}$

In this case, the service engine REG is declared to sequentially execute the service behaviour P_{REG} with a persistent state and without correlated variables. Indeed, it is reasonable to model a service register with a persistent state ($\mathcal{S}_{REG0}$) because it usually manages data

[5]We abuse of the notation $\mathcal{S}_i \in P_S$ for meaning that it exists a couple $(P, \mathcal{S}_i)$ within the term P_S

(NOT CORRELATED)

$$\frac{P_S \xrightarrow{\gamma}_S P'_S}{c \triangleright P_S \xrightarrow{\gamma}_H c \triangleright P'_S} \quad \gamma \neq \begin{cases} l\colon \omega(\vec{v}/\vec{x}) \mapsto (\vec{w}) \\ l'\colon o_r(\vec{v}/\vec{x}, \vec{y}, P)@l \mapsto \vec{w} \end{cases}$$

(PARALLEL)

$$\frac{c \triangleright P_S \xrightarrow{\gamma}_H c \triangleright P'_S}{c \triangleright P_S \mid Q_S \xrightarrow{\gamma}_H c \triangleright P'_S \mid Q_S}$$

(CORRELATEDONE-WAYIN)

$$\frac{P_S \xrightarrow{l:\omega(\vec{v}/\vec{x})\mapsto(\vec{w})}_S P'_S, \vec{v}/\vec{x} \vdash_c \vec{w}}{c \triangleright P_S \xrightarrow{l:\omega(\vec{v})}_H c \triangleright P'_S}$$

(CORRELATEDREQ-IN)

$$\frac{P_S \xrightarrow{l':o_r(\vec{v}/\vec{x},\vec{y},P)@l\mapsto(\vec{w})}_S P'_S, \vec{v}/\vec{x} \vdash_c \vec{w}}{c \triangleright P_S \xrightarrow{l':o_r(\vec{v},\vec{y})@l}_H c \triangleright P'_S}$$

STRUCTURAL CONGRUENCE

$$c \triangleright P_S \mid Q_S \equiv c \triangleright Q_S \mid P_S \qquad c \triangleright P_S \mid (Q_S \mid R_S) \equiv c \triangleright (P_S \mid Q_S) \mid R_S$$
$$c \triangleright P_S \mid (\mathbf{0}, \mathcal{S}) \equiv c \triangleright P_S$$

Table 3.5: Rules for service engine correlation lts layer

that do not depend on the single executed session but they must be remain available for all the possible sessions. Furthermore, we do not need correlated variables because the service behaviour only provides two Request-Response operations without any other message exchange.

(CONCURRENTNOTPERSISTENT)
$$\frac{c \triangleright (P, \mathcal{S}_\perp) \xrightarrow{\eta}_H c \triangleright (P', \mathcal{S}'), \nexists\, \mathcal{S}_i \in P_S\ c \triangleright (P, \mathcal{S}_i) \xrightarrow{\eta}_H c \triangleright (P', \mathcal{S}_i')}{!(c \triangleright P_\times)@l[c \triangleright P_S] \xrightarrow{\eta}_M\ !(c \triangleright P_\times)@l[c \triangleright P_S \mid (P', \mathcal{S}')]}$$

(CONCURRENTPERSISTENT)
$$\frac{c \triangleright (P, \mathcal{S}) \xrightarrow{\eta}_H c \triangleright (P', \mathcal{S}')}{!(c \triangleright P_\bullet)@l[c \triangleright (Q, \mathcal{S})] \xrightarrow{\eta}_M\ !(c \triangleright P_\bullet)@l[c \triangleright (Q \mid P', \mathcal{S}')]}$$

(SEQUENTIALNOTPERSISTENT)
$$\frac{c \triangleright (P, \mathcal{S}_\perp) \xrightarrow{\eta}_H c \triangleright (P', \mathcal{S}')}{(c \triangleright P_\times)^*@l[c \triangleright (\mathbf{0}, \mathcal{S}'')] \xrightarrow{\eta}_M (c \triangleright P_\times)^*@l[c \triangleright (P', \mathcal{S}')]}$$

(SEQUENTIALPERSISTENT)
$$\frac{c \triangleright (P, \mathcal{S}) \xrightarrow{\eta}_H c \triangleright (P', \mathcal{S}')}{(c \triangleright P_\bullet)^*@l[c \triangleright (\mathbf{0}, \mathcal{S})] \xrightarrow{\eta}_M (c \triangleright P_\bullet)^*@l[c \triangleright (P', \mathcal{S}')]}$$

(EXECUTION)
$$\frac{H \xrightarrow{\eta}_H H'}{D[H] \xrightarrow{\nu}_M D[H']} \quad \begin{cases} \eta = s, \bar{s}, \tau \Rightarrow \nu = \tau \\ \eta \neq s, \bar{s}, \tau \Rightarrow \nu = \eta \end{cases}$$

Table 3.6: Rules for service engine execution modality lts layer

(LOCATED ONEWAY-IN)
$$\frac{M \overset{l:\omega(\vec{v})}{\rightarrow}_M M'}{M_l \overset{l:\omega(\vec{v})}{\rightarrow}_Y M'_l}$$

(LOCATED REQ-IN)
$$\frac{M \overset{l':o_r(\vec{v},\vec{y},P)@l}{\rightarrow}_M M'}{M_{l'} \overset{l':o_r(\vec{v},\vec{y},P)@l}{\rightarrow}_Y M'_{l'}}$$

(LOCATED REQ-OUT)
$$\frac{M \overset{l':\overline{o_r}@l(\vec{v},\vec{y})}{\rightarrow}_M M'}{M_{l'} \overset{l':\overline{o_r}@l(\vec{v},\vec{y})}{\rightarrow}_Y M'_{l'}}$$

(NOT LOCATED ACTIONS)
$$\frac{M \overset{\gamma}{\rightarrow}_M M'}{M_{l'} \overset{\gamma}{\rightarrow}_Y M'_{l'}} \quad \gamma \neq \begin{cases} l : \omega(\vec{v}) \\ l' : o_r(\vec{v},\vec{y},P)@l \\ l' : \overline{o_r}@l(\vec{v},\vec{y}) \end{cases}$$

Table 3.7: Rules for service engine location lts layer

3.3 Services system calculus

Here we present the services system calculus which is based on the service engine one and it allows for the composition of different engines into a system. The calculus syntax follows:

$E ::= Y \mid E \parallel E$

A service engine system E can be a service engine Y or a parallel composition of them. We define X_E the set of all the possible services systems ranged over by E.

Semantics. Let $Act_E = Act_Y$ be the set of services system actions, ranged over by ρ. The semantics is defined in terms of a labelled transition system built on the relation $\rightarrow_E \subseteq X_E \times Act_E \times X_E$ whose rules are described in Table 3.8 and closed w.r.t. the structural congruence. In particular, rules ONE-WAYSYNC and REQ-SYNC describe synchronizations among different service engines. The former models a One-Way message exchange and the latter models the request message exchange in the case of a Request-Response. It is worth noting, that the response message exchange, in the case of a Request-Response, is modelled by the ruleONE-WAYSYNC indeed, by means of rules of Table 3.2, Request-

Response operations can be externally seen as two One-Ways.

(One-WaySync)
$$\frac{Y \xrightarrow{\bar{\omega}@l(\vec{v})}_E Y' \;,\; Z \xrightarrow{l:\omega(\vec{v})}_E Z'}{Y \parallel Z \xrightarrow{\tau}_E Y' \parallel Z'}$$

(Req-Sync)
$$\frac{Y \xrightarrow{l:\overline{o_r}@l'(\vec{v},\vec{y})}_E Y' \;,\; Z \xrightarrow{l':o_r(\vec{v},\vec{z})@l}_E Z'}{Y \parallel Z \xrightarrow{\tau}_E Y' \parallel Z'}$$

(Par-Ext)
$$\frac{E_1 \xrightarrow{\alpha}_E E_1'}{E_1 \parallel E_2 \xrightarrow{\alpha}_E E_1' \parallel E_2}$$

(EngineAct)
$$\frac{Y \xrightarrow{\gamma}_Y Y'}{Y \xrightarrow{\gamma}_E Y'}$$

(Structural Congruence over E)

$$E_1 \parallel E_2 \equiv E_2 \parallel E_1 \qquad E_1 \parallel (E_2 \parallel E_3) \equiv (E_1 \parallel E_2) \parallel E_3$$

Table 3.8: Rules for services system lts layer

3.4 Example

In this section we want to model a typical business scenario by exploiting the calculi discussed in the previous section. Firstly, we give an informal description of the business activity that abstracts away from some details and then we present a formal representation of it.

3.4.1 Informal specification.

There are five participants involved:

- a bank
- a supplier
- a market

- a service register
- a customer

The customer wants to buy a given product and it asks for its price to the market. The market queries the register in order to obtain a supplier which is able to satisfy the customer and then it requests the supplier for the price. The market forwards the price to the customer that decides to buy or not. If it decides to buy, the market requests for the order to the supplier and, concurrently, it asks to the bank to perform the financial transaction. In order to do that, the bank will request both the customer and the supplier for the bank data. At the end, the bank will notify the customer, the supplier and the market for the transaction termination.

3.4.2 Formal specification.

The business activity is represented by means of a services system where, for the sake of brevity, we abstract away from some details. We model the bank service and the supplier service by means of two service engines where the former models the necessity to have a persistent memory for dealing with all the data that must be always available and the latter models the necessity to have a session for each invoker that requests for a service. The services system is composed by the following service engines:

- the *Register Service*[6]
- the Bank Service composed by the *Bank Information Service* and the *Bank Master Service*
- the Supplier Service composed by the *Supplier Information Service* and the *Supplier Master Service*
- the *Market Service*
- the *Customer* (it is the starting application)

[6]For the sake of this example we call a service engine with the term service

For the sake of clarity, in the following we report the temporal evolution of the system:

1. the customer requests for a product price to the Market Service by sending also its personal data (modelled with the variable $pdata$ that we assume it represents the name, the surname, the date and the place of birth).

2. the Market Service queries the Register Service in order to obtain a supplier service for fulfilling the customer request.

3. after it has received the supplier location, the Market Service invokes the Supplier Information Service in order to ask for the product price.

4. the Market Service replies to the customer with the price

5. depending on the price the customer decides to buy. If it decides to not to buy the business activity terminates. On the contrary, if it decides to buy, it confirms the order to the market service.

6. when the Market Service has received the order confirmation it performs the following actions in a concurrent way:

   (a) it invokes the Supplier Master Service for starting the purchase order

   (b) it invokes the Bank Master Service for starting a financial transaction

7. the Bank Master Service requests both the customer and the supplier for their own bank data

8. the Bank Master Service performs a financial transaction by invoking the Bank Information Service

9. the Bank Master Service sends a commit to the customer, the supplier and the market

10. the Market Service sends a commit message to the customer

Before presenting the code for each service we report in Fig. 3.1 a sort of roadmap which can be useful for the reader in order to understand the relationships among the service operations. An operation exhibited by a service is represented by a rectangle and an operation invocation is represented by an arrow. Black ractangles represent Request-Response operations whereas white rectangles represent the One-Way ones. In the following we abstract away from some internal computation function and we represent them by exploiting the typewriter font.

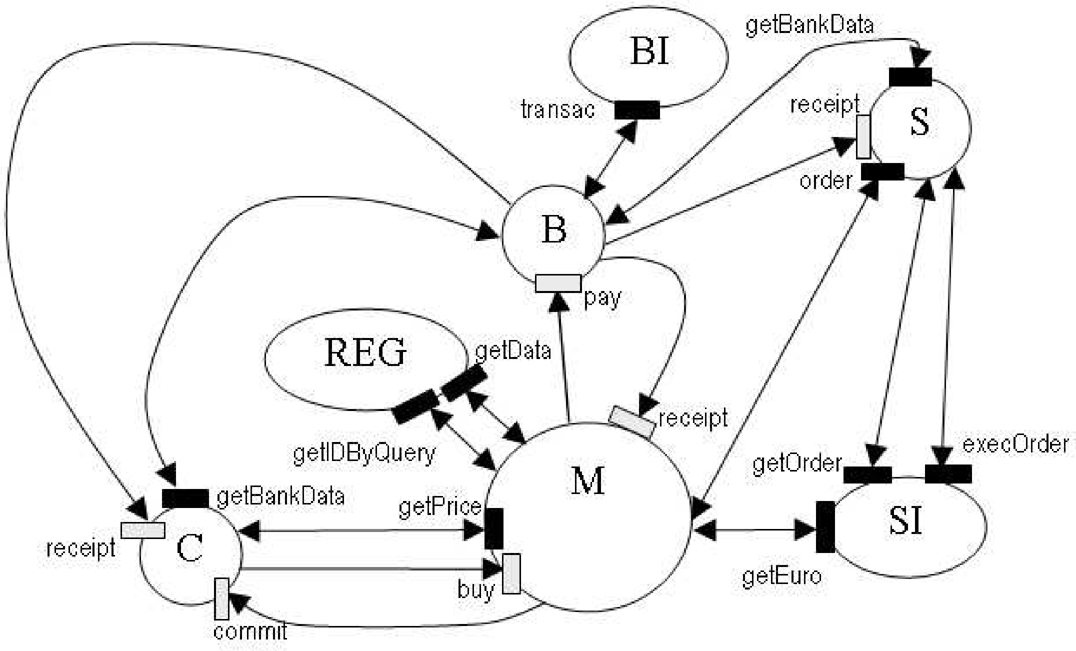

Figure 3.1: Operation relationships among the services of the example

Register Service: The Register Service stores a sort of service database where a unique id is joined to a set of general service information (for example the service location and the service semantics[7]) which are modelled by means of the variable serData. In the following we present the service engine code where $\mathcal{S}_{\mathrm{REG}0}$ is the initial state of the register where we assume it contains a not specified number of registered services.

[7]the modelling of the service semantics is out of the scope of this paper.

$P_{REG} := getData(id, serData, serData := \texttt{read}(id))$
$+getIdByQuery(query, id, id := \texttt{search}(query))$
$REG := (\emptyset \triangleright P_{REG\bullet})^{*}@REG[\emptyset \triangleright (\mathbf{0}, \mathcal{S}_{REG0})]_{REG}$

- The Request-Response operation $getData$ allows us to obtain the data joined with a specific id. The function `read` returns the service data joined with an id.
- The Request-Response operation $getIdByQuery$ models the possibility to receive a service id by querying the register about the general data. The function `search` allows us to model an id search based on a query. For teh sake of brevity we assume that the query must be specified by exploiting some kind of language.

Bank Service: It is composed of two services: The Bank Information Service (BI) and the Bank Master Service (B). The former processes the transactions and it is modelled with a sequential and persistent service engine whereas the latter manages transactions between two participants. It is modelled with a concurrent and not persistent service engine. It is worth noting that the Bank Master Service distinguishes sessions by means of the correlation set $\{data1, data2, idorder\}$ where $data1$ and $data2$ model all the data required for performing a financial transaction between the two particpants and $idorder$ is the supplier purchase order id. The code follows:

$P_{BI} := transac(\langle data1, data2, idorder, euro\rangle, \langle data1, data2, idorder, idtran\rangle,$
$\quad idtran := \texttt{trans}(data1, data2, idorder, euro)$
$)$
$BI := (\emptyset \triangleright P_{BI\bullet})^{*}@BI[\emptyset \triangleright (\mathbf{0}, \mathcal{S}_{BI0})]_{BI}$

$P_B := pay(\langle data1, data2, idorder, loc1, loc2, locI, euro\rangle);$
$\quad \overline{transac}@BI(\langle data1, data2, idorder, euro\rangle, \langle data1, data2, idorder, idtran\rangle);$
$\quad (\overline{receipt}@loc1(\langle idorder, idtran\rangle)$
$\quad \mid \overline{receipt}@loc2(\langle idorder, idtran\rangle)$

$\quad\quad | \ \overline{\mathsf{receipt}}@\mathsf{locI}(\langle \mathsf{idorder}, \mathsf{idtran} \rangle)$

$\quad\quad)$

$\mathsf{B} := !(\{\mathsf{data1}, \mathsf{data2}, \mathsf{idorder}\} \triangleright \mathsf{P}_{\mathsf{B}\times})@\mathsf{B}[\{\mathsf{data1}, \mathsf{data2}, \mathsf{idorder}\} \triangleright \mathbf{0}]_{\mathsf{B}}$

- The Request-Response operation $\mathsf{transac}$ receives the request for processing a transaction denoted by the data contained in $\mathsf{data1}$, $\mathsf{data2}$, $\mathsf{idorder}$ and euro and returns an id (idtran) for the transaction. The function `trans` processes the transaction.

- $\mathcal{S}_{\mathrm{BI0}}$ models the state of the Bank Information Service that we assume it is initiated with some not specified data.

- The One-Way operation pay starts a new session correlated by means of the variables $\mathsf{data1}$, $\mathsf{data2}$ and $\mathsf{idorder}$. The variable euro represents the amount of money to process within the transaction whereas $\mathsf{loc1}$, $\mathsf{loc2}$ and locI represent the location of the first participant, the location of the second participant and the location of the intermediary respectively.

- The Solicit-Response $\overline{\mathsf{transac}}$ allows to invoke the Bank Information Service for processing the transaction.

- The Notification operations $\overline{\mathsf{receipt}}$ sends the receipt to the service located at the received location $\mathsf{loc1}$, $\mathsf{loc2}$ and locI.

Supplier Service: The Supplier Service models a supplier of a certain set of products. The service is built on two services: the Supplier Information Service (SI) and the Supplier Master Service (S). As for the Bank Service, the former stores all the persistent information whereas the latter manages different correlated sessions.

$\mathsf{P}_{\mathsf{SI}} := \mathsf{getEuro}(\mathsf{product}, \mathsf{euro}, \mathsf{euro} := \mathtt{calc(}\mathsf{product}\mathtt{)})$

$+\mathsf{getOrder}(\langle \mathsf{id}, \mathsf{product} \rangle, \mathsf{idorder}, \mathsf{idorder} := \mathtt{calcord(}\mathsf{id}, \mathsf{product}\mathtt{)})$

$+\mathit{execOrder}(\langle \mathit{idorder}, \mathit{idtran}\rangle); \texttt{store}(\mathit{idorder}, \mathit{idtran}); \texttt{ex}(\mathit{idorder})$
$\mathit{SI} := (\emptyset \triangleright P_{SI_\bullet})^{*}@\mathit{SI}[\emptyset \triangleright (\mathbf{0}, \mathcal{S}_{SI0})]_{SI}$

$P_S := \mathit{order}(\langle \mathit{data}, \mathit{product}\rangle, \mathit{idorder}, \overline{\mathit{getOrder}@\mathit{SI}}(\langle \mathit{data}, \mathit{product}\rangle, \mathit{idorder}));$
$\quad \mathit{receipt}(\langle \mathit{idorder}, \mathit{idtran}\rangle);$
$\quad \overline{\mathit{execOrder}@\mathit{SI}}(\langle \mathit{idorder}, \mathit{idtran}\rangle)$
$S := !(\{\mathit{data}, \mathit{product}, \mathit{idorder}\} \triangleright P_{S_\times})@S[\{\mathit{data}, \mathit{product}, \mathit{idorder}\} \triangleright \mathbf{0}]_S$

- The Request-Response operation $\mathit{getEuro}$ returns the price in euros of a given product by means of the function `calc`.
- The Request-Response operation $\mathit{getOrder}$ returns, by means of the function `calcord`, the id of the order to supply.
- The One-Way operation $\mathit{execOrder}$ receives the data of the financial transaction related to a given order (represented by the variable idtran) and then, by means of the functions `store` and `ex`, it stores the received data and it executes the order.
- The Request-Response operation order allows to initiate a new session correlated by the variables data, $\mathit{product}$ and $\mathit{idorder}$ and returns an id order obtained by exploiting the solicit-response operation $\overline{\mathit{getOrder}}$.
- The One-Way operation $\mathit{receipt}$ receives the payment receipt from the Bank Master Service. It is worth noting that on the order Request-Response operation the variables data and $\mathit{product}$ drive the correlation whereas in the $\mathit{receipt}$ operation the variable $\mathit{idorder}$ does it.
- The Notification operation $\overline{\mathit{execOrder}}$ invokes the execution of the order.

Market Service: It is the central service of the system which manages the customer requests and invokes the other services.

```
$P_M := getPrice(\langle pdata, loc, product\rangle, euro,$
        $\overline{getIdByQuery}@REG(product, supId);$
        $\overline{getData}@REG(supId, supData);$
        $supLocEuro := \texttt{extractLoc(}supData, Euro\texttt{);}$
        $supLocOrder := \texttt{extractLoc(}supData, Order\texttt{);}$
        $myLoc := M;$
        $\overline{getEuro}@supLocEuro(product, euro)$
        $);$
    $(\overline{timeout} \mid (timeout$
            $+$
            $buy(\langle pdata, conf\rangle); conf = yes?($
                $\overline{order}@supLocOrder(\langle pdata, product\rangle, idorder) \mid$
                $\overline{pay}@B(\langle pdata, supData, idorder, loc, supLocOrder, myLoc, euro\rangle)$
                $); receipt(\langle idorder, idtran\rangle); \overline{commit}@loc(\emptyset)$
            $))$
$M := !(\{pdata\} \triangleright P_{M\times})@M[\{pdata\} \triangleright \mathbf{0}]_M$
```

- The Request-Response operation getPrice receives a customer request for the price of a product. It receives the variables pdata, which models the personal data of the customes and drives the correlation, and loc which is the location of the customer where the receipt will be sent. Such a location must be explicitly required because it will be used within the Market Service behaviour after the execution of the getPrice operation. Indeed, as we will discuss in Section 3.5, a Request-Response operation implicitly acquires the location of the sender for replying but such a location cannot be used at the level of service behaviour. Before replying to the customer with the product price the Market Service executes the following processes:
  - it queries the Register Service, by means of the Solicit-Responses $\overline{getIdByQuery}$ and $\overline{getData}$, for finding a service which is able to supply the requested product.

  - it extracts the supplier locations for requesting the product price and performing an order, by means of the function `extractLoc` and it stores it within the variables $\mathsf{supLocEuro}$ and $\mathsf{SupLocOrder}$.
  - it invokes the supplier for receiving the product price by means of the Solicit-Response operation $\overline{\mathsf{getEuro}}$.

- When the price is sent to the customer the service starts an internal race between the internal signal $\mathsf{timeout}$ and the One-Way operation buy. Since, for the sake of this thesis, we do not model time passing we exploit the signal $\mathsf{timeout}$ for modelling a sort of timeout alarm after which the session is finished.

- The One-Way operation buy receives the confirmation from the customer for buying the selected product.

- If the customer confirms to buy the product ($\mathsf{conf} = \mathsf{yes}?$), the Market Service concurrently invokes the Supplier Master Service in order to initiate the order and the Bank Master Service for requesting a transaction for the customer.

- At the end, it sends a commit to the customer by means of the Notification operation $\overline{\mathsf{commit}}$.

Customer. The customer is a *starting application*[8]. It invokes the Market Service by exploiting the Solicit-Response $\overline{\mathsf{getPrice}}$ and it receives the price of the product *carrot*. If the price is less than 10 euros it decides to buy by invoking the buy operation of the market service. Then, it waits for the receipt from the Bank Master Service and for a commit from the Market Service.

$$\begin{aligned}\mathsf{C} := {} & [(\overline{\mathsf{getPrice}}@\mathsf{M}(\langle \mathsf{pdata}, \mathsf{C}, \mathsf{carrot}\rangle, \mathsf{price}); \mathsf{price} <= 10? \\ & \overline{\mathsf{buy}}(\langle \mathsf{pdata}, \mathsf{yes}\rangle); (\mathsf{receipt}(\langle \mathsf{idorder}, \mathsf{idtran}\rangle) \mid \mathsf{commit}(\emptyset))\end{aligned}$$

[8]We remind that service behaviour starting application well-formedness rules are defined in Definition 3.3.

$, \mathcal{S}_{C0})]_C$

Services system. The system is modelled by exploiting the services system calculus and it is represented by the following process:

$$Sys := REG \parallel C \parallel B \parallel BI \parallel M \parallel S \parallel SI$$

3.5 Discussion on the semantics

In this section we discuss some interesting issues raised by the semantics of the presented calculi.

3.5.1 Concurrent sessions with a persistent state

Considering a service engine which allows to execute sessions in a concurrent way with a persistent state (ref. CONCURRENTPERSISTENT rule of Table 3.6) it is possible to notice that sessions can communicate each other by means of the shared state. In this case mutual exclusion could be necessary for accessing variables. In order to achieve it, it is possible to deploy the service engine equipped with a process in addition for supporting critical section implementation. Let us consider the following example:

$Q := \texttt{true} \rightleftharpoons (lock_y; \overline{unlock_y})$
$P := in(\emptyset,\ y,\ \overline{lock_y}; y := y + 5); unlock_y; \mathbf{0}$
$Y := !(\{\emptyset\} \triangleright P_\bullet)@L[\{\emptyset\} \triangleright (Q, \mathcal{S}[0/y])]_L$

Q is a looping process[9] which allows us to implement mutual exclusion on accessing y by means of signals $lock_y$ and $unlock_y$. The session P is formed by the Request-Response in which does not receive any value and it replies by sending the information contained within valriable y. Between the request and the response the service behaviour takes the lock of the variable y by means of the signal $\overline{lock_y}$, then it executes the process $y := y + 5$

[9]For the sake of this example, `true` models a condition which is always verified.

and, at the end, it waits for the unlocking signal. The service engine Y is initialized with a state where $y = 0$ and with the process Q that is running. Let us consider the case where two sessions are initiated by two other services located at l and l' respectively:

$$\begin{aligned} Y := {} & !(\{\emptyset\} \triangleright P_\bullet)@L[\{\emptyset\} \triangleright (\overline{\mathsf{lock}_y}; y := y + 5; \overline{\mathsf{in}}@l(y); \mathsf{unlock}_y; \mathbf{0} \\ & \quad \mid \overline{\mathsf{lock}_y}; y := y + 5; \overline{\mathsf{in}}@l'(y); \mathsf{unlock}_y; \mathbf{0} \mid Q, \mathcal{S}[0/y])]_L \end{aligned}$$

Depending on the first session which takes the lock of y the invokers will receive different values ($y = 5$ or $y = 10$). It is worth noting that mutual exclusion can be required also on the variables received within the first action. Indeed, let us consider the case above modified as follows:

$$P := \mathsf{in}(x,\ y,\ \overline{\mathsf{lock}_y}; y := x + y + 5); \mathsf{unlock}_y; \mathbf{0}$$

Now, let us consider the case where a first session is initiated with $x = 3$:

$$\begin{aligned} Y := {} & !(\{\emptyset\} \triangleright P_\bullet)@L[\{\emptyset\} \triangleright (\overline{\mathsf{lock}_y}; y := x + y + 5; \overline{\mathsf{in}}@l'(y); \mathsf{unlock}_y; \mathbf{0} \mid Q, \\ & \mathcal{S}[0/y, 3/x])]_L \end{aligned}$$

If in the meantime another session is initiated with $x = 4$ we have that:

$$\begin{aligned} Y := {} & !(\{\emptyset\} \triangleright P_\bullet)@L[\{\emptyset\} \triangleright (\overline{\mathsf{lock}_y}; y := x + y + 5; \overline{\mathsf{in}}@l'(y); \mathsf{unlock}_y; \mathbf{0} \\ & \quad \mid \overline{\mathsf{lock}_y}; y := x + y + 5; \overline{\mathsf{in}}@l'(y); \mathsf{unlock}_y; \mathbf{0} \mid Q, \mathcal{S}[0/y, 4/x])]_L \end{aligned}$$

In this case the value $x = 3$ has been lost because x is directly accessed when the message is received on the operation in. Mutual exclusion on variables received on the first actions can be achieved by exploiting correlation sets. The example above becomes:

$$\begin{aligned} & P := \mathsf{in}(x,\ y,\ \overline{\mathsf{lock}_y}; y := x + y + 5); \mathsf{unlock}_y; x := \perp; \mathbf{0} \\ & Y := !(\{x\} \triangleright P_\bullet)@L[\{x\} \triangleright (Q, \mathcal{S}[0/y])]_L \end{aligned}$$

When the first session is initiated the service engine Y appears as it follows:

$$Y := !(\{x\} \triangleright P_\bullet)@L[\{\emptyset\} \triangleright (\overline{lock_y}; y := x + y + 5; \overline{in}@l'(y); unlock_y; \mathbf{0} \mid Q,$$
$$\mathcal{S}[0/y, 3/x])]_L$$

In this case, the second session cannot be initiated because the variable x is correlated on the value 3. In order to allow for a new session starting, it is necessary to assign the value $\perp$ to the variable x. In this case indeed, the correlation does not take place because x is not initialized and a new session with a new value for x can be initiated. Clearly, this is a short example where the engine acts like a sequential one but it is possible to imagine more complicated service behaviours where the sessions run concurrently and mutual exclusion is fundamental for accessing variables in a suitable way.

3.5.2 System deadlock

A services system can reach a deadlock depending on the service engine execution modalities. Indeed, if an invoker does not take into account the fact that the service it is interacting with is deployed in a sequential order or in a concurrent one, a system deadlock is possible. Thus, to the end of service composition, it is not sufficient to consider only service behaviour but it is necessary to take into account deployment features too. Let us consider the following example where we assume that the Bank Master Service and the Bank Information Service of Section 3.4 are modified as follows:

$$P_{BI} := transac(\langle data, idorder, euro\rangle, \langle data, idtran\rangle,$$
$$\quad idtran := \texttt{trans(}data, idorder, euro\texttt{)}$$
$$)$$
$$+verify(\langle data, euro\rangle, \langle data, ver\rangle,$$
$$\quad ver := \texttt{verif(}data, euro\texttt{)}$$
$$)$$
$$BI := (\emptyset \triangleright P_{BI\bullet})^*@BI[\emptyset \triangleright (\mathbf{0}, \mathcal{S}_{BI0})]_{BI}$$

$$
\begin{aligned}
P_B := {} & pay(\langle data, idorder, loc, euro\rangle, ver, \\
& \quad \overline{verify}@BI(\langle data, idorder, euro\rangle, \langle data, ver\rangle) \\
& \quad); \\
& ; confirm(\langle conf, data\rangle) \\
& ; conf = yes? \\
& \qquad \overline{transac}@BI(\langle data, idorder, euro\rangle, \langle data, idtran\rangle) \\
& \qquad ; \overline{receipt}@loc(idtran) \\
B := {} & (\{data\} \triangleright P_B^*)@B[\{data\} \triangleright \mathbf{0}]_B
\end{aligned}
$$

The services has been redesigned and they work as it follows:

- The Bank Information Service does not process a transaction between two participants but it process only a participant for each invocation of the Request-Response operation $transac$. The Request-Response operation $verify$ checks the availability of the bank account of the participant described by the variable $data$.

- The Bank Master Service will start a session for each participant. The operation pay is a Request-Response operation where, between the request and the response, the Bank Master Service invokes the Request-Response operation $verify$ for verifying if the transaction can be performed, as a response the Bank Master Service sends the verification result to the invoker. Furthermore, on the operation $confirm$, the Bank Master Service waits for a confirmation from the invoker and, if the confirmation is positive, it continues to perform the transaction by invoking the Request-Response operation $transac$. It is worth noting that in this example the engine is defined as a sequential engine and we assume the variable $data$ is sufficient for correlating sessions.

- A transaction between two participants can be achieved as it follows: let us consider another e-commerce service eCS which, at a given time, invokes twice the Bank Master Service for performing two transactions for a seller and a buyer located at l_1 and l_2 respectively:

$$
\begin{aligned}
eSC ::= \ldots[\ldots; (&\overline{pay}@B(\langle dataS, 1234, l_1, 100\rangle, ans1); ans1 = yes?\\
&\qquad\qquad\qquad\qquad \overline{confirm}(\langle dataS, yes\rangle)\\
|\ &\overline{pay}@B(\langle dataB, 1234, l_2, -100\rangle, ans2); ans2 = yes?\\
&\qquad\qquad\qquad\qquad \overline{confirm}(\langle dataB, yes\rangle)\ldots]_{eSC}
\end{aligned}
$$

The eSC executes two parallel threads where, within the former, it invokes the Bank Master Service by sending the data of the seller and, within the latter, it sends the data of the buyer. The amount of the transaction is 100 euros and the $idorder$ is 1234. As far as of the seller is concerned, the eSC asks for adding 100 euros to its bank account whereas, as far as the buyer is concerned, it asks for subtracting 100 euros. Since the Bank Master Service is executed in a sequential order, the two parallel threads of the eSC service will be processed sequentially.

Now, it is reasonable to design the eSC service in a way that it waits for the verifications of both the buyer and the seller accounts before continuing the transaction. Such an issue can be achieved by introducing an internal synchronization between the two parallel threads. In particular, we introduce the signal $sync$ as it follows:

$$
\begin{aligned}
eSC ::= \ldots[\ldots; (&\overline{pay}@B(\langle dataSeller, 1234, l_1, 100\rangle, ans1); ans1 = yes?\\
&\qquad\qquad\qquad\qquad \overline{sync}; \overline{confirm}(\langle dataS, yes\rangle)\\
|\ &\overline{pay}@B(\langle dataBuyer, 1234, l_2, -100\rangle, ans2); ans2 = yes?\\
&\qquad\qquad\qquad\qquad sync; \overline{confirm}(\langle dataB, yes\rangle)\ldots]_{eSC}
\end{aligned}
$$

In this case, the e-commerce service starts two transactions by invoking the Request-Response operation pay. If both the transactions can be performed (because they are verified within the Bank Information Service), it internally synchronizes them by exploiting the signal $sync$. After that, it proceeds with the confirmation. The composed system $eSC \parallel B \parallel BI$ reaches a deadlock because B is executed in a sequential order. Indeed, let us consider the eCS service when the seller theread is initiated and the $ans1$ variable is yes:

$$\begin{aligned} eSC ::= \ldots[\ldots\overline{sync}; \overline{confirm}(\langle dataS, yes\rangle) \\ \mid \overline{pay}@B(\langle dataBuyer, 1234, l_2, -100\rangle, ans2); ans2 = yes? \\ sync; \overline{confirm}(\langle dataB, yes\rangle)\ldots]_{eSC} \end{aligned}$$

The eCS service has two concurrent processes, the former is waiting for internally synchronizing on the $sync$ signal whereas the latter has to initiate the second transaction. The deadlock is reached because the Bank Master Service is waiting for a confirmation in the session opened for the seller and the eCS service is waiting for an internal synchronization which can not be fulfilled because it needs to initiate also a buyer session on the Bank Master Service. But it is impossible to start the buyer session because the Bank Master Service is executed in a sequential order.

3.5.3 Request-Response and Solicit-Response

The service oriented computing basic communication mechanisms are the operations. A SOC application must be developed over a communication framework layer which guarantees that message exchanges are based upon the operation mechanisms. As far as the double message operations are concerned, there is an implicit location mobility[10] related to the request message. In a Request-Response message exchange indeed, the location of the sender is coupled to the receiver in the moment of the request reception. This is due to the fact that the receiver needs to know the communication channel, provided by the sender, it has to exploit in order to send its reply message. At the present, the Request-Response mechanism in service oriented computing has not been formally defined yet and there is an interesting issue which deserves to be discussed. It is not clear if the exchanged location can be used at the level of the application or it has to be hidden into the communication mechanism. Such a choice has a deep impact on the semantics of the service oriented application languages primitives. In WS-BPEL, when a request message of a Request-Response is received, the sender reference is automatically bound to the `partnerLink` construct which represents the sender service within

[10]Location mobility will be deeply discussed in the Section 4. Briefly, it deals with the communication of a service location by means of a message exchange.

the WS-BPEL workflow. Since in WS-BPEL it is not possible to manage such a kind of information at the level of the application, here we have modelled this feature within rule REQ-IN of Table 3.2 and rule REQ-SYNC of Table 3.8 where the location is hidden within the Request-Response mechanism. Such a kind of issue can be observed in the example of the Market Service 3.4 where the $getPrice$ Request-Response requires also the location loc of the sender. In order to reply to the customer as far as the $getPrice$ operation is concerned, the Market Service does not need the location loc because it is implicit in the Request-Response mechanism but it requires it for performing the Notification $\overline{commit}@loc$ operation and for invoking the Bank Master Service by means of the $\overline{pay}$ Notification. The fact that the exchanged location, in a Request-Response operation, is not visible at the level of the application has a consequence also for the Solicit-Response primitive. Let us consider the following case:

$P := a(x, y, y = x + 1)$
$Y := !(\emptyset \triangleright P_\times)@L[\emptyset \triangleright \mathbf{0}]_L$

$W := !(\emptyset \triangleright Q_\times)@L'[\emptyset \triangleright (...\overline{a}(h, z)@L..., \mathcal{S}[5/h]) \mid (...\overline{a}(h, k)@L..., \mathcal{S}[6/h])]_{L'}$

where Y is a service engine which exhibits only a Request-Response operation a that adds 1 to the passed variable x and stores the result into the variable y which will be sent within the response. W is a service sketch where, for the sake of this example, we do not explicit Q. Here we assume that, at a given point, two Solicit-Response $\overline{a}$ are perfomed by two sessions in parallel[11]. After the two requests of the two Request-Response processes are performed the services become as follows:

$P := a(x, y, y = x + 1)$
$Y := !(\emptyset \triangleright P_\times)@L[\emptyset \triangleright (\overline{a}@L'(y), \mathcal{S}[6/y]) \mid (\overline{a}@L'(y), \mathcal{S}[7/y])]_L$

[11]For the sake of clarity, we have represented the passed values by means of the constants 5 and 6 even if, formally, there should be two variables which refer to the current state.

$W := !(\emptyset \triangleright Q_\times)L'[\emptyset \triangleright (...a(z)..., \mathcal{S}[5/h]) \mid (...a(k)..., \mathcal{S}[6/h])]_{L'}$

In this case, there is nothing which allows us to couple the replies to the right Solicit-Response because the only reference we use to synchronize the message exchange are the name of the operation and the location. Such a situation can be avoided by exploiting correlation sets:

$P := a(x, \langle x, y \rangle, y = x + 1)$
$Y := !(\emptyset \triangleright P_\times)@L[\emptyset \triangleright \mathbf{0}]_L$

$W := !(\{h\} \triangleright Q_\times)@L'[\{h\} \triangleright (...\overline{a}(h, \langle h, z \rangle)..., \mathcal{S}[5/h]) \mid (...\overline{a}(h, \langle h, k \rangle)..., \mathcal{S}[6/h])]_{L'}$

This solution implies that both services must be modified. It is a solution at the level of services system and there should be cases where it is not possible to modify all the services involved in a system. In those cases it is possible to design *W* with a persistent state and, by means of shared variables, synchronize the Solicit-Responses in order to execute them sequentially.

3.6 Comparing SOCK and WS-BPEL

This section is devoted to compare SOCK and WS-BPEL expressiveness in order to highlight the differences and the similarities between the two languages. Such a kind of comparison will be traced considering the service behaviour calculus constructs and the service engine calculus mechanisms. The services system calculus will be not taken into account because it does not directly deal with the WS-BPEL language constructs. It is worth noting that the comparison that follows will not take into consideration faults and long running transaction mechanisms which are not modelled within SOCK.

3.6.1 Service behaviour calculus

In Tables 3.9 and 3.10 are reported the corresponding relationships between the constructs of the SOCK language an those of WS-BPEL. In the following we discuss the constructs:

- The input signal and the output one are mapped with the tags <*sources*> and <*targets*> respectively. In WS-BPEL it is possible to specify some boolean guard within the source and the target tags. Such a kind of guard aims at enabling or disabling the link depending they are true or false. In SOCK it is not possible to directly join a condition with a signal but it is possible to exploit the *if_then_else* process in order to obtain the same result. In the following we present an example where, after the execution of a message on the One-Way rec, an output signal $sync$ is executed only if the variable x is greater than zero. Such a kind of signal will enable the notification $\overline{send}$ defined in a parallel process.

  $EX ::= rec(x); x > 0?\overline{sync} : \mathbf{0} \mid sync; \overline{send}@l(x)$

  It is worth noting that in WS-BPEL the source and target links must be defined within an activity. The example above can be translated as follows:

```
<flow>
 <receive ... operation="rec" variable="x">
  <sources>
   <transitionCondition ...> x > 0 </transitionCondition>
   <source linkName="sync" />
  </sources>
 </receive>
 <invoke .... operation="send" outputVariable="x">
  <targets>
   <target linkName="sync" />
  </targets>
 </invoke>
</flow>
```

- The One-Way is mapped in WS-BPEL with a receive activity. It is worth noting that a receive activity needs to specify both the partnerLink and the portType. The former is not modelled in SOCK where partnerLink are trivially modelled by considering the fact that a One-Way exhibited by a service engine can be invoked only by a service engine which knows the name of the operation and the location of the receiver. The latter is not mapped in SOCK because in our calculus we abstract away from such a kind of construct, indeed in SOCK engine all the operations are deployed within the same location without any distinction.

- The Notification is mapped in WS-BPEL with an invoke activity where only the output variable is specified. In SOCK a location is needed in order to deliver the message to the right receiver, in the invoke activity such an information is retrieved by following the partnerLink reference.

- The Request-Response in SOCK is modelled with a single primitive whereas in WS-BPEL it is formed by a receive activity followed by a reply one. Between the receive and the reply the body P of the Request-Response is executed. It is worth noting that in WS-BPEL the reply is related to the receive by exploiting the partnerLink, the portType and the operation references.

- The Solicit-Response in WS-BPEL is modelled by exploiting an invoke activity where both the input variables and the output ones are specified. As for the Notification the location in WS-BPEL is retrieved by exploiting the partenrLink reference.

- The assignment, the if then else, the sequence, the parallel and the iteration are easy to determine from the tables and they do not need to be commented.

- The non deterministic choice is modelled in WS-BPEL with the activity *pick*. In a pick activity is possible to define a non-deterministic choice only on a set of incoming messages, whereas in SOCK there is the possibility to specify also an input signal. This fact fact allows for the specification of a race condition between an external message reception and an internal activity which is not possible in WS-BPEL. Let us consider the following example where a non determinitic choice is defined between a One-Way $\mathtt{rec}$ and the signal $\mathtt{sync}$.

$\mathsf{EX} ::= ((\mathtt{rec}(x) + (\mathtt{sync}; x := 3)); \overline{\mathtt{send}}@l(x)) \mid \overline{\mathtt{sync}}$

Depending on the fact that the output signal is executed before the One-Way the Notification $\mathtt{send}$ will send the value 3 or that received on the $\mathtt{rec}$ operation. We remind that, in Table 3.10, the `partnerLink` construct and the `portType` one allow for the identification of the operation and the partnerLinkType on which the message is received.

The WS-BPEL constructs *repeatUntil* and *forEach* are not explicitly modelled in SOCK but they can be easily mapped by exploiting the iteration process as it follows:

$\mathtt{repeatUntil} ::= P; \chi \rightleftharpoons P$
$\mathtt{forEach} ::= x < N \rightleftharpoons (P; x := x + 1)$

3.6.2 Service engine calculus

The service engine calculus allows for the specification of the state modality (shared or not shared), the correlation set and the execution modality (sequence or concurrent). In the following we discuss such a kind of features w.r.t. WS-BPEL.

- *State modality*: WS-BPEL does not allow for the specification of a shared state or a not shared one. Each instance has always its own state.
- *Correlation set*: WS-BPEL has a specific construct for defining the correlation set which identifies an instance. Every activity that must be correlated has to specify the correlation set to use for correlating. Differently, In SOCK the correlation set is expressed by a set of variables whose values determine the session and the routing mechanism is intrinsic within the reception primitives without specifying the correlated data. In the following we present an example where a receive activity in WS-BPEL specifies the correlation data:

```
...
<receive name="receiveAuctionRegistrationInformation"
partnerLink="auctionRegistrationService"
portType="as:auctionRegistrationAnswerPT"
operation="answer"
variable="auctionAnswerData">
 <correlations>
  <correlation set="auctionIdentification" />
 </correlations>
</receive>
...
```

In this example the correlation set *auctionIdentification* refers to a variable *auctionId* which is a part of the variable *auctionAnswerData*. Such a variable indeed contains two parts: *auctionId* and *CreditCardNumber*. The corresponding SOCK engine is:

$P := ...; answer(\langle auctionId, CreditCardNumebr \rangle); ...$
$Eng ::= !(\{auctionId\} \triangleright P_{\times})@l[\{auctionId\} \triangleright \mathbf{0}]$

- *Execution modality*: such a feature is not mentioned within the WS-BPEL specification and depends on the engine which executes the process even if, at the best of our knowledge, there are no engines which distinguish between the two modalities, the instances are always executed in a concurrent way.

3.6.3 Modelling a WS-BPEL example with SOCK

In this section we present the WS-BPEL *Auction service* of Section 2.1.3.3 modelled with the SOCK calculus where, for the sake of simplicity, we use short variable names. In the following we show the variable name correspondances:

$\$sellerData.CreditCardNumber \mapsto CCNS$
$\$sellerData.ShippingCosts \mapsto ShipC$

	SOCK	$WS - \mathrm{BPEL}$

input signal	s	`<sources>`
		` <source linkName="ship-to-invoice" />`
		`</sources>`

output signal	$\overline{s}$	`<targets>`
		` <target linkName="ship-to-invoice" />`
		`</targets>`

One-Way	$o(x)$	`<receive partnerLink=?`
		`portType=?`
		`operation="o"`
		`variable="x">`
		`</receive>`

Notification	$\bar{o}@k(x)$	`<invoke partnerLink=[k]`
		`portType=?`
		`operation="o"`
		`outputVariable="x">`
		`</invoke>`

Request-Response	$o_r(x, y, P)$	`<receive partnerLink=?`
		`portType=?`
		`operation="o_r"`
		`variable="x">`
		`</receive>`
		`P`
		`<reply partnerLink= [received location]`
		`portType=?`
		`operation="o_r"`
		`variable="y">`
		`</reply>`

Solicit-Response	$\overline{o_r}@k(x, y)$	`<invoke partnerLink=[k]`
		`portType=?`
		`operation="o_r"`
		`inputVariable="y"`
		`outputVariable="x">`
		`</invoke>`

Table 3.9: Comparing communication primitives between SOCK and WS-BPEL

	SOCK	$WS - BPEL$
Assigment	$x := e$	`<assign>` `<copy>` `<from expressionLanguage="anyURI">` `e` `</from>` `<to variable="x"/>` `</copy>` `</assign>`
If then else	$\chi?P : Q$	`<if>` `<condition>chi</condition>` `P` `<else> Q </else>` `</if>`
Sequence	$P; Q$	`<sequence>` `P` `Q` `</sequence>`
Parallel	$P\|Q$	`<flow>` `P` `Q` `</flow>`
Non-det. choice	$\sum^{+}_{i\in W} \epsilon_i; P_i$	`<pick>` `...` `<onMessage partnerLink=? portType=?` `operation="epsilon_i">` `P_i` `</onMessage>` `...` `</pick>`
Iteration	$\chi \rightleftharpoons P$	`<while>` `<condition> chi </condition>` `P` `</while>`

Table 3.10: Comparing constructs of SOCK and WS-BPEL

$\$sellerData.auctionId \mapsto auctionId$

$\$sellerData.endpointReference \mapsto EPRS$

$\$buyerData.CreditCardNumber \mapsto CCNB$

$\$buyerData.phoneNumber \mapsto phN$

$\$buyerData.ID \mapsto auctionId$

$\$buyerData.endpointReference \mapsto EPRB$

$\$auctionData.amount \mapsto amount$

$\$auctionAnswerData.registrationId \mapsto regId$

It is worth noting that the variable $auctionId$ maps both the variable $\$sellerData.auctionId$ and the variable $\$buyerData.ID$. This is due to the fact that such a variable it will be used for correlating the sessions. We remind that in WS-BPEL such a kind of correspondance is obtained by exploiting the *property* construct and the *propertyAlias* one. Indeed, in the example of section 2.1.3.3 the variables $\$sellerData.auctionId$ and $\$buyerData.ID$ refer to the same property by means of a propertyAlias declaration. The SOCK example code follows:

```
Bh ::= (submitS(⟨CCNS, ShipC, auctionId, EPRS⟩)
         ; submitB(⟨CCNB, phN, auctionId, EPRB⟩)
        +
         submitB(⟨CCNB, phN, auctionId, EPRB⟩)
         ; submitS(⟨CCNS, ShipC, auctionId, EPRS⟩)
        )
         amount := 1; RS := ref; myLoc := AD
         ; $\overline{process}$@RS(amount, auctionId, myLoc)
         ; AuctionRegInf(regId, auctionId); msg :=' Thank you'
        ;
        ($\overline{answer}$@EPRS(msg)
         |
         $\overline{answer}$@EPRB(msg)
        )
```

$\mathsf{Engine} ::= !(\{\mathsf{auctionId}\} \triangleright \mathsf{Bh}_\times)[\{\mathsf{auctionId}\} \triangleright \mathbf{0}]$

The service behaviour starts with a non-determinist choice between the two One-Ways $\mathsf{submitS}$ and $\mathsf{submitB}$. In this case indeed, it is not possible to predict which One-Way will be firstly executed but both contain the correlated variable $\mathsf{acutionId}$. In this way the first One-Way will initiate a session and the second one will refer to it by means of the variable $\mathsf{auctionId}$. Such a construct in WS-BPEL is modelled by exploiting a parallel composition between two receives where the correlation set tag has the attribute *initiate* set on the value *join*. This means that the first receive will set the correlation set value and the second will refer to it for routing the message to the right instance. Moreover, in the example we model the endpoint reference of the Registration Service with the value ref and we store it into the variable RS.

3.6.4 Comments

Here we want to briefly summarize the differences between SOCK and WS-BPEL as languages for representing service orchestrators. The fact that WS-BPEL is defined by exploting XML, implies that the code is verbose and difficult to be read by a human designer. Indeed, WS-BPEL engines are always released with visual tools for designing orchestrators for its high level of management complexity. On the contrary, SOCK supplies a limited set of constructs which allows for a short, clear and immediate comprehension of the code. In particular, the non-deterministic choice of SOCK introduce the possibility to program a race between an external message reception and an internal signal that it is not possible within WS-BPEL. SOCK models and give constructs for directly representing engine features such as the persistent state and the execution modality that are not considered within WS-BPEL and the correlation set mechanism in SOCK is very concise w.r.t. that of WS-BPEL. Furthermore, SOCK is equipped with a formal semantics that it is not supplied for WS-BPEL and that allows us to supply a precise implementation of SOCK. In Chapter 10 indeed, we will present JOLIE that is an implementation of the service behaviour part of SOCK which we intend, in the future, to promote as a good alternative w.r.t. WS-BPEL for designing services. Moreover, we also intend to develop an algorithm for traslating SOCK in WS-BPEL in order to provide an easy means for programming WS-BPEL orchestrators. By exploiting such a kind of algorithm indeed, it will be possible to design an orchestrator in JOLIE and then traslating it in WS-BPEL. Furthermore, SOCK formalization can be exploited to the end of the WS-BPEL specification development in order to eliminate code redudancy. WS-BPEL indeed, provides a lot of constructs whose behaviour can be altered by adding some specific attributes. Such a fact raises difficulties when WS-BPEL orchestrators must be designed for dealing with complex tasks that involve a lot of services. For instance, let us consider the case of the Auction Service example presented in Section 2.1.3.3 where, within the correlation tag contained into the two initial concurrent receives, it is necessary to include the tag `initiate="join"` in order to specify that only the first activity which receives the message can initialize the correlation set. In SOCK we have formalized such a behaviour by exploiting the existing non-deterministic choice, as presented in Section 3.6.3, without adding any particular construct.

3.7 Related works

There are other works which exploit formal models for representing services and service composition. In general, they use different models for representing service behaviours and service composition but they do not deal with service deployment features as persistent or not persistent state and concurrent or sequential execution modality. In [DD04] the authors use Petri Nets for describing service behaviours, called *provider behaviour*, but they focus only on workflow aspects without distinguishing among the different kind of operations (single message exchange and double message exchange). In [LM07] Mazzara and Lucchi define the semantics of WS-BPEL in terms of pi-calculus processes where operations, scopes and componing operators are defined even if they do not deal with correlation sets. In [MC06] the authors present a language, called *Orc*, where services are considered as functions and they are called *sites*. In Orc communication mechanisms abstract away from operations primitives and a service invocation is expressed by a function call, namely a site call. In [BBC$^+$06], the authors propose a calculus for service design and composition based on the pi-calculus without dealing with the deployment aspects as the variable state, the correlation sets and the execution modalities. In [LPT06] the authors present a calculus inspired to WS-BPEL, which does not take into account the Request-Response operations, and they propose a type system for enforcing many of the constraints imposed by WS-BPEL and WSDL specifications. Finally, as far as correlation sets are concerned, in [Vir04] Viroli propose a first formalization of the mechanism specified within BPEL4WS specifications. Viroli distinguishes between variables and properties where the former are syntactic elements related to some variables which allows for the identification of a session.

Chapter 4

Mobility mechanisms

Starting from SOCK we analyze the different kind of mobility within Service Oriented Computing. We distinguish among the *state mobility*, the *location mobility*, the *interface mobility* and the *behaviour mobility*. The state mobility models the data exchange between the states of two services by means of a communication message. The location mobility models the possibility to pass, by means of a message exchange, service locations. The interface mobility allows for the representation of the communication of information related to the interface of the service. Finally, the behaviour mobility represents a sort of code mobility where the exchange information represent a piece of service behaviour to execute by the receiver. This chapter is devoted to discuss such a kind of mobility mechanisms. This investigation is useful to the end of the system design issues because different kind of mobility mechanisms need different language primitives which straightforwardly affect the composition of a system. Moreover, these kinds of mobility mechanisms will be discussed within the Web Services technology where only the internal state mobility and the location one are actually implemented.

The chapter is organized as follows:

- we discuss mobility mechanisms by starting from the general model and the SOCK calculus presented in the previous section

- we exploit a subpart of the SOCK calculus for showing as the mobility mechanisms could affect Service Oriented Computing systems

- we discuss the mobility mechanisms w.r.t. the Web Services technology

4.1 Four kind of mobility mechanisms

Here we discuss mobility mechanisms by starting from SOCK calculus investigation presented in the previous section from which we extract the different parts a service is based upon. Then, for each of them we investigate its mobility mechanism exploiting a subpart of the SOCK calculus as a workbench. In particular, for the sake of this chapter, we can generally refer to the term service by considering a computational entity located at a specific unique *location* which has a *state* and is able to perform some *functionalities*. A service can receive a message by means of an input operation and it can send a message by means of an output one. The set of all the input and output actions of a service represents the *interface* of the service. Let Loc, ranged over by l, be the set of locations where $\mathrm{Loc} \subseteq \mathit{Val}$. Formally, here we represent a service by means of the following tuple:

$$\mathit{Service} := (I, \mathcal{M}, P_f, l)$$

where I is the interface containing all the input and output actions it can use, $\mathcal{M}$ is the state of the service we use to represent all the information it manages (e.g. variables, databases), P_f is the service behaviour encoded by exploiting the formalism f and $l \in \mathit{Loc}$ is the location where the service is deployed. We remark that, in order to be as general as possible, in this section we abstract away from the specific formalism f and the representation of the state; in the next section such notions will be represented by exploiting a subset of SOCK.

4.1.1 The mobility mechanisms

In this section we describe the mobility mechanisms related to each element of the service tuple previously described, that is: state mobility, location mobility, interface mobility and behaviour mobility.

- ***State mobility:*** The mobility of the state is strongly related to the SOC communication mechanism which is based on message exchange. Indeed, the content of a sent message is part of the information contained in the state of the sender that the receiver acquires and stores in its state. In other words, a message exchange between two services can be seen as an information mobility from the sender state to the receiver one.

- ***Location mobility:*** Location mobility deals with the possibility to receive a location by means of a message exchange and to exploit it to access the service deployed at that location. In this way, for instance, a service can acquire, at run-time, the exact location of a service whose functionalities are known.

- ***Interface mobility:*** Interface mobility means that a service can acquire at run-time all the infomation about an input or an output action and then it can exhibit it in its interface. In other words, it can receive an input or an output action and execute it as it belongs to its interface.

- ***Behaviour mobility:*** The mobility of this component implies that a service behaviour can be communicated within a message exchange and executed by the service which receives it. In this case the receiver can enrich its internal behaviour by executing the received one. It is important to highlight the fact that the receiver must be able to execute the received behvaiour by exploiting the specific formalism used for encoding it (the issues related to this aspect are out of the scope of this thesis).

It is worth noting that there exist a correspondence between some of these notions and those from process calculi, on which we shall discuss in Section 4.4.

4.2 Using SOCK for discussing the mobility mechanisms

This section is devoted to model the mobility mechanisms discussed above by exploiting a subpart of the SOCK language. In particular, we proceed as it follows:

- we extend the operation definition in order to consider the operation templates which allow for the definition of the expected data type joint to a message. Operation will be identified by a name and a template
- we define the subpart of SOCK that we will exploit as a workbench for reasoning about the mobility mechanisms
- we formalize all the mobility mechanisms by extending the selected subpart of the SOCK calculus and we describe how services are affected by them.

4.2.1 Extending operation definition with templates

In Service oriented computing, messages are structured containers of typed data. Here we start by considering a single data type we name *information* which represents a general information exploited by a SOC application and, in the following, we will introduce, step by step, additional data types which will be exploited to support mobility; in particular the *location*, the *operation*, the *template* and the *behaviour* data types. Informally, locations univocally identify the services in the system, operations and templates define the interface of services and, finally, behaviours represent the internal behavior of services. For the sake of this work, we abstract away from a detailed classification of types even if it is possible to refine types classification by considering other data types (e.g. integer, float, string). As far as the message structure is concerned, for the sake of generality, here we consider a flat structure where messages are seen as arrays of typed data. In the following message structures will be described by introducing the notion of message *template* where a template describes the expected sequence of data types contained within a message.

Let $\mathtt{inf}$ be the type denoting the generic information, $\mathcal{T}$, ranged over by $\vec{t}$, be the set of templates defined as arrays of type elements. For example $\vec{t'} = \langle \mathtt{inf}, \mathtt{inf}, \mathtt{inf} \rangle$ represents the structure of a message with three elements whose type is $\mathtt{inf}$. Let Val, ranged over by v, be the set of values on which is defined a total order relation, $\mathsf{InfVal} \subseteq \mathsf{Val}$, ranged over by δ, be the set of generic information and Type be the function that, given $v \in \mathsf{Val}$, returns the type of v. Since currently we are considering only the generic information

type, we define $Type(v) = inf$ if $v \in InfVal$; the remaining cases where $v \notin InfVal$ will be defined in the following where additional types are introduced. We denote with $\vec{v} = \langle v_0, v_1, ..., v_n \rangle$ an array of values.
Let $\vec{t} = \langle t_1, \dots, t_n \rangle$ be a template and $\vec{v} = \langle v_1, \dots, v_s \rangle$ be an array of values, we say that $\vec{v}$ satisfies $\vec{t}$, denoted as $\vec{t} \vdash \vec{v}$, if the following conditions hold:

1. $n = s$,

2. $\forall v_i, Type(v_i) = t_i$.

So far, we have distinguished the operation names into two different sets: $\mathcal{O}$ and $\mathcal{O}_R$ where the former identifies the single message operation names and the latter the double message operation ones. Now, we consider a single set of operation names because single message and double message operations will be univocally distinguished by means of the associated templates. Formally, let $\mathcal{O} \subseteq Val$ be a set of operation names and let Op be the set of operations defined as it follows:

$$
\begin{aligned}
&Op = \{(o, ow, \vec{t}) \mid o \in \mathcal{O}, \vec{t} \in \mathcal{T}\} \\
&\cup \{(o, n, \vec{t}) \mid o \in \mathcal{O},\ \vec{t} \in \mathcal{T}\} \\
&\cup \left\{(o, rr, \vec{t}, \vec{t'}) \mid o \in \mathcal{O},\ \vec{t}, \vec{t'} \in \mathcal{T}\right\} \\
&\cup \left\{(o, sr, \vec{t}, \vec{t'}) \mid o \in \mathcal{O},\ \vec{t}, \vec{t'} \in \mathcal{T}\right\}
\end{aligned}
$$

an operation is identified by a name (o), an interaction modality (ow, n, rr and sr represent One-Way, Notification, Request-Response and Solicit-Response interaction modalities respectively) and one or two templates ($\vec{t}, \vec{t'}$) depending on the fact that the operation deals with a single message (One-Way and Notification operations) or two messages (Request-Response or Solicit-Response operations). In the former case, $\vec{t}$ represents the template of the exchanged message whereas in the latter one $\vec{t}$ represents the template of the request message and $\vec{t'}$ represents the template of the reply one. In the following we use $o_{\vec{t}}$, $\overline{o}_{\vec{t}}$, $o_{\vec{t},\vec{t'}}$ and $\overline{o}_{\vec{t},\vec{t'}}$ to range over Op where $o_{\vec{t}}$ represents a One-Way operation whose name is o and the joint template is $\vec{t}$, $\overline{o}_{\vec{t}}$ represents a Notification operation whose name is o and the joint template is $\vec{t}$, $o_{\vec{t},\vec{t'}}$ represents a Request-Response operation whose name is o and the joint templates are $\vec{t}$ for the receiving message and $\vec{t'}$ for the sending one and, finally, $\bar{o}_{\vec{t},\vec{t'}}$ represents a Solicit-Response operation whose name is o and the joint templates are $\vec{t}$ for the sending message and $\vec{t'}$ for the receiving one. We say that two operations $o_{\vec{t}}$ and $\overline{o'}_{\vec{t'}}$ are *dual* if $o = o'$ and $\vec{t} = \vec{t'}$. Analogously, we say that two operations $o_{\vec{t},\vec{t'}}$ and $\overline{o'}_{\vec{t''},\vec{t'''}}$ are *dual* if $o = o'$, $\vec{t} = \vec{t}''$ and $\vec{t}' = \vec{t}'''$. Formally we define duality in the following way:

$o_{\vec{t}} \bowtie \overline{o'}_{\vec{t}'} \Leftrightarrow o = o' \wedge \vec{t} = \vec{t}'$
$o_{\vec{t},\vec{t}'} \bowtie \overline{o'}_{\vec{t}'',\vec{t}'''} \Leftrightarrow o = o' \wedge \vec{t} = \vec{t}'' \wedge \vec{t}' = \vec{t}'''$.
It is worth noting that the Interface I is a subset of the operation set: $I \subseteq Op$. Furthermore, if we reconsider the Interface mobility considering the new definition of operations we can see Interface mobility as the feature which allows for the communication of the templates and the operation names as usual information. Such a characteristic, from a designing point of view, allows a human designer to program an input or an output operation without specifying its name and/or its templates by considering the fact that they can be acquired dynamically during the execution of the service.

4.2.2 SOCK as a workbench

In the following we present a limited set of the SOCK syntax extended with the new definition of operation that we will use for analyzing the different aspects of each kind of mobility:

$P, Q ::= \mathbf{0} \mid x := e \mid \epsilon \mid \overline{\epsilon} \mid \chi?P : Q$
$\mid P;P \mid P \mid P \mid \sum^{+}_{i \in W} \epsilon_i; P_i \mid \chi \rightleftharpoons P$

$\epsilon ::= s \mid o_{\vec{t}}(\vec{x}) \mid o_{\vec{t},\vec{t}'}(\vec{x}, \vec{y}, P)$
$\overline{\epsilon} ::= \bar{s} \mid \bar{o}_{\vec{t}}@l(\vec{x}) \mid \bar{o}_{\vec{t},\vec{t}'}@l(\vec{x}, \vec{y})$

$P_S := (P, \mathcal{S})$

$E ::= [P_S]_l \mid E \parallel E$

where a service-based system E consists of the parallel composition of services. A service $[P_S]_l$ is a couple of a process P and a state $\mathcal{S}$ identified by a location $l \in Loc$. It is worth noting that, here, all the service engine features introduced within the SOCK calculus are omitted. All the other primitives have been defined in Chapter 3 where, as far as the operations are concerned, the process $o_{\vec{t}}(\vec{x})$ represents a One-Way operation where o is the name of the operation, $\vec{t}$ is the template of the received message and $\vec{x}$ is the array of variables where the received information will be stored. $o_{\vec{t},\vec{t}'}(\vec{x},\vec{y},P)$ represents the Request-Response operation where o is the name of the operation, $\vec{t}$ is the template of the received message and $\vec{t}'$ is the template of the sent message. $\bar{o}_{\vec{t}}@l(\vec{x})$ represents the Notification operation where o is the name of the operation, $\vec{t}$ is the template of the sent message, l is the location of the invoked service and $\vec{x}$ is the tuple of variables which contain the sent message. Finally, $\bar{o}_{\vec{t},\vec{t}'}@l(\vec{x},\vec{y})$ represents the Solicit-Response operation, where o is the name of the operation, $\vec{t}$ is the template of the sent message, $\vec{t}'$ is the template of the received message, l is the location of the invoked service.

The semantics of such a language is redefined in a restrict way w.r.t. that of SOCK in order to deal with the introduction of the templates. It is defined in terms of a labelled transition system whose axioms and rules are reported in Tables 4.1, 4.2 and 4.3. In Table 4.1 we present the rules realted to the communication primitives where each rule requires that a received or a sent message must satisfy the current operation template in order to be performed. Table 4.2 deals with the rules over P_S where the behaviour of a process coupled with a state is expressed. Rule ASSIGN deals with variable assignment within the services; $e \hookrightarrow_{\mathcal{S}} w$ means that the evaluation process of the expression e within state $\mathcal{S}$ reduces to w. Rule INT-SYNC deals with internal synchronization over signals and CONGRP with internal structural congruence denoted by $\equiv_P$. PAR-INT and SEQ describe the behaviour of processes composed in parallel and sequentially respectively, whereas CHOICE and ITERATION 1/2 describe the behavior of the non-deterministic choice and the guarded iteration respectively. The former one non-deterministically selects an input guarded process among the ones listed in the choice operator, while the latter ones model

(In)

$$(s, \mathcal{S}) \xrightarrow{s} (\mathbf{0}, \mathcal{S})$$

(Out)

$$(\bar{s}, \mathcal{S}) \xrightarrow{\bar{s}} (\mathbf{0}, \mathcal{S})$$

(Notification)

$$\frac{\vec{t} \vdash \mathcal{S}(\vec{x})}{(\bar{o}_{\vec{t}}@l(\vec{x}), \mathcal{S}) \xrightarrow{\bar{o}_{\vec{t}}@l(\mathcal{S}(\vec{x}))} (\mathbf{0}, \mathcal{S})}$$

(One-Way)

$$\frac{\vec{t} \vdash \vec{v}}{(o_{\vec{t}}(\vec{x}), \mathcal{S}) \xrightarrow{o_{\vec{t}}(\vec{v})} (\mathbf{0}, \mathcal{S}[\vec{v}/\vec{x}])}$$

(Solicit)

$$\frac{\vec{t} \vdash \mathcal{S}(\vec{x})}{(\bar{o}_{\vec{t},\vec{t}'}@l(\vec{x}, \vec{y}), \mathcal{S}) \xrightarrow{\bar{o}_{\vec{t},\vec{t}'}@l(\mathcal{S}(\vec{x}), \vec{y})} (o_{\vec{t},\vec{t}'}@l(\vec{y}), \mathcal{S})}$$

(Request)

$$\frac{\vec{t} \vdash \vec{v}}{(o_{\vec{t},\vec{t}'}(\vec{x}, \vec{y}, P), \mathcal{S}) \xrightarrow{o_{\vec{t},\vec{t}'}@l(\vec{v}, \vec{y})} (P; \bar{o}_{\vec{t},\vec{t}'}@l(\vec{y}), \mathcal{S}[\vec{v}/\vec{x}])}$$

(Response-Out)

$$\frac{\vec{t}' \vdash \mathcal{S}(\vec{x})}{(\bar{o}_{\vec{t},\vec{t}'}@l(\vec{x}), \mathcal{S}) \xrightarrow{\bar{o}_{\vec{t},\vec{t}'}@l(\mathcal{S}(\vec{x}))} (\mathbf{0}, \mathcal{S})}$$

(Response-In)

$$\frac{\vec{t}' \vdash \vec{v}}{(o_{\vec{t},\vec{t}'}@l(\vec{x}), \mathcal{S}) \xrightarrow{o_{\vec{t},\vec{t}'}@l(\vec{v})} (\mathbf{0}, \mathcal{S}[\vec{v}/\vec{x}])}$$

Table 4.1: Communication rules

iteration behaviour. Finally, IF THEN and ELSE rules express the if-the-else semantics. In Table 4.3 the rules at the level of services system are considered. Rule ONE-WAYSYNC deals with the synchronization on a One-Way operation between two services whereas rules REQ-SYNC and RESP-SYNC deal with the request and the response message exchanges between a Solicit-Response operation and a Request-Response one. PAR-EXT deals with external parallel composition and CONGRE is for external structural congruence denoted by $\equiv$. INT-EXT expresses the fact that a service behaves in accordance with its internal processes.

Now, we remind the service formalization presented in the previous section where a service is represented by the tuple $(I, \mathcal{M}, P_f, l)$ and we show how a service $[P, \mathcal{S}]_l$ is related to it:

- $\mathcal{M}$ is modeled by $\mathcal{S}$.

- l represents the location within both the service model and the subpart of the SOCK language.

- P_f is represented by a process P where the formalism f corresponds to the subpart of the SOCK language.

- I represents the interface of a service and it is not explicitly modeled in the subpart of the SOCK language but it can be extracted from the process P. Indeed, by considering a service $[P, \mathcal{S}]_l$, its interface I is defined by the function $\Theta(P)$ where Θ is inductively defined by the following rules:

(ASSIGN)
$$\frac{e \hookrightarrow_{\mathcal{S}} v}{(x := e, \mathcal{S}) \xrightarrow{\tau} (\mathbf{0}, \mathcal{S}[v/x])}$$

(INT-SYNC)
$$\frac{(P, \mathcal{S}) \xrightarrow{s} (P', \mathcal{S})\ ,\ (Q, \mathcal{S}) \xrightarrow{\bar{s}} (Q', \mathcal{S})}{(P \mid Q, \mathcal{S}) \xrightarrow{\tau} (P' \mid Q', \mathcal{S})}$$

(CONGRP)
$$\frac{P \equiv_P P'\ ,\ (P', \mathcal{S}) \xrightarrow{\gamma} (Q', \mathcal{S}'),\ Q' \equiv_P Q}{(P, \mathcal{S}) \xrightarrow{\gamma} (Q, \mathcal{S}')}$$

(PAR-INT)
$$\frac{(P, \mathcal{S}) \xrightarrow{\gamma} (P', \mathcal{S}')}{(P \mid Q, \mathcal{S}) \xrightarrow{\gamma} (P' \mid Q, \mathcal{S}')}$$

(SEQ)
$$\frac{(P, \mathcal{S}) \xrightarrow{\gamma} (P', \mathcal{S}')}{(P; Q, \mathcal{S}) \xrightarrow{\gamma} (P'; Q, \mathcal{S}')}$$

(CHOICE)
$$\frac{(\epsilon_i; P_i, \mathcal{S}) \xrightarrow{\gamma} (P', \mathcal{S}') \quad i \in W}{(\sum^{+}_{i \in W} \epsilon_i; P_i, \mathcal{S}) \xrightarrow{\gamma} (P', \mathcal{S}')}$$

(ITERATION 1)
$$\frac{\mathcal{S} \vdash \chi}{(\chi \rightleftharpoons P, \mathcal{S}) \xrightarrow{\tau} (P; \chi \rightleftharpoons P, \mathcal{S})}$$

(ITERATION 2)
$$\frac{\mathcal{S} \nvdash \chi}{(\chi \rightleftharpoons P, \mathcal{S}) \xrightarrow{\tau} (\mathbf{0}, \mathcal{S})}$$

(IF THEN)
$$\frac{\mathcal{S} \vdash \chi}{(\chi?P : Q, \mathcal{S}) \xrightarrow{\tau} (P, \mathcal{S})}$$

(ELSE)
$$\frac{\mathcal{S} \nvdash \chi}{(\chi?P : Q, \mathcal{S}) \xrightarrow{\tau} (Q, \mathcal{S})}$$

(STRUCTURAL CONGRUENGE OVER P)

$$P \mid \mathbf{0} \equiv_P P \quad \mathbf{0}; P \equiv_P P$$

$$(P \mid Q) \equiv_P (Q \mid P) \quad (P \mid Q) \mid R \equiv_P P \mid (Q \mid R)$$

Table 4.2: Rules over P_S

(ONE-WAYSYNC)

$$\frac{[P_S]_l \xrightarrow{\bar{o}_{\vec{t}}@l'(\vec{v})} [P'_S]_l \, , \; [Q_S]_{l'} \xrightarrow{o'_{\vec{t}'}(\vec{v})} [Q'_S]_{l'}, \bar{o}_{\vec{t}} \bowtie o'_{\vec{t}'}}{[P_S]_l \parallel [Q_S]_{l'} \xrightarrow{\tau} [P'_S]_l \parallel [Q'_S]_{l'}}$$

(REQ-SYNC)

$$\frac{[P_S]_l \xrightarrow{\sigma} [P'_S]_l \, , \; [Q_S]_{l'} \xrightarrow{\sigma'} [Q'_S]_{l'}}{[P_S]_l \parallel [Q_S]_{l'} \xrightarrow{\tau} [P'_S]_l \parallel [Q'_S]_{l'}} \quad \begin{cases} \sigma = \bar{o}_{\vec{t},\vec{t}'}@l'(\vec{v},\vec{y}) \\ \sigma' = o'_{\vec{t}'',\vec{t}'''}@l(\vec{v},\vec{y}) \\ \bar{o}_{\vec{t},\vec{t}'} \bowtie o'_{\vec{t}'',\vec{t}'''} \end{cases}$$

(RESP-SYNC)

$$\frac{[P_S]_l \xrightarrow{\bar{o}_{\vec{t},\vec{t}'}@l'(\vec{v})} [P'_S]_l \, , \; [Q_S]_{l'} \xrightarrow{o_{\vec{t},\vec{t}'}@l(\vec{v})} [Q'_S]_{l'}}{[P_S]_l \parallel [Q_S]_{l'} \xrightarrow{\tau} [P'_S]_l \parallel [Q'_S]_{l'}}$$

(CONGRE)

$$\frac{E_1 \equiv E'_1 \, , \; E'_1 \xrightarrow{\gamma} E'_2, \; E'_2 \equiv E_2}{E_1 \xrightarrow{\gamma} E_2}$$

(PAR-EXT) (INT-EXT)

$$\frac{E_1 \xrightarrow{\gamma} E'_1}{E_1 \parallel E_2 \xrightarrow{\gamma} E'_1 \parallel E_2} \qquad \frac{P_S \xrightarrow{\gamma} P'_S}{[P_S]_l \xrightarrow{\gamma} [P'_S]_l}$$

(STRUCTURAL CONGRUENCE OVER E)

$$\frac{P \equiv_P Q}{[P, \mathcal{S}]_l \equiv [Q, \mathcal{S}]_l}$$

$$E_1 \parallel E_2 \equiv E_2 \parallel E_1 \qquad E_1 \parallel (E_2 \parallel E_3) \equiv (E_1 \parallel E_2) \parallel E_3$$

Table 4.3: Rules over E

1. $\Theta(\mathbf{0}) = \phi$
2. $\Theta(x := e) = \phi$
3. $\Theta(s) = \phi$
4. $\Theta(\bar{s}) = \phi$
5. $\Theta(\bar{o}_{\vec{t}}@l(\vec{x})) = \{(o, n, \vec{t})\}$
6. $\Theta(\bar{o}_{\vec{t},\vec{t}'}@l(\vec{x}, \vec{y})) = \{(o, sr, \vec{t}, \vec{t}')\}$
7. $\Theta(o_{\vec{t}}(\vec{x})) = \{(o, ow, \vec{t})\}$
8. $\Theta(o_{\vec{t},\vec{t}'}(\vec{x}, \vec{y}, P)) = \{(o, rr, \vec{t}, \vec{t}')\} \cup \Theta(P)$
9. $\Theta(P; P') = \Theta(P) \cup \Theta(P')$
10. $\Theta(P \mid P') = \Theta(P) \cup \Theta(P')$
11. $\Theta(\sum^{+}_{i \in W} \epsilon_i; P_i) = \bigcup_{i \in W} \Theta(\epsilon_i; P_i)$
12. $\Theta(\chi?P : Q) = \Theta(P) \cup \Theta(Q)$
13. $\Theta(\chi \rightleftharpoons P) = \Theta(P)$

It is worth noting that the interface $\Theta(P)$, during the evolution of a service $[P, \mathcal{S}]_\iota$, is monotonically reduced dependently on the consumption of P. Indeed, let us consider the following example:

$$[\bar{a}_{\vec{t}}(x), \mathcal{S}[4/x]]_\iota \parallel [a_{\vec{t}}(y), \mathcal{S}']_{\iota'} \xrightarrow{\tau}$$
$$[\mathbf{0}, \mathcal{S}[4/x]]_\iota \parallel [\mathbf{0}, \mathcal{S}'[4/y]]_{\iota'}$$

Before the synchronization the interfaces of the two services are $I_\iota = \{(a, n, \vec{t})\}$ and $I_{\iota'} = \{(a, ow, \vec{t})\}$ respectively, whereas after the synchronization they are $I_\iota = \phi$ and $I_{\iota'=}\phi$.

4.2.3 State mobility

Considering Table 4.1 and Table 4.3, such a kind of mobility is expressed by the rules which deal with operation primitives. In particular, let us consider rules NOTIFICATION and ONE-WAY of Table 4.1 in order to clarify how it works. In the former the state information $\vec{v}$ contained within the variables $\vec{x}$ are sent by exploiting a message whereas in the latter the received information $\vec{v}$ are stored into the variables $\vec{x}$ contained within the state of the receiver. Rule ONE-WAYSYNC of Table 4.3 couples the two rules by correlating the receiver location to that explicited within the notification process. In this case the message content is represented by the tuple of values $\vec{v}$. Summarizing, state mobility is modeled as an information exchange between the state of the sender and the state of the receiver. Such a mobility mechanism is the cornerstone of service-based systems and supplies the basic layer on which the other mobility mechanisms can be implemented.

4.2.4 Location mobility

So far the subpart of SOCK we have proposed, does not deal with location mobility. Locations, indeed, are statically explicited within the Notification and the Solicit-Response primitives. In order to deal with location mobility we modify the syntax by introducing the possibility to express the location as the content of a variable. To this end we add two new primitives for the Notification and the Solicit-Response where z is a variable:

$$P ::= \ldots \mid \bar{o}_{\vec{t}}@z(\vec{x}) \mid \bar{o}_{\vec{t},\vec{t}'}@z(\vec{x},\vec{y}) \mid \ldots$$

These new primitives allow us to dynamically bind the receiver location when performing the Notification and Solicit-Response operations by evaluating the content of variable z. Since locations will be acquired by means of an input operation, we introduce a new data type, we call loc, representing the type used for identifying a location. Furthermore, we introduce the set $LocVal \subseteq Val$, ranged over by l, which represents the set of all the locations. The function used to test the conformance between tuples of values and templates will be enriched by considering that, given $v \in Val$, $Type(v) = loc$ if $v \in LocVal$ (we assume that the set of values of each type is disjunct with each other).

The semantics follows:

$$\text{(NOTIFICATION WITH LOCATION MOBILITY)}\quad \frac{\vec{t} \vdash \mathcal{S}(\vec{x}),\ Type(\mathcal{S}(z)) = loc}{(\bar{o}_{\vec{t}}@z(\vec{x}), \mathcal{S}) \xrightarrow{\bar{o}_{\vec{t}}@\mathcal{S}(z)(\mathcal{S}(\vec{x}))} (\mathbf{0}, \mathcal{S})}$$

$$\frac{\vec{t} \vdash \mathcal{S}(\vec{x}),\ \mathrm{Type}(\mathcal{S}(z)) = \mathrm{loc}}{(\bar{o}_{\vec{t},\vec{t}'}@z(\vec{x},\vec{y}),\mathcal{S}) \xrightarrow{\bar{o}_{\vec{t},\vec{t}'}@\mathcal{S}(z)(\mathcal{S}(\vec{x}),\vec{y})} (o_{\vec{t},\vec{t}'}@\mathcal{S}(z)(\vec{y}),\mathcal{S})}\ \text{(Solicit with Location Mobility)}$$

Variable z is evaluated when the processes are executed. In that phase we exploit types in order to prevent the execution of bad processes: in the case z does not hold a location value, the primitive is not performed. This mechanism allows us to design a service which does not know *a priori* the locations of the services to be invoked that can be acquired during the execution.

4.2.4.1 Example

In order to clarify how location mobility works, let us consider the business scenario example depicted in Fig. 4.1 where a customer purchases a good invoking a shopping service, the shopping service invokes a bank service for performing the payment and the bank service invokes the customer for sending the invoice. In Fig. 4.1 we have exploited an informal graphical representation where services are represented by circles, the symbol $@\mathrm{uri}$ expresses the fact that the service is available at the location uri, the input operations exhibited by a service are represented by a black line whose name is shown within a rectangle and the arrows represent a message exchange. The shopping service exhibits the One-Way BUY, the Bank service exhibits the One-Way PAY and the Customer service exhibits the One-Way REC.

In the following we formalize such a scenario by supposing that the bank service does not know the location of the customer. For the sake of simplicity, we make a syntax abuse by expliciting the known locations as constants.

$\vec{t} = \langle \mathrm{loc} \rangle \quad \vec{t}' = \langle \mathrm{inf} \rangle$

$\mathrm{C} ::= [\mathrm{add} := \mathrm{uri1}; \mathrm{inv} := \bot; \overline{\mathrm{BUY}}_{\vec{t}}@\mathrm{uri2}(\mathrm{add}); \mathrm{REC}_{\vec{t}'}(\mathrm{inv}), \mathcal{S}_c]_{\mathrm{uri1}}$

$\mathrm{SH} ::= [\mathrm{fwadd} := \bot; \mathrm{BUY}_{\vec{t}}(\mathrm{fwadd}); \overline{\mathrm{PAY}}_{\vec{t}}@\mathrm{uri3}(\mathrm{fwadd}), \mathcal{S}_s]_{\mathrm{uri2}}$

$\mathrm{B} ::= [z_3 := \bot; \mathrm{invoice} := \mathrm{msg}; \mathrm{PAY}_{\vec{t}}(z_3); \overline{\mathrm{REC}}_{\vec{t}'}@z_3(\mathrm{invoice}), \mathcal{S}_b]_{\mathrm{uri3}}$

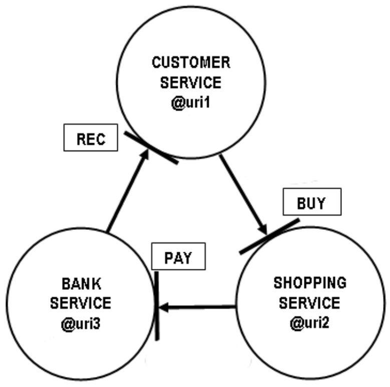

Figure 4.1: Business scenario example

$System ::= C \parallel SH \parallel B$

The shopping service SH located at $uri2$ receives on the One-Way operation BUY the location of the customer C ($uri1$) and stores it within the variable $fwadd$. Moreover, it forwards it to the bank service B (at $uri3$) by exploiting the Notification operation $\overline{\text{PAY}}$. The bank service receives on PAY the customer location and then exploits it for invoking the REC operation of the customer sending the invoice here represented by the value msg. Finally, the customer receives the invoice on REC and stores the message content within the variable inv.

4.2.4.2 Comments

Location mobility introduces a powerful mechanism for designing services in a flexible way. If we consider the Bank service of the example indeed, it exploits the operation $\overline{\text{REC}}_{\vec{t}'}@z_3(invoice)$ in order to be independent from the customer address. The Bank service can send invoices to all the customers which exhibit a One-Way whose name is REC and has a template $\vec{t}'$. On the contrary, if we do not exploit location mobility the Bank service should know the customer address before its execution binding the service to interact to a specific customer. This is the case of the shopping service that, by exploiting the operation $\overline{\text{PAY}}_{\vec{t}}@uri3(fwadd)$, is designed for sending the payment request always to the same Bank service. Furthermore, the example shows that location mobility is built on top of the state mobility because the acquired locations are stored within the state.

4.2.5 Interface mobility

Interface mobility deals with the mobility of all the information related to an operation that is the name of the operation and the templates joint to it. In general, all these information can be acquired dynamically. We model interface mobility by introducing the following primitives where z, u, k and j are variables:

$$\begin{array}{l} P ::= \ldots \mid \bar{u}_{\vec{k}}@z(\vec{x}) \mid u_{\vec{k}}(\vec{x}) \\ \mid \bar{u}_{\vec{k},\vec{j}}@z(\vec{x},\vec{y}) \mid u_{\vec{k},\vec{j}}(\vec{x},\vec{y},P) \mid \ldots \end{array}$$

The name of the operations and the templates are evaluated at run-time by reading them from the state. To this end, we introduce two new data types op and t which are used to represent the type of operation names and of templates, respectively. Let $v \in Val$, the function $Type$ is extended by defining the following cases: i) $Type(v) = \mathsf{op}$ if $v \in \mathcal{O}$, ii) $Type(v) = \mathsf{t}$ if $v \in \mathcal{T}$. We will exploit this data types to test that the values stored in the variables are in accordance with the expected data types. The semantics follows:

(Notification with Interface Mobility)

$$\frac{Type(\mathcal{S}(k)) = \mathsf{t}, Type(\mathcal{S}(u)) = \mathsf{op}, \overrightarrow{\mathcal{S}(k)} \vdash \mathcal{S}(\tilde{x}), Type(\mathcal{S}(z)) = \mathsf{loc}}{(\bar{u}_{\vec{k}}@z(\tilde{x}), \mathcal{S}) \xrightarrow{\overline{\mathcal{S}(u)}_{\overrightarrow{\mathcal{S}(k)}}@\mathcal{S}(z)(\mathcal{S}(\tilde{x}))} (\mathbf{0}, \mathcal{S})}$$

(One-Way with Interface Mobility)

$$\frac{Type(\mathcal{S}(k)) = \mathsf{t}, Type(\mathcal{S}(u)) = \mathsf{op}, \overrightarrow{\mathcal{S}(k)} \vdash \mathcal{S}(\tilde{x})}{(u_{\vec{k}}(\tilde{x}), \mathcal{S}) \xrightarrow{\overline{\mathcal{S}(u)}_{\overrightarrow{\mathcal{S}(k)}}(\mathcal{S}(\tilde{x}))} (\mathbf{0}, \mathcal{S})}$$

(SOLICIT WITH INTERFACE MOBILITY)

$$\frac{\begin{array}{l}Type(\mathcal{S}(k)) = Type(\mathcal{S}(j)) = t, Type(\mathcal{S}(u)) = op,\\ \overrightarrow{\mathcal{S}(k)} \vdash \mathcal{S}(\tilde{x}), Type(\mathcal{S}(z)) = loc\end{array}}{(\bar{u}_{\vec{k},\vec{j}}@z(\tilde{x},\tilde{y}),\mathcal{S}) \xrightarrow{\alpha} (\mathcal{S}(u)_{\overrightarrow{\mathcal{S}(k)},\overrightarrow{\mathcal{S}(j)}}@\mathcal{S}(z)(\tilde{y}),\mathcal{S})}$$

$$\alpha = \overline{\mathcal{S}(u)}_{\overrightarrow{\mathcal{S}(k)},\overrightarrow{\mathcal{S}(j)}}@\mathcal{S}(z)(\mathcal{S}(\tilde{x}),\tilde{y})$$

(REQUEST WITH INTERFACE MOBILITY)

$$\frac{Type(\mathcal{S}(k)) = Type(\mathcal{S}(j)) = t, Type(\mathcal{S}(u)) = op, \overrightarrow{\mathcal{S}(k)} \vdash \mathcal{S}(\tilde{x})}{(u_{\vec{k},\vec{j}}(\tilde{x},\tilde{y},P),\mathcal{S}) \xrightarrow{\alpha} (P;\overline{\mathcal{S}(u)}_{\overrightarrow{\mathcal{S}(k)},\overrightarrow{\mathcal{S}(j)}}@l(\tilde{y}),\mathcal{S})}$$

$$\alpha = \mathcal{S}(u)_{\overrightarrow{\mathcal{S}(k)},\overrightarrow{\mathcal{S}(j)}}@l(\mathcal{S}(\tilde{x}),\tilde{y})$$

It is worth noting that the introduction of the interface mobility allows us to distinguish the concept of operation programming from that of the information which characterize it. The former expresses the service capability to perform a One-Way, a Notification, a Request-Response or a Solicit-Response operations represented by the processes $u_{\vec{k}}(\vec{x})$, $\bar{u}_{\vec{k}}@z(\vec{x})$ $u_{\vec{k},\vec{j}}(\vec{x},\vec{y},P)$ and $\bar{u}_{\vec{k},\vec{j}}@z(\vec{x},\vec{y})$ respectively, whereas the latter deals only with the information that are necessary for performing an operation represented by the content of the variables u, k and j.

In this case the interface can change during the evolution of the service, thus we need to modify some rules of the inductive definition of Θ. To this end we first introduce the functions $tN : Val \rightarrow Val \cup \{?\}$ and $tT : Val \rightarrow Val \cup \{?\}$ for testing if the content of a variable is an operation name or a template respectively. We exploit the symbol ? for expressing the fact that an information related to an operation is unknown. The definition of the functions follows:

$$tN(v) = \begin{cases} v \text{ if } v \in \mathcal{O} \\ ? \quad otherwise \end{cases}$$

$$tT(v) = \begin{cases} v \text{ if } v \in \mathcal{T} \\ ? \quad otherwise \end{cases}$$

For the sake of brevity, we report below only the rules that change w.r.t. the original definition of Θ that are the 5, 6, 7 and 8 ones. It is worth noting that here we extend the domain of Θ by considering also the state. This is due to the fact that now the interface depends on the contents of the variables.

5. $\Theta(\bar{u}_{\vec{k}}@z(\vec{x}), \mathcal{S}) = \{tN(\mathcal{S}(u)), n, tT(\mathcal{S}(k))\}$

6. $\Theta(\bar{u}_{\vec{k},\vec{j}}@z(\vec{x},\vec{y}), \mathcal{S}) =$
   $\{tN(\mathcal{S}(u)), sr, tT(\mathcal{S}(k), tT(\mathcal{S}(j))\}$
7. $\Theta(u_{\vec{k}}(\vec{x}), \mathcal{S}) = \{tN(\mathcal{S}(u)), ow, tT(\mathcal{S}(k))\}$
8. $\Theta(u_{\vec{k},\vec{j}}(\vec{x},\vec{y},P), \mathcal{S}) =$
   $\{tN(\mathcal{S}(u)), rr, tT(\mathcal{S}(k), tT(\mathcal{S}(j))\} \cup \Theta(P)$

4.2.5.1 Example

Let us consider the example of Fig. 4.1 where we suppose that the Bank service does not know *a priori* both the location and the One-Way operation of the customer:

$\vec{t} = \langle loc, op, t\rangle \quad \vec{t}' = \langle inf\rangle$

$C ::= [add := uri1; opN := REC; opT := \vec{t}'; inv := \bot$
$\quad ; \overline{BUY}_{\vec{t}}@uri2(add, opN, opT); REC_{\vec{t}'}(inv), \mathcal{S}_c]_{uri1}$
$SH ::= [fwadd := \bot; fwopN := \bot; fwopT := \bot;$
$\quad ; BUY_{\vec{t}}(fwadd, fwopN, fwopT)$
$\quad ; \overline{PAY}_{\vec{t}}@uri3(fwadd, fwopN, fwopT), \mathcal{S}_s]_{uri2}$
$B ::= [z_3 := \bot; op := \bot; tp := \bot; invoice = msg$
$\quad ; PAY_{\vec{t}}(z_3, op, tp); \overline{op}_{\vec{tp}}@z_3(invoice), \mathcal{S}_b]_{uri3}$

$System ::= C \parallel SH \parallel B$

The customer sends, by means of the variable opN and opT, the operation name (REC) and the operation template ($\vec{t}'$) on which it will wait for receiving the invoice. The bank service receives from the shopping service the location, the name of the operation and the template of the operation of the customer and stores them in z_3, op and tp respectively.

4.2.5.2 Comments

The example shows how is possible to design a service (in this case the bank one) with a functionality which deals with an output operation without statically knowing its interface. In general, it is possible to have scenarios where a service partially knows the interface information that is, for example, it knows the name of the operation but it does not know the template or, viceversa, it knows the template but it does not know the name of the operation. In particular, the mobility of the information related only to the templates raise some interesting designing issues. A designer that does not know the template of an operation is able to program an input or an output operation but he is not able to predict the structure of the received (or sent) data and, as a consequence, he cannot exactly specify the variables related to the received (sent) data. Let us consider, for example, the output operation of the bank service:

$$\overline{op}_{\vec{tp}}@z_3(invoice)$$

in this case, even if the content of the variable tp is unkwon, there is an implicit knowledge of the template because it can be indirectly extracted by considering the variable $invoice$ which is programmed as the variable that contain the data to send. In general, a full interface mobility cannot be supported without considering a mechanism which allows a designer to formulate some kinds of predictions about the received (sent) data. We can imagine indeed, that the designer could be able to program some kinds of specifications about the variables from which it should be possible to build a sort of dynamic *adaptor* for binding the variables with the received template. The discussion and the formalization of such a kind of machinery is out of the scope of this work and, at the best of our knowledge, it is an open issue. As a first attempt towards this direction, works on component adaption can be taken into consideration. For example, in [BCPV04, BP06] Brogi et al. discuss the problem of the adaption between Web Services and WS-BPEL interfaces where adaptor specifications are discussed for composing different services with different interfaces.

4.2.6 Behaviour mobility

In order to deal with behaviour mobility we extend the subpart of the SOCK language by introducing the following primitive:

$P ::= \ldots \mid \mathtt{run}(x)$

$\mathtt{run}(x)$ allows us to execute the code contained within the variable x. As previosly done for the other kinds of mobility, we introduce a new data type representing processes. Let $proc$ be the data type denoting processes, $v \in Val$: $Type(v) = proc$ if $proc$ is defined by the term P presented in Section 4.2.2. The semantics of such a primitive is expressed by a new rule that must be added to those presented in Table **??**:

$$\text{(RUN)}\ \frac{Type(\mathcal{S}(x)) = proc}{(\mathtt{run}(x), \mathcal{S}) \xrightarrow{\tau} (\mathcal{S}(x), \mathcal{S})}$$

Since the received code can be formed by operation processes, we add a new rule for inductively defining the function Θ which allows us to extract the interface of the service:

13. $\Theta(\mathtt{run}(x), \mathcal{S}) = \begin{cases} \Theta(\mathcal{S}(x)) \ \mathit{if}\ \mathcal{S}(x) \neq \bot \\ \phi \quad \mathit{otherwise} \end{cases}$

Service functionality mobility directly deals with code mobility. In particular it allows us to design services where a specific part of its functionalities are unknown at design time and they are acquired during the execution of the service.

4.2.6.1 Example

In order to clarify how service behaviour mobility works, let us consider the example of the shopping service again where we suppose that the customer, that wants to interact with the shopping service, does not know *a priori* the conversation rules to follow. In other words, the customer does not know that it has to exhibit the REC operation in order to receive the invoice from the bank service.

$\vec{t} = \langle \mathtt{loc}, \mathtt{proc} \rangle \quad \vec{t}' = \langle \mathtt{inf} \rangle \quad \vec{t}'' = \langle \mathtt{loc} \rangle$

$$
\begin{aligned}
&\mathsf{C} ::= [\mathtt{add} := \mathtt{uri1}; \mathtt{code}_1 := \bot; \overline{\mathrm{BUY}}_{\vec{t}}@\mathtt{uri2}(\mathtt{add}, \mathtt{code}_1); \mathtt{run}(\mathtt{code}_1), \mathcal{S}_c]_{\mathtt{uri1}}\\
&\mathsf{SH} ::= [\mathtt{fwadd} := \bot; \mathtt{code}_2 := \text{``}\mathtt{inv} := \bot; \mathrm{REC}_{\vec{t}'}(\mathtt{inv})\text{''}; \mathrm{BUY}_{\vec{t}}(\mathtt{fwadd}, \mathtt{code}_2)\\
&\qquad ; \overline{\mathrm{PAY}}_{\vec{t}''}@\mathtt{uri3}(\mathtt{fwadd}), \mathcal{S}_s]_{\mathtt{uri2}}\\
&\mathsf{B} ::= [z_3 := \bot; \mathtt{invoice} = \mathtt{msg}; \mathrm{PAY}_{\vec{t}''}(z_3); \overline{\mathrm{REC}}_{\vec{t}'}@z_3(\mathtt{invoice}), \mathcal{S}_b]_{\mathtt{uri3}}
\end{aligned}
$$

$\mathsf{System} ::= \mathsf{C} \parallel \mathsf{SH} \parallel \mathsf{B}$

Here, the customer invokes the operation BUY of the shopping service which is modeled as a Request-Response operation. The customer receives as a response a piece of code and stores it within the variable $\mathtt{code}_1$, then it executes it by exploiting the primitive $\mathtt{run}(\mathtt{code}_1)$. After the execution of the code stored within $\mathtt{code}_1$ the system behaves as the example presented in the location mobility section. It is worth noting that the customer receives the input operation REC which enriches at run-time its interface similarly to the case of the interface mobility. Even if the two kind of mobility could appear similar w.r.t. the effects on the interface, they are different from a system design point of view. In the case of interface mobility the designer must specify that an input or an output operation has to be performed without knowing its name and its templates on the contrary, in the case of internal process mobility, the designer does not know the process which will be executed at all.

4.2.6.2 Comments

Some considerations about code mobility issues are necessary. On the one hand, when a service executes a process which has been acquired at run-time, it does not know how it behaves. On the other hand, when programming a process which will be executed by another service the internal behavior of such a service is not known. This fact implies a number of issues. First of all, internal processes share the variables state thus the acquired process could interfere with the behavior of the other ones. Moreover, an acquired process could exploit a certain name s to perform internal synchronizations but the same name could be already used by other internal processes, thus alterating also in this case the behavior of the other processes. A formal analysis of these issues is out of the scope of this work but we consider that, to avoid at least the issues listed above, a mechanisms which syntactically renames all the variables and names of the acquired process which interferes with the ones of the internal processes is necessary before executing it.

4.3 Mobility mechanisms in Web Service technology

This section is devoted to discuss the four kind of mobility mechanisms w.r.t. the Web Service technology. It will emerge that only the state mobility and the location one are actually implemented in such a kind of technology. Here, we hope to open a discussion about the opportunity to introduce the other kind of mobility also in the Web Service technology.

- ***State mobility***: Web Services are a technology based upon the message exchange thus, they fully support the state mobility as we have formalized it in the previous sections. In particular, an information exchange between two services is an XML document whose schema is defined within the SOAP specification.

- ***Location mobility***: As we have shown, location mobility is strictly related to the communication primitives of the service behaviour that we have formalized by exploiting SOCK. Although that, Web Services are platform independent and there is not a standard formalism for describing the service behaviour, here we consider orchestration languages as a class of languages which can be used for expressing it. Indeed, they deal with service coordination aspects which are fundamental to the end of location mobility. In particular, WS-BPEL supports location mobility by managing endpoints within its internal variables. An endpoint, which is defined within WS-Addressing [W3Cd] specification, is a data structure which contains all the information required for invoking a service, that is the operation and the location.

- ***Interface mobility***: The interface mobility, as the location one, is strictly related to the communication primitives of the service behaviour process. Following the same approach of location mobility we consider WS-BPEL. As previously mentioned, WS-BPEL is able to manage endpoints which contain the information related to the operations. However it does not support interface mobility because the operations it exploits for invoking and receiving messages are defined statically at design time and they cannot be bound at run-time. To the best of our knowledge interface mobility is not supported by the Web Services technology even if it is possible to consider other solutions that indirectly allows us to partially achieve it. Several programming languages, at a low-level w.r.t. the orchestration ones, are equipped of libraries which permit to simplify the service composition. In particular, there exist libraries in Java [Apac, Apab, Sunb] that, given a WSDL document[1], automatically produce the corresponding classes which allow for the invocation of all the operations supplied by the Web service described in that document. Such a kind of libraries allows us to partially achieve interface mobility. The interfaces indeed are not communicated as information but extracted from a WSDL document. Furthermore, they cannot be joint automatically with the service internal variables but, at the state of the art, they require to be joint by considering the presence of a human designer.

[1]A WSDL interface could be modeled by exploiting the service interface I but there are some relevant issues to take into account: a WSDL document is statically defined and cannot change dynamically during the evolution of the service by adding or removing some of the exhibited operations and, generally, Notification and Solicit-Response operations are unused

- ***Behaviour mobility***: To the best of our knowledge Web Services technology does not explicitly support such a kind of mobility. Nevertheless, we trace a comparison between service functionality mobility and some languages for describing conversational behaviours of service-based systems as, for instance, choreography languages. As we have said, such a kind of languages are exploited for describing the communication protocols services have to follow in order to participate to a given service-based system. We can imagine that a service which is willing to access that system could download the related choreography document and extracts a piece of code which allows it to follows the protocol.

4.4 Mobility mechanisms in process calculi

This section is devoted to illustrate the existing correspondances between the different kinds of Service Oriented Computing mobility mechanisms and the characteristics of some process calculi.

- **State mobility:** state mobility can be easily compared to the standard value passing proposed within CCS [Mil89] and CSP [Hoa85]. In CCS and CSP abstractions indeed, values represent the data contained within a process that are communicated by means of static input and output channels.

- **Location mobility:** location mobility can be related to the name passing mechanism of the π-calculus [MPW92]. In pi-calculus indeed, there is no distinction between channels and variables but there are only names that can be exploited both as variables and channels. Such a feature allows for the communication of channel names among processes. As for service locations, in π-calculus when a process receives a channel name, it can exploit it for sending a message over that channel.

- **Interface mobility:** at the best of our knowledge, it does not exist a process calculus that directly provides a communication mechanism which can be compared to that of interface mobility.

- **Behaviour mobility:** As we have noticed, the behaviour mobility is strongly related to code mobility. Such a kind of mechanism has been formalized by Davide Sangiorgi in [San93], where the author proposed a calculus named Higher-Order π-calculus (HOπ). In HOπ, the simple communication paradigm proposed within the π-calculus has been enriched in order to allow for the transmission of processes as well as channel names.

4.5 Discussion

The investigation about mobility mechanisms we have presented in this chapter has shown that there are four kinds of mobility mechanisms where only two of them are actually implemented in Web Services technology. They are the state mobility and the location mobility. SOCK models both of them and, in particular, it also models a hidden location mobility related to the Request-Response primitive (Section 3.5.3). The other two kinds of mobility mechanisms, the interface mobility and the service behaviour one, are not implemented. Here, we intend to open the discussion about the opportunity to consider also these kinds of mobility mechanisms within Service Oriented Compunting technologies. As we have briefly commented in the previous sections, their implementation raises a lot of technical difficulties to address. Nevertheless, as far as interface mobility is concerned, it will allow designers to be indipendent from the service interface design. Service interfaces indeed, could be received at run-time. In this way, services could be more context adaptable and their composition could be achieved easily because there are less constraints related to the interfaces. On the other hand, as far as service behaviour mobility is concerned, it could be useful in those cases where a service has to automatically join an unknown services system. In this case indeed, the service is not aware about the conversation protocol it has to fulfilll in order to interact with the other services of the system. In order to do that, it could download the code necessary to follow the system requirements and then execute it. Such a kind of issue could be also addressed by downloading a choreography of the system from which the service will self-extract the code. In our opinion, the best trade-off will exploit both choreography description and service behaviour mobility in the sense that choreography could be used for a security verification of the downloaded code. The choreography of a system indeed, could be a public document which declares how the system works. A service that wants to join the system could download the code and then perform a trusting check on the public choreography in order to verify if the downloaded code is conformant with it [2]. We believe the issues raised by this investigation deserve to be discussed within the computer science community in order to define and standardized the basic concepts the SOC paradigm is based upon. As far as our opinion is concerned, we believe both the interface mobility and the service behaviour one are useful to be considered within Service Oriented Computing paradigm and we think they deserve to be implemented within the SOC technologies.

[2]In the second part of this thesis we will investigate the issue related to the conformance between a choreography and an orchestrated system.

Chapter 5

A general model for Service Oriented Computing

The general model we propose for Service Oriented Computing follows the three-layered structure proposed for SOCK by modelling the three concepts of *service behaviour*, *service engine* and *services system* by means of formal machineries which abstract away, as much as possible, from the language details. The service behaviour deals with the representation of the behaviour of a service by means of a finite state automaton where both computational and communication capabilities are exploited, the service engine deals with the formalization of a machinery which is able to execute a service behaviour infinitely often and, finally, the services system deals with the representation of a system composed by more than one service engine by means of a language inspired by a process algebra such as CCS [Mil89] and CSP [Hoa85]. By definition, there exists a hierarchic dependency that allow for the wrapping of the service behaviour within the service engine and the service engine within the services system. All the capabilities of the service behaviour, represented by means of *service behaviour actions*, are exploited by the service engine to perfom actions at the level of the services system (*services system actions*). In this sense the service engine plays the role of connection between the service behaviour and the services system. In general, the service oriented computing design issue formalization is related to the concepts of service behaviour and service engine whereas the composition one is related to the concepts of the service engine and services system. In Fig. 5.1 we abstractly show the relationship between the service behaviour, the service engine and the services system w.r.t. the design and composition issues.

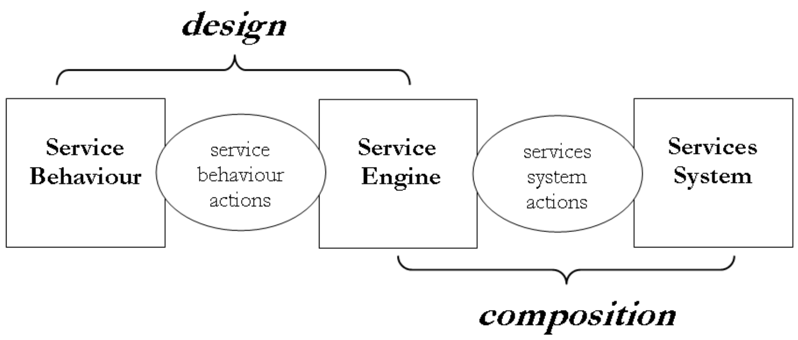

Figure 5.1: Design and composition formal framework

In the following, we formally define the sets of actions for the service behaviour and the services system. Let $\mathcal{A}_B = In_B \cup Out_B \cup Internal_B$, ranged over by $a, b, c, ...$, be the set[1] of service behaviour actions on which is defined a total order relation, where In_B is the set of the external input actions which are related to the reception of outer messages, Out_B is the set of the external output actions which are related to the sending of messages and $Internal_B$ is the set of the internal ones. Let $\mathcal{A}_S = In_S \cup Out_S \cup \{\tau\}$, ranged over by $\alpha, \beta, \gamma, ...$, be the set of services system actions where In_S is the set of input actions, Out_S is the set of the output actions and τ is the not-observable action. In the following we present the definition of service behaviour, the service engine and the services system.

5.1 Service behaviour

A service behaviour is represented by the following finite state automaton:

$$X := (\mathcal{A}_B, I, i_0, i_f, g)$$

where $\mathcal{A}_B$ is the set of service behaviour actions, I is a finite set of states, $i_0 \in I$ is the initial state, i_f is the final state and g is the transition relation $g \subseteq (I \times \mathcal{A}_B \times I)$, obviously extended to sequences of actions, that satisfies the conditions:

a) $\forall (i_0, a, i') \in g, a \in In$

[1]The set of actions could be possibly infinite.

b) $\forall a \in \mathcal{A}_B, \nexists i' \in I.(i_f, a, i') \in g$

Condition a) states that each session starts with an external input action. Condition b) states that the final state is a terminal state, i.e. there is no transition that starts from it. Let t_B be a trace of X, we exploit the following notation for denoting that the automaton X reaches the state i' starting from the initial one (i_0) by performing the sequence of actions t_B:

$$i_0 \xrightarrow{t_B} i'$$

We call *session* a trace of X after which the automaton reaches a final state. Formally, let t_B be a trace of X, we say that t_B is a session of X if:

$$i_0 \xrightarrow{t_B} i_f$$

We denote with Σ_X the set of all the sessions of X. It is worth noting that the external input actions and the external output ones are those actions which allow a service for the communication with other services. In light of this observation condition a) states that a service behaviour always starts when a message is received.

5.2 Service engine

A service engine is a service container for sessions under execution. Essentially, it is able to manage session execution by following different modalities, for example sessions can be executed in a sequential order or in a concurrent one. In particular, a service engine is always composed by a service behaviour and a formal machinery which allows for the execution of the sessions where for each action performed at the level of the service behaviour an action at the level of services system is performed. Formally, let $X := (\mathcal{A}_B, I, i_0, i_f, g)$ be a service behaviour. Let S_X be the set of all the suffixes of Σ_X ranged over by σ and let $\mathcal{M}(S_X)$ be the set of all the multisets on S_X. By definition it follows that $\Sigma_X \subseteq S_X$. Let $\sigma \circ \sigma'$ be the concatenation of the strings σ and σ'. A service engine is described by the following tuple:

$$Y := (X, \mathcal{A}_S, T, t_0, r)$$

where $\mathcal{A}_S$ is the set of services system actions, $T \subseteq \mathcal{M}(S_X)$ is a (finite or infinite) set of states where a state is represented by a multiset of suffixes of Σ_X. Each suffix within a state represent the remaining part of a session which can be executed in that state. The initial state is $t_0 \in T$ and it contains only sessions of X ($t_0 \subseteq \Sigma_X$) because at the beginning only not initiated sessions can be executed. r is a transition relation $r \subseteq (T \times \mathcal{A}_B \times \mathcal{A}_S \times T)$ which satisfies the following conditions:

a) $\forall (t, a, \alpha, t') \in r, a \in In_B \Rightarrow \alpha \in In_S$

b) $\forall (t, a, \alpha, t') \in r, a \in Out_B \Rightarrow \alpha \in Out_S$

c) $\forall (t, a, \alpha, t') \in r, a \in Internal_B \Rightarrow \alpha = \tau$

d) $\forall (t', a, \alpha, t'') \in r, \exists \sigma \in t', \exists \theta \subseteq \Sigma_X . \sigma = a \circ \sigma', t'' = t' - \{\sigma\} \uplus \{\sigma'\} - \Sigma_X \uplus \theta$

e) $\forall t \in T, \exists t' \in T, \exists a \in \mathcal{A}_B, \exists \alpha \in \mathcal{A}_S . \ (t, a, \alpha, t') \in r$

Conditions a) and b) state that external input actions and external output actions, at the level of service behaviour, must be mapped with input and output actions respectively, at the level of services system. Condition c) states that service behaviour internal actions must be mapped with a not-observable action at the level of services system. Condition d) states that, in a given state t', an action a can be performed only if it is the first action of a suffix (σ) that belongs to that state. The next state t'' must contain the suffix σ', in place of σ, because the action a has been consumed; moreover, it contains a subset (θ) of Σ_X because, in general, when an action is performed some new sessions can be enabled to be executed whereas other previously available sessions can be disabled ($-\Sigma_X \uplus \theta$). Condition e) states that a service engine is always willing to perform an action, that is, a service engine is deadlock-free. Such a feature allows us to model the fact that a service engine must be able to execute infinitely some of its sessions. Summarizing, a service engine is a machinery which infinitely often executes finite service behaviours.

5.2.1 Example.

In order to clarify these concepts, let us consider the case of a service behaviour X whose sessions are $\Sigma_X = \{abcd, efgh\}$ with $a, d, e, f \in In_B$ $b, h \in Out_B$ and $c, g \in Internal_B$, and a service engine described by the following states and transitions:

$$t_0 = \{abcd, efgh\} \;\; t_1 = \{bcd\} \;\; t_2 = \{cd\} \;\; t_3 = \{d\}$$

$$t_4 = \{fgh\} \;\; t_5 = \{gh\} \;\; t_6 = \{h\}$$

$$(t_0, a, \alpha, t_1) \;\; (t_1, b, \beta', t_2) \;\; (t_2, c, \tau, t_3) \;\; (t_3, d, \delta, t_0)$$

$$(t_0, e, \epsilon, t_4) \;\; (t_4, f, \phi, t_5) \;\; (t_5, g, \tau, t_6) \;\; (t_6, h, \kappa', t_0)$$

where $\alpha, \delta, \epsilon, \kappa \in In_S$ and $\beta', \kappa' \in Out_s$. The conditions on the transition relation r are all satisfied. In t_0 the service engine can perform an a or an e by exploiting transitions (t_0, a, α, t_1) and (t_0, e, ϵ, t_4), thus in t_0 the service engine is able to execute both sessions $abcd$ and $efgh$. In t_1 the service engine can only continue to execute the trace bcd of the initiated session $abcd$ whereas in t_4 it can continue to execute fgh. In t_3 and t_6 the service engine terminates sessions $abcd$ and $efgh$ respectively and it goes to state t_0 where it will be able to perform both sessions again.

5.3 Services system

A services system is a formal machinery which allows for the representation of a finite number of service engines that interact each others. A services system can evolve by performing two kind of actions:

- A *synchronization* between two service engines. A synchronization can be performed when an input action which belongs to the set In_S of an involved engine corresponds to an output action, which belongs to the set Out_S, of another engine.
- An *internal action*. Each service engine involved in the system can perform an internal action (τ) that is not observable to the other engines.

In order to formalize synchronizations among the service engines, we define the following relation $\mathcal{R} \subseteq (In_S \times Out_S)$ between the sets In_S and Out_S. We say that an action $\alpha \in In_S$ can synchronize itself with an action $\beta \in Out_S$ iff $\alpha \mathcal{R} \beta$. We define a services system as a process by following a CCS-like approach. Here, we report the syntax that we will exploit for representing services systems.

$$P = \mathbf{0} \mid \alpha.P \mid \; P \mid P \; \mid P + P \mid A$$

where $\mathbf{0}$ is the null process, $\alpha.P$ is the prefix, $P \mid P$ is the parallel composition of processes, $P + P$ is the choice composition of processes and A ranges over a set of constants where we suppose each constant equipped with a process definition. We consider that α ranges over the set of actions $\mathcal{A}_S$. The calculus semantics is that defined by Milner except the synchronization rule which is parameterized w.r.t. a synchorization relation $\mathcal{R}$ as it follows:

(Synchronization rule)

$$\frac{P \xrightarrow{\alpha} P', Q \xrightarrow{\beta} Q', \alpha\mathcal{R}\beta}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

In general, a services system composed by the service engines $Y_1, Y_2, ..., Y_n$ can be seen as the parallel composition of processes as it follows:

$$P_{Y_1} \mid P_{Y_2} \mid ... \mid P_{Y_n}$$

where the processes $P_{Y_1}, P_{Y_2}, ..., P_{Y_n}$ describe the service engines $Y_1, Y_2, ..., Y_n$ respectively. Given a service engine $Y = (X, \mathcal{A}_S, T, t_0, r)$ indeed, it is always possible to express it by exploiting a process. In order to do that, we express each state $t \in T$ of the service engine as a process (P_t) and then we take the process related to the initial state t_0 as the service engine process. Each state $t \in T$ can be represented by ordering[2] the suffixes it contains in the following way:

$$t = \{a_{1,1}a_{1,2}...a_{1,n_1}\,,\ a_{2,1}a_{2,2}...a_{2,n_2}\,,\ ...\ ,\ a_{m,1}a_{m,2}...a_{m,n_m}\}$$

where $a_{i,j}$ is a service behaviour action and i, j are two labels which denote the suffix and the position of the action within the suffix respectively. For example the action $a_{1,2}$ is the second action of the first suffix. Now, let t be a state of Y and let

$$(t, a_{1,1}, \alpha_1, t_1), (t, a_2, \alpha_{2,1}, t_2), ..., (t, a_m, \alpha_{m,1}, t_m)$$

[2]The ordering can be obtained by applying the Cantor dovetailing over the suffixes and their occurrences.

the transitions in r where t is the starting state, $t_1, ..., t_m$ are the target states of the transitions and $\alpha_1, ..., \alpha_m$ are the services system actions raised by each transition. Since in the state t the service engine can choose to preform the first actions of its own suffixes, we represent it by exploiting a process P_t defined in terms of an equation as it follows:

$$P_t = \alpha_1.P_{t_1} + \alpha_2.P_{t_2} + ... + \alpha_m.P_{t_m}$$

where $P_{t_1}, P_{t_2}, ..., P_{t_m}$ are the processes joined with the states $t_1, t_2, ..., t_m$ respectively. A service engine Y is represented by the process joined with its initial state t_0. Its definition equation follows:

$$P_Y := P_{t_0}$$

5.4 Starting application.

It is worth noting that, by definition, a service behaviour always starts with an external input action. It descends that a services system process cannot evolve because all the startintg actions of the involved service engines are input actions. As we have introduced in Section 2.3.3, a *starting application* is an application able to starts a services system by performing an output action which *fires* one of the involved service engines at least. Here, we assume that in a services system there exists one or more starting applications. In particular, we say that *each services system needs at least a starter application for beginning its evolution*. A starter application is not a service engine because starts with an output operation, although it can be modelled by exploiting a service engine which executes a sort of service behaviour that starts with an external output operation. Here, for the sake of brevity, we do not present a formal definition of starter application because it can be easily extracted from those of service behaviour and service engine. In the context of the services system we treat a starter application as a CCS process which starts with an output operation.

5.5 Example.

In order to clarify these concepts, let us consider a services system example where, firstly, we model the service engine presented in the previous section as a CCS process (we call it P_{t_0}) and then we introduce two other service engines (Q_{t_0} and R_{t_0}) and a starter application (SA_{t_0})[3]:

$$
\begin{array}{lll}
P_{t_0} = \alpha.P_{t_1} + \epsilon.P_{t_4} & Q_{t_0} = \rho.Q_{t_1} & R_{t_0} = \omega.R_{t_1} \\
P_{t_1} = \overline{\beta}.P_{t_2} & Q_{t_1} = \overline{\alpha}.Q_{t_2} & R_{t_1} = \overline{\epsilon}.R_{t_2} \\
P_{t_2} = \tau.P_{t_3} & Q_{t_2} = \overline{\delta}.Q_{t_0} & R_{t_2} = \overline{\phi}.R_{t_3} \\
P_{t_3} = \delta.P_{t_0} & & \\
P_{t_4} = \phi.P_{t_5} & & \\
P_{t_5} = \tau.P_{t_6} & & \\
P_{t_6} = \overline{\kappa}.P_{t_0} & &
\end{array}
$$

where the actions $\beta, \rho, \omega, \kappa$ belong to the set In_S, the actions $\overline{\alpha}, \overline{\delta}, \overline{\epsilon}, \overline{\phi}, \overline{\rho}, \overline{\omega}$ belong to the set Out_S and the synchronization relation is:

$$\mathcal{R} = \{(\alpha, \overline{\alpha}), (\beta, \overline{\beta}), ..., (\omega, \overline{\omega})\}$$

The starter application is:

$$SA_{t_0} = \overline{\rho}.\beta.\mathbf{0} \mid \overline{\omega}.\kappa.\mathbf{0}$$

The services system is modelled by the following CCS process:

$$SerSystem = P_{t_0} \mid Q_{t_0} \mid R_{t_0}$$

The evolution of such a services system, fired by the application SA_{t_0}, can be analyzed

[3]For the sake of this example, we denote the output actions by using overlined letters and we consider the usual CCS synchronization relation where not overlined letter actions synchronize themselves with the overlined ones

by considering the following CCS process:

$$\mathsf{FiredSystem} = SA_{t_0} \mid P_{t_0} \mid Q_{t_0} \mid R_{t_0}$$

Depending on the first action ($\overline{\rho}$ or $\overline{\varpi}$) performed by the process SA_{t_0} will be enabled the process Q_{t_0} or the R_{t_0} one. Now let us discuss the case where Q_{t_0} is enabled, the other case plays similarly. When Q_{t_0} is enabled, it synchronizes itself with P_{t_0} on the actions $(\alpha, \overline{\alpha})$ and P_{t_0} goes into P_{t_1}. Then, P_{t_1} synchronizes itself with the starter application on the action $(\beta, \overline{\beta})$. Finally, Q_{t_2} synchronizes itself with P_{t_2} and both the processes restarts from Q_{t_0} and P_{t_0}. Now, P_{t_0} is able to continue by synchronizing itself with R_{t_0}. The final state of the $\mathsf{FiredSystem}$ is equal to the $\mathsf{SerSystem}$ because the starter application is consumed and the service engines are all at the initial state.

Part II

Toward a new set of concrete languages for services system design: the bipolar approach

Chapter 6

Choreography

In this chapter we introduce a formal representation of choreography. Such a kind of formalization is based upon the concepts of *roles*, *conversations*, and *knowledge*. The roles abstractly represent the participants involved within a choreography that are able to interact with other participants by means of operations. The conversations allows us to express the evolution of the interactions among the roles by means of a language equipped with a formal semantics and the knowledge expresses the information known by each role that can be communicated during the execution of a choreography. In particular, roles formalization deals with the so-called *static part* of the choreography where each participant is denoted by a name and a set of operations it is able to execute. The formalization of the conversations denotes the *dinamic part* of the choreography and it is represented in terms of a process calculus where its main constructs are inspired to those of WS-CDL specification. Precisely, it provides primitives for expressing a single or a double message exchange between two roles and it provides operators for composing in sequence, parallel and non-deterministic choice the basic primitives. Finally, the knowledge formalization directly deals with the *execution part* of the choreography where knowledges are distributed among the involved roles and a semantics is given to the choreography in terms of a labelled transition system. Moreover, in the following, we introduce the definition of connected choreographies and error-free choreographies. The former ones are choreographies where for every interaction the sender role corresponds to the receiver one of the interaction that logically precedes it, whereas the latter ones are choreographies where the information are correctly communicated among the roles i.e. information are communicated only if they are known by the senders. These kinds of choreographies will be exploited in Chapter 9 where we use the bipolar approach for designing systems. In general we can say that a designer must always obtain a system choreography which is connected and error-free. Finally, we compare the choreography formal model with WS-CDL specification and we discuss an example extracted from the specification by representing it with our formal model.

6.1 Communication mechanisms

The choreography language exploits the same basic communication mechanisms of SOCK: the operations. Here, we consider the operation definition as extended in section 4.2.1 where message templates are introduced and where we consider only the location mobility mechanism. For the sake of clarity, we remind some definitions: let inf denote the type information and let loc denote the type location. Let $\mathcal{T}$, ranged over by $\vec{t}$, be the set of templates defined as arrays of type elements. For example $\vec{t'} = \langle inf, inf, inf, loc, inf, loc \rangle$ represents the structure of a message where the fourth and the sixth elements are locations and the other elements are information. Let Val, ranged over by v, be the set of values on which is defined a total order relation, $InfVal \subseteq Val$, ranged over by δ, be the set of generic information and $LocVal \subseteq Val$, ranged over by l, be the set of the locations. Let $Type$ be the function that, given $v \in Val$, returns the type of v. Since currently, we are considering only the generic information type and the location one we define:

$Type(v) = inf$ if $v \in InfVal$
$Type(v) = loc$ if $v \in LocVal$

We denote with $\vec{v} = \langle v_0, v_1, ..., v_n \rangle$ a tuple of values. Let $\vec{t} = \langle t_1, \ldots, t_n \rangle$ be a template and $\vec{v} = \langle v_1, \ldots, v_s \rangle$ be a tuple, we say that $\vec{v}$ satisfies $\vec{t}$, denoted as $\vec{t} \vdash \vec{v}$, if the following conditions hold:

1. $n = s$,

2. $\forall v_i, Type(v_i) = t_i$.

Let $\mathcal{O}$ be a set of operation names and Op be the set of operations defined as follows:

$$\begin{aligned}
&Op = \left\{(o, ow, \vec{t}) \mid o \in \mathcal{O}, \vec{t} \in \mathcal{T}\right\}\\
&\cup \left\{(o, n, \vec{t}) \mid o \in \mathcal{O},\ \vec{t} \in \mathcal{T}\right\}\\
&\cup \left\{(o, rr, \vec{t}, \vec{t'}) \mid o \in \mathcal{O},\ \vec{t}, \vec{t'} \in \mathcal{T}\right\}\\
&\cup \left\{(o, sr, \vec{t}, \vec{t'}) \mid o \in \mathcal{O},\ \vec{t}, \vec{t'} \in \mathcal{T}\right\}
\end{aligned}$$

An operation is identified by a name (o), an interaction modality (ow, n, rr and sr represent One-Way, Notification, Request-Response and Solicit-Response interaction modalities respectively) and one or two templates ($\vec{t}, \vec{t'}$) depending on the fact that the operation deals with a single message (One-Way and Notification operations) or two messages (Request-Response or Solicit-Response operations). In the former case, $\vec{t}$ represents the template of the exchanged message whereas in the latter one $\vec{t}$ represents the template of the request message and $\vec{t'}$ represents the template of the reply one. In the following we use $o_{\vec{t}}$, $\overline{o}_{\vec{t}}$, $o_{\vec{t},\vec{t'}}$ and $\overline{o}_{\vec{t},\vec{t'}}$ to range over Op for denoting the operations. We say that two operations $o_{\vec{t}}$ and $\overline{o'}_{\vec{t'}}$ are *dual* if $o = o'$ and $\vec{t} = \vec{t'}$. Analogously, we say that two operations $o_{\vec{t},\vec{t'}}$ and $\overline{o'}_{\vec{t''},\vec{t'''}}$ are *dual* if $o = o'$, $\vec{t} = \vec{t''}$ and $\vec{t'} = \vec{t'''}$. *Duality* is defined in the following way:

$$\begin{aligned}
&o_{\vec{t}} \bowtie \overline{o'}_{\vec{t'}} \Leftrightarrow o = o' \land \vec{t} = \vec{t'}\\
&o_{\vec{t},\vec{t'}} \bowtie \overline{o'}_{\vec{t''},\vec{t'''}} \Leftrightarrow o = o' \land \vec{t} = \vec{t''} \land \vec{t'} = \vec{t'''}.
\end{aligned}$$

6.2 A formal model for choreography

Choreography allows for the representation of a closed service oriented system in a top view manner. Each choreography is composed by three main parts: the *static part*, the *dynamic part* and the *execution part*.

- the static part allows for the definition of the *roles* where a role is an abstract representation of a system participant.
- the dynamic part allows for the description of the interactions evolution performed within a choreography where an interaction represents a message exchange between two roles.
- the execution part allows for the representation of the behavior of a choreography by means of a set of labelled transition systems.

6.2.1 Static part

The static part allows for the definition of the participants involved in a choreography, called *roles*, and the definition of the exchanged information.

- *Role definition*: a role is represented by a name and a set of operations. Let $RName$ be the set of the role names, ranged over by ρ and $Role$ be the set of all the possible roles defined as follows:

  $$\{(\rho, \omega) \mid \rho \in RName,\ \omega \subseteq Op\}$$

  where each role is univocally identified by its name ρ. The set ω contains both input and output operations. The input operations must be intended as the operations exhibited by the roles whereas the output ones must be considered as a knowledge which allows the role for invoking other services. In particular, a role can invoke only the operations which are dual of the output ones it knows. For example, let us consider the following roles:

  $$(A, \{a_{\vec{t_1}}, \overline{b}_{\vec{t_2}}\}) \quad (B, \{b_{\vec{t_2}}, \overline{c}_{\vec{t_2},\vec{t_3}}\}) \quad (C, \{\overline{a}_{\vec{t_1}}, c_{\vec{t_2},\vec{t_3}}\})$$

  Role A can invoke role B on operation b. Role B can invoke role C on operation c and role C can invoke role A on operation a. The fact that two roles are able to communicate through a couple of dual operations (in the following called *communication link*), does not imply that the interaction will be performed during the execution of the choreography. Interactions are designed within the dynamic part, the static part supplies only the description of the possible interactions that can be performed. Given a set of roles it is possible to represent the communication links by exploiting a graphical representation. In Fig. 6.1 is reported the graphical representation of the example above. The circles represent the roles, the black segments represent the operation exhibited by a role, a single arrow represents that a role can invoke a One-Way operation exhibited by another role by exploiting the dual notification and, analogously, a double arrow represents a Request-Response invocation.

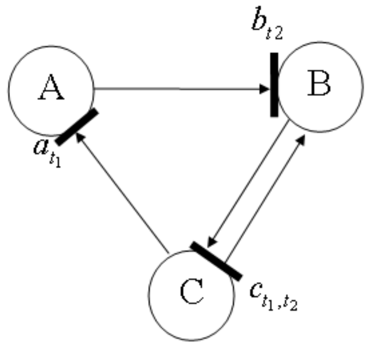

Figure 6.1: Graphical representation of communication links

- *Information definition*: in a message exchange some data are communicated from a role to another one by means of an operation. Data can be information or locations where the former represent all the application information exchanged among the roles whereas the latter represent the role locations[1]. Both information and locations are represented by two disjunct set of names. In particular, since in choreography we abstract away from physical locations, we represent locations by exploiting role names. Formally, let I_C, ranged over by i, be a set o of names which represent information and $RName$ be the set which corresponds to the set of locations and let x range over $I_C \cup RName$.

[1]The possibility to express the locations as an exchanged data differently from application information, will allow us to model the location mobility within the choreography language.

6.2.2 Dynamic part

The dynamic part allows for the design of the evolution of the choreography by means of *conversations*. Conversations are programmed by exploiting a language, called C_L, and they allows for the representation of interactions among the involved roles. We denote with $\vec{x}$ a vector of information and/or locations. The syntax of C_L follows:

$$C ::= \mathbf{0} \mid \eta \mid i :=_\rho e \mid C; C \mid C|C \mid \sum^{+}_{i \in H} \eta_i; C_i$$
$$\mid \texttt{if}\ \chi_\rho\ \texttt{then}\ C\ \texttt{else}\ C \mid \texttt{while}(\chi_\rho, C)$$

$$\eta ::= \rho_A \rightharpoonup^{\vec{x}}_{o_{\vec{t}}} \rho_B \mid \rho_A \rightleftharpoons^{\vec{x},\vec{y}}_{o_{\vec{t},\vec{t'}}} \rho_B(C) \mid \mathtt{int}(\rho)$$

$\mathbf{0}$ represents the null conversation and η represents the basic conversations. In particular, $\rho_A \rightharpoonup^{\vec{x}}_{o_{\vec{t}}} \rho_B$ is a One-Way interaction between the role ρ_A and the role ρ_B on the operation $o_{\vec{t}}$ where the exchanged message is represented by the vector $\vec{x}$ whose elements are both information and locations, $\rho_A \rightleftharpoons^{\vec{x},\vec{y}}_{o_{\vec{t},\vec{t'}}} \rho_B(C)$ represents a Request-Response interaction between role ρ_A and role ρ_B on the operation $o_{\vec{t},\vec{t'}}$ where the request message is represented by the vector $\vec{x}$ whereas the response one by the vector $\vec{y}$. The elements of the vectors are both information and locations and the conversation C within the brackets represent the conversation to be executed between the request and the response message. $\mathtt{int}(\rho)$ represents an internal action performed by the role ρ. Such a kind of primitive allows for the modelling of some internal tasks, performed by a role, whose nature is unknown to the designer. $i :=_\rho e$ represents the assignment where e denotes an expression which can contain numerical constants and information names and ρ denotes the role where the assignment is performed. The ; and | operators represent sequence and parallel composition respectively whereas $\sum^{+}_{i \in H} \eta_i; C_i$ represents the non-deterministic choice among choreographies where H is a set of indexes. It is worth noting, that all the branches of the non-deterministic choice start with an interaction η. Such an operator, indeed, allows for the representation of conversations where it is not possible to predict which interaction has to be performed in a given moment but it depends on the system run-time configuration. By definition, η can be also an internal action; such a fact implies that the non-deterministic choice can be designed by considering a race among interactions and internal actions of some involved role. In this case a role can internally choose to select a branch of the choreography discarding the other ones. `if` χ_ρ `then` C `elseif` C represents a deterministic choice where χ_ρ is a condition evaluated within the role ρ. The conditions can be expressed only on information and they are expressed by exploiting the following grammar:

$$\chi ::= i \leq e \mid e \leq i \mid \neg\chi \mid \chi \wedge \chi$$

It is worth noting that conditions such as $i = v$, $i \neq v$ and $v_1 \leq i < v_2$ can be defined as abbreviations. Finally, `while`(χ_ρ, C) allows us to express iteration where the choreography C is executed until the condition χ_ρ is true, ρ is the role where the condition is evaluated. In the following we present the well-formedness rules for conversations.

6.2.2.1 Well-formedness.

Well-formedness rules deal with the One-Way interaction and the Request-Response one. If they are satisfied, the interaction can be potentially performed because the transmitted information satisfy the operation template and both the sender and the receiver are able to exploit the operation on which the message exchange has to be performed. Let $\mathtt{length}$ be a function which returns the number of elements of an array. Given a template $\vec{t}$ and a vector $\vec{x}$ we say that the vector $\vec{x}$ satisfies the template $\vec{t}$ ($\vec{x} \vdash \vec{t}$) if :

$$\vec{x} \vdash \vec{t} \Rightarrow length(\vec{t}) = length(\vec{x}) \wedge$$
$$\forall i < length(\vec{t}), \begin{cases} \text{if } t[i] = inf \Rightarrow x[i] \in I_C \\ \text{if } t[i] = loc \Rightarrow x[i] \in RName \end{cases}$$

We say that a conversation is well formed if the following conditions hold:

a) for any interaction $\rho_A \rightharpoonup^{\vec{x}}_{o_{\vec{t}}} \rho_B$

- $\vec{x} \vdash \vec{t}$

- $o_{\vec{t}} \in \omega_B$
- $\overline{o}_{\vec{t}} \in \omega_A$

b) for any interaction $\rho_A \rightleftharpoons^{\vec{x},\vec{y}}_{o_{\vec{t},\vec{t'}}} \rho_B(C)$

- $\vec{x} \vdash \vec{t}, \vec{y} \vdash \vec{t'}$
- $o_{\vec{t},\vec{t'}} \in \omega_B$
- $\overline{o}_{\vec{t},\vec{t'}} \in \omega_A$

where ω_A and ω_B represent the sets of the operations of the roles ρ_A and ρ_B respectively. The conditions state that:

- within an interaction the exchanged message has to satisfy the operation template
- the operation on which the interaction is performed has to belong to the receiver and the dual one has to belong to the sender.

6.2.3 Choreography execution

Choreography execution deals with the semantic representation of a choreography. The semantics is given in terms of a set of *labelled transition systems* [Kel76] whose nodes are tuples of three elements $(C, \mathcal{K}, \gamma)$: the conversation (C), the *knowledge* ($\mathcal{K}$) and the *state* (γ). The former is expressed by exploiting the language C_L and represents the actual conversation to perform, the knowledge supplies a formal representation of the information and the locations known by each role, and the state represents the actual values of each information exploited within the choreography. Each choreography is equipped with a logic condition on the information called *initial constraints* which allows us to bound the range of the values of each information. For each configuration of information values which satisfies the initial constraints, a labelled transition system is generated starting from a tuple $(C_0, \mathcal{K}_0, \gamma_0)$ where C_0 represents the initial conversation, $\mathcal{K}_0$ represents the initial knowledge and γ_0 is the initial state coherent with the given configuration of information values.

6.2.3.1 The knowledge.

The knowledge for a choreography represents:

- the set of all the information known by each involved role
- the set of all the locations known by each involved role

Formally, the knowledge is defined as follows:

$$\mathcal{K} = (I, \Lambda)$$

where I and Λ are two functions:

1. $I : RName \rightarrow \mathbf{P}(I_C)$
2. $\Lambda : RName \rightarrow \mathbf{P}(RName)$

the function I is called *information function* and joins each role to a set of known information, the function Λ is called *location function* and represents all the locations known by each role. In the following, given a distribution of knowledge $\mathcal{K}$, we exploit the notation $\mathcal{K}^I$ for denoting the function I and $\mathcal{K}^\Lambda$ for denoting the function Λ. The update operators for a knowledge $\mathcal{K}$ is defined as follows:

$$\mathcal{K}[\vec{x}/\rho] = \mathcal{K}' \qquad \forall i \leq length(\vec{x}), \begin{cases} \mathcal{K}^{I'}(\rho) = \mathcal{K}^I(\rho) \cup x_i \text{ if } x_i \in I_C \\ \mathcal{K}^{I'}(\rho') = \mathcal{K}^I(\rho') \\ \mathcal{K}^{\Lambda'}(\rho) = \mathcal{K}^\Lambda(\rho) \cup x_i \text{ if } x_i \in RName \\ \mathcal{K}^{\Lambda'}(\rho') = \mathcal{K}^\Lambda(\rho') \end{cases}$$

The operator $\mathcal{K}[\vec{x}/\rho]$ allows for the updating of the knowledge of a role ρ with the information and the locations contained within the vector $\vec{x}$. If the element x_i is an information, the function I will be updated by adding it to the known information of ρ and, if the element x_i is a location, the function Λ will be updated by adding it to the known locations of ρ. Furthermore, we exploit the following notation for representing the information and the locations known by a role:

$$\mathcal{K}(\rho) = I(\rho) \cup \Lambda(\rho)$$

6.2.3.2 The state.

The state of a choreography is represented by the following function which holds the actual values of the infotmation used within the choreography:

$$\gamma : I_C \rightarrow InfVal$$

The update operator for γ is defined as follows:

$$\gamma[\delta/i] = \gamma' \qquad \gamma(j) = \begin{cases} \gamma'(j) = \gamma(j) & \text{if } j \neq i \\ \gamma'(j) = \delta & \text{if } j = i \end{cases}$$

Let x be a vector of information and/or locations, we exploit the following notation for expressing its actual content:

$\dot{x} = \gamma(x)$ if $x \in I_C$
$\dot{x} = x$ if $x \in RName$

We extend such a notation for a vector of elements as follows:

$\vec{x} = \langle x_1, x_2, ..., x_n \rangle \Rightarrow \dot{\vec{x}} = \langle \dot{x_1}, \dot{x_2}, ..., \dot{x_n} \rangle$

Furthermore, we exploit the notation $\gamma \vdash \chi$ for denoting that the function γ satisfies the logic condition χ. The satisfaction relation for $\vdash$ is defined by the following rules:

1. $e \hookrightarrow_\gamma \delta, \gamma(i) \leq \delta \Rightarrow \gamma \vdash i \leq e$
2. $e \hookrightarrow_\gamma \delta, \delta \leq \gamma(i) \Rightarrow \gamma \vdash e \leq i$
3. $\gamma \vdash \chi' \wedge \gamma \vdash \chi'' \Rightarrow \gamma \vdash \chi' \wedge \chi''$
4. $\neg(\gamma \vdash \chi) \Rightarrow \gamma \vdash \neg\chi$

where $e \hookrightarrow_\gamma \delta$ means that the expression e is evaluated into δ under the state γ.

6.2.3.3 The labelled transition system.

The labelled transition system of a choreography is defined by exploiting the semantic rules of C_L which describe the evolution of a conversation joined with a knowledge and a state. Let $Act_C = \{\mu \mid \mu = (\rho_A, \rho_B, o_t, \vec{x})\} \cup \{\mu \mid \mu = (\rho_A, \rho_B, o_{t,t'}, \vec{x}, dir)\} \cup \{\tau, \pi(\rho)\}$ be the set of actions ranged over by ν where μ represents parameterized interactions and $dir \in \{\uparrow, \downarrow\}$ is the set whose elements $\uparrow, \downarrow$ describe the direction of a message exchange in a Request-Response interaction: $\uparrow$ is the request message and $\downarrow$ is the response one. τ represents an unobservable action whereas $\pi(\rho)$ represents an internal action performed by a role ρ. It is worth noting that, to the end of the conformance we will present in Chapter 8, the action $\pi(\rho)$ will be observable even if, considering the language C_L, it models an action performed by a role whose nature is unknown for the designer. $(C, \mathcal{K}, \gamma) \xrightarrow{\nu} (C', \mathcal{K}', \gamma')$ means that the conversation C with the knowledge $\mathcal{K}$ and the state γ, evolves in one step in a configuration $(C', \mathcal{K}', \gamma')$ performing the action ν. Let K_C be the set of all possible knowledge couples and let Γ_C be the set of all the possible states. We define $\rightarrow \subseteq (C_L, K_C, \Gamma_C) \times Act_C \times (C_L, K_C, \Gamma_C)$ as the least relation which satisfies the axioms and rules of Table 6.1 and closed w.r.t. $\equiv$, where $\equiv$ is the least congruence relation satisfying the axioms at the end of Table 6.1. The structural congruence $\equiv$, which equates the conversations whose behaviour cannot be distinguished, expresses that $(C, \mid)$ is an abelian monoid where $\mathbf{0}$ is the null element. Furthermore, the rule $\mathbf{0}; C \equiv C$ means that when a conversation completes then the other one which follows in sequence can be performed. It is worth noting that, in order to model the Request-Response message exchange, we have introduced the syntactic term: $\rho_A \xleftharpoondown[o_{\vec{t},\vec{t}'}]{\vec{x}} \rho_B$ which represents a response message interaction.

The description of axioms and rules follows. The axiom ONE-WAY describes the evolution of a One-Way interaction. It is worth noting that the interaction is performed in an atomic way: the message is sent and received and the knowledge is updated adding the message $\vec{x}$ to the knowledge of the receiver role ρ_B, the label μ reports the actual values of the vector $\vec{x}$ which will be fundamental to the end of conformance. Axioms REQUEST and RESPONSE deal with the Request-Response interaction pattern. The former one deals with the request interaction whereas the latter one deals with the response

(One-Way)

$$(\rho_A \rightharpoonup^{\vec{x}}_{o_{\vec{t}}} \rho_B, \mathcal{K}, \gamma) \xrightarrow{\mu} (\mathbf{0}, \mathcal{K}[\vec{x}/\rho_B], \gamma), \mu = (\rho_A, \rho_B, o_{\vec{t}}, \dot{\vec{x}})$$

(Request)

$$(\rho_A \rightleftharpoons^{\vec{x},\vec{y}}_{o_{\vec{t},\vec{t}'}} \rho_B(C), \mathcal{K}, \gamma) \xrightarrow{\mu} (C; \rho_A \leftharpoondown^{\vec{y}}_{o_{\vec{t},\vec{t}'}} \rho_B, \mathcal{K}[\vec{x}/\rho_B], \gamma), \mu = (\rho_A, \rho_B, o_{\vec{t},\vec{t}'}, \dot{\vec{x}}, \uparrow)$$

(Response)

$$(\rho_A \leftharpoondown^{\vec{x}}_{o_{\vec{t},\vec{t}'}} \rho_B, \mathcal{K}, \gamma) \xrightarrow{\mu} (\mathbf{0}, \mathcal{K}[\vec{x}/\rho_A], \gamma), \mu = (\rho_A, \rho_B, o_{\vec{t},\vec{t}'}, \dot{\vec{x}}, \downarrow)$$

(Internal)

$$(\mathtt{int}(\rho), \mathcal{K}, \gamma) \xrightarrow{\pi(\rho)} (\mathbf{0}, \mathcal{K}, \gamma)$$

(Assign)

$$\frac{e \hookrightarrow_\gamma \delta}{(i :=_\rho e, \mathcal{K}, \gamma) \xrightarrow{\tau} (\mathbf{0}, \mathcal{K}, \gamma[\delta/i])}$$

(Sequence)

$$\frac{(C, \mathcal{K}, \gamma) \xrightarrow{\nu} (C', \mathcal{K}', \gamma')}{(C; D, \mathcal{K}, \gamma) \xrightarrow{\nu} (C'; D, \mathcal{K}', \gamma')}$$

(Parallel)

$$\frac{(C, \mathcal{K}, \gamma) \xrightarrow{\nu} (C', \mathcal{K}', \gamma')}{(C \mid D, \mathcal{K}, \gamma) \xrightarrow{\nu} (C' \mid D, \mathcal{K}', \gamma')}$$

(Congr)

$$\frac{C' \equiv C, (C, \mathcal{K}, \gamma) \xrightarrow{\nu} (D, \mathcal{K}', \gamma'), D \equiv D'}{(C', \mathcal{K}, \gamma) \xrightarrow{\nu} (D', \mathcal{K}', \gamma')}$$

(Choice)

$$\frac{(\eta_i, \mathcal{K}, \gamma) \xrightarrow{\nu} (D, \mathcal{K}', \gamma'), i \in H}{(\sum^{+}_{i \in H} \eta_i; C_i, \mathcal{K}, \gamma) \xrightarrow{\nu} (D; C_i', \mathcal{K}', \gamma')}$$

(If then)

$$\frac{\gamma \vdash \chi}{(\mathtt{if}\ \chi_\rho\ \mathtt{then}\ C\ \mathtt{else}\ D, \mathcal{K}, \gamma) \xrightarrow{\tau} (C, \mathcal{K}', \gamma')}$$

(Else)

$$\frac{\gamma \nvdash \chi}{(\mathtt{if}\ \chi_\rho\ \mathtt{then}\ C\ \mathtt{else}\ D, \mathcal{K}, \gamma) \xrightarrow{\tau} (D, \mathcal{K}', \gamma')}$$

(While 1)

$$\frac{\gamma \vdash \chi}{(\mathtt{while}(\chi_\rho, C), \mathcal{K}, \gamma) \xrightarrow{\tau} (C; \mathtt{while}(\chi_\rho, C), \mathcal{K}', \gamma')}$$

(While 2)

$$\frac{\gamma \nvdash \chi}{(\mathtt{while}(\chi_\rho, C), \mathcal{K}, \gamma) \xrightarrow{\tau} (\mathbf{0}, \mathcal{K}, \gamma)}$$

(Structural Congruenge)

$$\mathbf{0}; C \equiv C \qquad C \mid \mathbf{0} \equiv C$$

$$C \mid D \equiv D \mid C \qquad (C \mid D) \mid F \equiv C \mid (D \mid F)$$

Table 6.1: Semantics rules for choreography execution

interaction where the syntactic element $\rho_A \leftharpoondown^{\vec{y}}_{o_{\vec{t},\vec{t}'}} \rho_B$ describes that a response interaction must be performed. The labels μ of the request interaction and the response one differs for the direction represented by the arrows: the arrow $\uparrow$ represents the request and the arrow $\downarrow$ represents the response. Rule ASSIGN describes how the state is updated when an assignment is performed[2]. Rules SEQUENCE, PARALLEL and CONGR are standard. Rules IF THEN and ELSE deal with deterministic choice whereas rules WHILE 1 and WHILE 2 deal with the while construct. Rule CHOICE deals with the non-deterministic choice where the construct $\eta_i; C_i$ guarantees that each branch of the choice always starts with an interaction between two roles or an internal action.

6.2.3.4 Initial constraints.

The initial constraints, denoted with X, are expressed in terms of a logic condition on the information which follows the rules of the χ condition presented in Section 6.2.3.1. For each initial configuration of the information values which satisfies the initial constraints X an initial state γ will be generated. For example let us consider a choreography where the information named $apple$ and $banana$ are defined. The initial constraints could be defined as follows:

$$X := apple > 0 \land apple < 3 \land banana >= 0 \land banana < 2$$

Four different initial states will be generated:

$\gamma_0[1/apple, 0/banana]$
$\gamma_1[2/apple, 0/banana]$
$\gamma_2[1/apple, 1/banana]$
$\gamma_3[2/apple, 1/banana]$

[2]The role ρ of the assign primitive $i :=_\rho e$ will be exploited in the following for defining the error-free choreographies.

It is worth noting that the initial constraints are considered only for the generation of the initial states and they will be ignored during the different executions of the choreography. Given an initial constraint X we define the set of all the possible initial states as follows:

$$\gamma_X := \{\gamma_i \mid \gamma_i \vdash X\}$$

where, with the term $\gamma_i \vdash X$ we denote a state γ_i which satisfies the initial constraints defined in X.

6.2.4 The choreography

Now we are ready to define a choreography. A choreography, denoted by $\mathcal{C}$, is defined by the tuple $(\Sigma, C_0, \mathcal{K}_0, X)$ where $\Sigma \subseteq \mathtt{Role}$ is a finite set containing the involved roles, $C_0 \in C_L$, $\mathcal{K}_0$ is the initial knowledge and X is a logic condition which expresses the variables constraints of the initial state. The execution of a choreography $\mathcal{C}$ is expressed by the set of all the labelled transition systems generated starting from a tuple $(C_0, \mathcal{K}_0, \gamma_i)$ where γ_i ranges over γ_X.

6.3 Connected choreographies.

A choreography is connected when its conversation is connected. Intuitively, a conversation is connected when for every interaction the sender role corresponds to the receiver one of the interaction that logically precedes it. For example, let us consider the following not-connected conversation:

$$\rho_A \rightharpoonup^{x}_{o_{\tau}} \rho_B ; \rho_C \rightharpoonup^{y}_{o'_{\tau'}} \rho_D$$

In this example, the first interaction is performed when the role ρ_B receives the message whereas the following one performs a message exchange from ρ_C to ρ_D. The conversation is not connected because there are any logical connections between the first interaction and the second one. Indeed, the first one finishes on role ρ_B whereas the latter one starts from role ρ_C which is the sender role of the second interaction and ρ_C cannot be aware of the termination of the first interaction. In order to be connected, the conversation could be modified as follows by adding a new interaction:

$$\rho_A \rightharpoonup^{x}_{o_{\vec{t}}} \rho_B; \rho_B \rightharpoonup^{z}_{o''_{\vec{t}''}} \rho_C; \rho_C \rightharpoonup^{y}_{o'_{\vec{t}'}} \rho_D$$

Such a conversation is connected because the first interaction finishes on role ρ_B where starts the second one. Moreover, the second one finishes on role ρ_C where starts the third one. The connection issue in the choreography setting has been proposed by Honda et al. in [MCY] where the authors introduce such a definition in order to limit the set of the accepted choreographies to those that are connected. Here, we introduce connected choreographies in order to deal with system design without limiting the accepted choreographies to the accepted ones. Such an aspect will be explained in Chapter 9 where we present a design system example which exploits a not-connected choreography.
Before defining connected conversations we introduce the sets Final and Init which allows us to define, for any syntactic element of a conversation, the set of the roles where it finishes and begins respectively. The set Final is inductively defined as it follows:

1. $\mathit{Final}(\rho_A \rightharpoonup^{\vec{x}}_{o_{\vec{t}}} \rho_B) = \{\rho_B\}$
2. $\mathit{Final}(\rho_A \rightleftharpoons^{\vec{x},\vec{y}}_{o_{\vec{t},\vec{t}'}} \rho_B(C)) = \{\rho_A\}$
3. $\mathit{Final}(\mathit{int}(\rho)) = \{\rho\}$
4. $\mathit{Final}(i :=_{\rho} e) = \{\rho\}$
5. $\mathit{Final}(C; C') = \mathit{Final}(C')$
6. $\mathit{Final}(C \mid C') = \mathit{Final}(C) \cup \mathit{Final}(C')$
7. $\mathit{Final}(\sum^{+}_{i \in H} \eta_i; C_i) = \bigcup_{i \in H} \mathit{Final}(C_i)$
8. $\mathit{Final}(\texttt{if}\ \chi_{\rho}\ \texttt{then}\ C\ \texttt{else}\ C') = \mathit{Final}(C) \cup \mathit{Final}(C')$

9. $\mathtt{Final}(\mathtt{while}(\chi_\rho, C)) = \mathtt{Final}(C) \cup \{\rho\}$

In general the set $\mathtt{Final}$, given a conversation, contains all the roles on which the conversation ends. For instance, let us consider rule 1 which deals with the One-Way interaction: in this case the set $\mathtt{Final}$ contains the receiver role ρ_B because the interaction is completed only when the message is received by role ρ_B. On the contrary, in rule 2, where the Request-Response interaction is taken into account, the set $\mathtt{Final}$ contains the sender role ρ_A because all the interaction is completed only when the response message is received by the sender role. Rule 3 and 4 deal with the internal action and the assignment respectively; in both cases the set $\mathtt{Final}$ contains the role where the conversation is performed. In rule 5 the sequence composion of two conversation is taken into account; it is worth noting that, obviously, only the set $\mathtt{Final}$ of the latter conversation takes relevance. Rule 6 deals with the parallel composition where the set $\mathtt{Final}$ is obtained as the union of the set $\mathtt{Final}$ of the two conversations; this is due to the fact that a parallel conversation can be considered completed when both the conversations are completed. Rules 7, 8 and 9 deal with the non-deterministic choice, the *if_then_else* and the while constructs respectively. It is worth noting that in while construct the set $\mathtt{Final}$ is represented by the union of the set $\mathtt{Final}$ of the body conversation C, and the role ρ where the condition is evaluated; this is due to the fact that the condition could be immediately evaluated to false without never executing the body C. In the following we present the inductive definition of the set $\mathtt{Init}$:

1. $\mathtt{Init}(\rho_A \overset{\vec{x}}{\underset{o_{\vec{t}}}{\rightharpoonup}} \rho_B) = \{\rho_A\}$

2. $\mathtt{Init}(\rho_A \overset{\vec{x},\vec{y}}{\underset{o_{\vec{t},\vec{t}'}}{\rightleftharpoons}} \rho_B(C)) = \{\rho_A\}$

3. $\mathtt{Init}(\mathtt{int}(\rho)) = \{\rho\}$

4. $\mathtt{Init}(i :=_\rho e) = \{\rho\}$

5. $\mathtt{Init}(C; C') = \mathtt{Init}(C)$

6. $\mathtt{Init}(C \mid C') = \mathtt{Init}(C) \cup \mathtt{Init}(C')$

7. $\mathtt{Init}(\sum^{+}_{i\in H}\eta_i; C_i) = \bigcup_{i\in H}\mathtt{Rec}(\eta_i), \begin{cases} \mathtt{Rec}(\rho_A \rightharpoonup^{\vec{x}}_{o_{\vec{t}}} \rho_B) = \mathtt{Final}(\rho_A \rightharpoonup^{\vec{x}}_{o_{\vec{t}}} \rho_B) \\ \mathtt{Rec}(\rho_A \rightleftharpoons^{\vec{x},\vec{y}}_{o_{\vec{t},\vec{t'}}} \rho_B(C)) = \{\rho_B\} \\ \mathtt{Rec}(\mathtt{int}(\rho)) = \mathtt{Final}(\mathtt{int}(\rho)) \end{cases}$

8. $\mathtt{Init}(\texttt{if}\ \chi_\rho\ \texttt{then}\ C\ \texttt{else}\ C') = \{\rho\}$

9. $\mathtt{Init}(\texttt{while}(\chi_\rho, C)) = \{\rho\}$

In general, the set $\mathtt{Init}$ contains all the roles which initiate a conversation. The rules for generating the set $\mathtt{Init}$ follow the same approach exploited for the set $\mathtt{Final}$ generation with the exception that conversation initial roles are considered. In particular, the set $\mathtt{Init}$ of the non-deterministic choice, defined in rule 7, deserves to be commented. By definition a non deterministic choice always starts with a conversation η_i that can be an interaction or an internal action. Since the interactions are always performed in an atomic way, it descends that only the receiver is able to non-deterministically select the branch to execute. In other words, the choice is always performed by the receiver roles of the η_i interactions. In order to be clear as much as possible, let us consider the following conversation where there is a non-deterministic choice between two conversations:

$$\rho_A \rightharpoonup^{\vec{x}}_{o_{\vec{t}}} \rho_B; C + \rho_C \rightharpoonup^{\vec{y}}_{o'_{\vec{t}'}} \rho_B; C'$$

depending on the first interaction received (from ρ_A or from ρ_C) the conversations C or C' are executed. The choice is performed within the role ρ_B that is the receiver of both interactions and can evaluate which message is firstly received. In light of these observations, the set $\mathtt{Init}$ of the non-deterministic choice corresponds to the union of the receiver roles of the interactions η_i. Now, we inductively define the following function $\mathtt{Conn} : C_L \rightharpoonup \mathtt{Bool}$ which allows us to state if a conversation is connected or not where with the notations $\#\mathtt{Final}(C)$ and $\#\mathtt{Init}(C)$ we denote the cardinality of the sets $\mathtt{Final}$ and $\mathtt{Init}$ respectively.

1. $\mathtt{Conn}(\rho_A \rightharpoonup^{\vec{x}}_{o_{\vec{t}}} \rho_B) = \mathtt{true}$

2. $\mathtt{Conn}(\mathtt{int}(\rho)) = \mathtt{true}$

3. $\mathit{Conn}(i :=_\rho e) = \mathit{true}$

4. $\mathit{Conn}(\rho_A \rightleftharpoons^{\vec{x},\vec{y}}_{o_{\vec{t},\vec{t}'}} \rho_B(C)) = \begin{cases} \mathit{true} & \text{if } \mathit{Init}(C) = \mathit{Final}(C) = \{\rho_B\} \wedge \\ & \mathit{Conn}(C) = \mathit{true} \\ \mathit{false} & \mathit{otherwise} \end{cases}$

5. $\mathit{Conn}(C; C') = \begin{cases} \mathit{true} & \text{if } \mathit{Final}(C) = \mathit{Init}(C') \wedge \\ & \#\mathit{Final}(C) = \#\mathit{Init}(C') = 1 \wedge \\ & \mathit{Conn}(C) = \mathit{Conn}(C') = \mathit{true} \\ \mathit{false} & \mathit{otherwise} \end{cases}$

6. $\mathit{Conn}(C \mid C') = \begin{cases} \mathit{true} & \text{if } \mathit{Conn}(C) = \mathit{Conn}(C') = \mathit{true} \\ \mathit{false} & \mathit{otherwise} \end{cases}$

7. $\mathit{Conn}(\sum^{+}_{i \in H} \eta_i; C_i) = \begin{cases} \mathit{true} & \text{if } \#\mathit{Init}(\sum^{+}_{i \in H} \eta_i; C_i) = 1 \wedge \\ & \forall i \in H, \mathit{Conn}(\eta_i; C_i) = \mathit{true} \\ \mathit{false} & \mathit{otherwise} \end{cases}$

8. $\mathit{Conn}(\texttt{if}\ \chi_\rho\ \texttt{then}\ C\ \texttt{else}\ C') = \begin{cases} \mathit{true} & \text{if } \mathit{Init}(C) = \mathit{Init}(C') = \{\rho\} \\ \mathit{false} & \mathit{otherwise} \end{cases}$

9. $\mathit{Conn}(\texttt{while}(\chi_\rho, C)) = \begin{cases} \mathit{true} & \text{if } \mathit{Final}(C) = \mathit{Init}(C) = \{\rho\} \\ \mathit{false} & \mathit{otherwise} \end{cases}$

One-Way interactions, internal actions and assignment are always connected whereas a Request-Response is connected if the conversation C performed within the request and the reply always starts and finishes with the receiver role ρ_B. A sequence between two conversations, C and C', is connected only if the conversation C finishes in the same role where the C' starts and the parallel composition of two conversations is connected if the two components are connected. The non-deterministic choice is connected only if the set Init contains only one role. The `if then else` construct is connected if the conversation C and C' has a set Init that corresponds to the role on which the condition χ is evaluated. In the same way the `while` primitive is connected if the role on which the condition is evaluated is the same of that contained within the set Init of the body conversation C, it is worth noting that also the set Final of the conversation C has to correspond to the role on which the condition is evaluated. In the following we exploit the function Conn for defining connected choreographies.

Definition 6.1 (Connected choreographies) *Let $\mathcal{C}$ be a choreography where C is its conversation. We say that $\mathcal{C}$ is a connected choreography if* $\mathsf{Conn}(C) = \mathsf{true}$*.*

6.4 Error-free choreographies

The knowledge allows for the representation of the information flow among the roles. The One-Way interactions and the Request-Response one allows for the communication of the information from a role to another one. Since conversations are programmed indipendently from the knowledge, it is possible to design choreographies where some interactions try to perform an information exchange (or a location exchange) even if the information does not belong to the sender knowledge. Moreover, it is possible to design conversations where the sender role try to send a message to the receiver one without knowing its location. In this cases, we say that the choreography has an *error* because it allows for the communication of some information that cannot be communicated. For example, let us consider the following knowledge and conversation:

$$
\begin{aligned}
&\mathcal{K}^{I}(\rho_A) = \{\mathsf{apple}\}\\
&\mathcal{K}^{I}(\rho_B) = \{\mathsf{banana}\}\\
&\mathcal{K}^{I}(\rho_C) = \emptyset\\
&\mathcal{K}^{\Lambda}(\rho_A) = \{\rho_B\}\\
&\mathcal{K}^{\Lambda}(\rho_B) = \mathcal{K}^{\Lambda}(\rho_C) = \emptyset
\end{aligned}
$$

$$\rho_A \overset{\mathsf{apple}}{\underset{o_{\vec{t}}}{\rightharpoonup}} \rho_B ; \rho_B \overset{\mathsf{nuts}}{\underset{o'_{\vec{t}'}}{\rightharpoonup}} \rho_C$$

where $I_C = \{\mathsf{apple}, \mathsf{nuts}, \mathsf{banana}\}$. The first interaction, from role ρ_A to the role ρ_B, can be performed because role ρ_A knows the information apple and it knows also the location of role ρ_B ($\mathcal{K}^{\wedge}(\rho_A) = \{\rho_B\}$), but the second interaction from role ρ_B to role ρ_C cannot be performed for two reasons: on the one hand, ρ_B does not know the location of ρ_C (the set $\mathcal{K}^{\wedge}(\rho_B)$ is empty) and, on the other hand, the information $nuts$ does not belong to the knowledge of the role ρ_B. Since we intend to manage choreographies where the information can be sent only if they are known by the sender and the interaction can be performed if the sender knows the location of the receiver, here we introduce the defintion of the error-free choreography. An error-free choreography is a choreography where, for each performed interaction, the communicated information and locations are always known by the sender. In order to define it, we extend the semantics rules of Table 6.1 with the rules of Table 6.2 where we consider the set of actions extended with the action err. It is worth noting that rule RESPONSE does not deal with the condition on the location knowledge of the sender, that is the role ρ_B. This is due to the fact that we consider a hidden location mobility within a Request-Response interaction, as we have discussed within Section 3.5.3, which implies that the response sender is not aware of the location of the response receiver because it is managed at the level of the Request-Response primitive. Rule ASSIGN deserves to be commented:

- the assigned information (i) has to be known only by the declared role ρ. An assignment on an information shared by two or more roles indeed, would allow all of them to access the new value immediately by implicitly representing a system where such a new value is communicated among all the roles without any message exchange.
- all the information contained within the expression e (represented by the symbol i_e) has to be known by the role ρ. In this case indeed, all the information needed for computing the expression e must be available at the same role ρ where the information i to be assigned is known.

Rules IF THEN ELSE and WHILE specify that a condition can be evaluated only if all the involved information belong to the specified role ρ. With abuse of the notation $i_e \in \chi$ for representing an information i_e that is contained within the logic condition χ. The definition of the error-free choreography follows:

(ONE-WAY)

$$\frac{\forall x_i \in \vec{x}, \exists x_i \notin \mathcal{K}(\rho_A) \wedge \rho_B \in \mathcal{K}^{\wedge}(\rho_A)}{(\rho_A \rightharpoonup^{\vec{x}}_{o_{\vec{t}}} \rho_B, \mathcal{K}, \gamma) \overset{err}{\rightarrow} (\mathbf{0}, \mathcal{K}, \gamma)}$$

(REQUEST)

$$\frac{\forall x_i \in \vec{x}, \exists x_i \notin \mathcal{K}(\rho_A) \wedge \rho_B \in \mathcal{K}^{\wedge}(\rho_A)}{(\rho_A \rightleftharpoons^{\vec{x},\vec{y}}_{o_{\vec{t},\vec{t'}}} \rho_B(C), \mathcal{K}, \gamma) \overset{err}{\rightarrow} (\mathbf{0}, \mathcal{K}, \gamma)}$$

(RESPONSE)

$$\frac{\forall x_i \in \vec{x}, \exists x_i \notin \mathcal{K}(\rho_B)}{(\rho_A \leftharpoondown^{\vec{x}}_{o_{\vec{t},\vec{t'}}} \rho_B, \mathcal{K}, \gamma) \overset{err}{\rightarrow} (\mathbf{0}, \mathcal{K}, \gamma)}$$

(ASSIGN)

$$\frac{\mathtt{i} \notin \mathcal{K}(\rho) \vee (\exists \rho' \in \mathtt{Role}, \mathtt{i} \in \mathcal{K}(\rho') \vee (\exists i_e \in e, i_e \notin \mathcal{K}(\rho))}{(\mathtt{i} :=_\rho e, \mathcal{K}, \gamma) \overset{err}{\rightarrow} (\mathbf{0}, \mathcal{K}, \gamma)}$$

(IF THEN ELSE)

$$\frac{\exists i_e \in \chi, i_e \notin \mathcal{K}(\rho)}{(\mathtt{if}\ \chi_\rho\ \mathtt{then}\ C\ \mathtt{else}\ D, \mathcal{K}, \gamma) \overset{err}{\rightarrow} (\mathbf{0}, \mathcal{K}', \gamma')}$$

(WHILE)

$$\frac{\exists i_e \in \chi, i_e \notin \mathcal{K}(\rho)}{(\mathtt{while}(\chi_\rho, C), \mathcal{K}, \gamma) \overset{err}{\rightarrow} (\mathbf{0}, \mathcal{K}', \gamma')}$$

Table 6.2: Error Rules

Definition 6.2 (Error-free choreographies) *Let $\mathcal{C} = (\Sigma, C_0, \mathcal{K}_0, X)$ be a choreography and let $\rightharpoonup_{err} \subseteq (C_L, K_C, \Gamma_C) \times Act_C \cup \{err\} \times (C_L, K_C, \Gamma_C)$ be the least relation which satisfies the axioms and rules of Tables 6.1 and 6.2 and closed w.r.t. $\equiv$. We say that $\mathcal{C}$ is an error-free choreography if it does not exists a state $\gamma_0 \in \gamma_X$ for which the relation $\rightharpoonup_{err}$, defined starting from the tuple $(C_0, \mathcal{K}_0, \gamma_0)$, contains a transaction labelled with* err.

6.5 Discussion.

This section is devoted to reason about some interesting aspects raised by the choreography approach.

- *Choreography is receiver-centered.* The most interesting aspect of the choreography approach is the fact that an interaction is performed in an atomic way that is, the interaction is performed only when the message is sent and received. Such a feature has a strong impact from the designing point of view because it obliges the designer to reason on the reception of the message and not on its sending. In this sense, we say that choreography is receiver-centered. Let us consider, for example, the following conversation:

  $$\rho_A \rightharpoonup^{\vec{x}}_{o_{\tilde{t}}} \rho_B; \rho_B \rightharpoonup^{\vec{y}}_{o'_{\tilde{t}'}} \rho_C$$

  Such a conversation describes a choreography where the message from the role ρ_B to the role ρ_C can be sent only after the reception of the message from ρ_A to ρ_B. The choreography forces the designer to implement the role ρ_B in a way that, firstly, it receives the message from ρ_A and then it sends the message to ρ_C. The choreography is finished when ρ_C has received the message and not when ρ_B has sent the message to ρ_C. Moreover, let us consider the following not-connected conversation:

  $$\rho_A \rightharpoonup^{\vec{x}}_{o_{\tilde{t}}} \rho_B; \rho_A \rightharpoonup^{\vec{y}}_{o'_{\tilde{t}'}} \rho_C$$

  Such a conversation describes a system where role ρ_A sends a message to role ρ_B and only when the role ρ_B has received the message, role ρ_A can send the message to role ρ_C. Such a kind of choreography implicitly describes a system where there is a hidden synchronization between the role ρ_A and the role ρ_B. Role ρ_A indeed, must be aware of the reception of the message at ρ_B before sending the second message to role ρ_C. In general, we can say that in a choreography interaction we have to always include a receiver's action together with a sender's action even if we do not want it. This fact directly implies that in a choreography it is not possible to design an interaction where a participant in unknown, in conclusion a choreography system is always closed w.r.t. the participants in the sense that all the roles involved within a choreography must be known *a priori*. For instance, let us consider the SOCK system presented in Section 3.4 where we do not consider the customer application:

$$Sys := REG \parallel B \parallel BI \parallel M \parallel S \parallel SI$$

Such a kind of system cannot be represented with a choreography because it is not possible to represent the customer role.

- *Silent actions and Internal actions.* Here we want to highlight the difference between silent actions and internal actions in choreography. The former is strictly related to the computational actions and to the condition evaluation ones, it is performed within a role but it is not relevant from the point of view of the message exchanges. Indeed, it does not directly involve an interaction. On the contrary, the latter deals with an internal action performed within a role which could involve a hidden communication. A role indeed, can be enroled by more than one service which interact each others or it can be implemented by an application where different threads are executed concurrently. In this cases, it is possible that some interactions among the internal services (or threads) exist. In general such a kind of interactions, here represented by $\mathtt{int}(\rho)$, are not relevant to the end of the choreography but there could exist some conversation where they take relevance: it is the case of the non-deterministic choice. To this end, let us consider the following conversation example:

$$\rho_A \rightharpoonup^{\vec{x}}_{o_{\vec{t}}} \rho_B; C + \rho_C \rightharpoonup^{\vec{y}}_{o'_{\vec{t}'}} \rho_B; C' + \mathtt{int}(\rho_B); C''$$

Such a conversation implies a non-deterministic choice among two interactions and an internal action. In this case the role ρ_B can perform an internal action (i.e. an internal message exchange or synchronization within the role ρ_B) and choose to execute the conversation C''. It is worth noting that the action $int(\rho_B)$ is non-deterministically predictable as well as the message receptions of the interactions $\rho_A \rightharpoonup^{\vec{x}}_{o_{\vec{t}}} \rho_B$ and $\rho_C \rightharpoonup^{\vec{y}}_{o'_{\vec{t}'}} \rho_B$. A possible implementation of role ρ_B can be given by exploiting SOCK. In particular, we can imagine that the service behaviour of a service which implements the role ρ_B is modelled as it follows where we abstract away from operation templates:

$$P_{\rho B} ::= (o(\vec{z}); ... + o'(\vec{z'}); ... + sync; ...) \mid ...\overline{sync}...$$

The signal $sync$ is an internal signal exploited for synchronizing internal threads which is designed within the non-deterministic choice. If the synchronization on $sync$ happens before the reception of the messages on o and o', the branch which starts with the primitive $sync$ will be selected.

6.6 A choreography example

In this section we present a choreography example. Let us consider a business scenario where a customer invokes a market service in order to buy some goods and it receives the price as a response. Considering the price, the customer will buy or not the goods. If the customer sends a message for buying the goods, the market will invoke a supplier service for making the order. The supplier service will accept or not the order. In the case the order can be fulfilled, the market service will invoke a bank service for the payment and will return a positive answer to the customer. On the contrary, if the order is not accepted by the supplier service, the market service will notify the negative response to the customer. In order to show a code as clean as possible, we exploit the following notation for representing the One-Way interactions and the Request-Response ones:

$\rho_A \rightharpoonup^{\vec{x}}_{o_{\vec{t}}} \rho_B \equiv OW(\rho_A, \rho_B, o_{\vec{t}}, \vec{x})$

$\rho_A \rightleftharpoons^{\vec{x},\vec{y}}_{o_{\vec{t},\vec{t'}}} \rho_B(C) \equiv RR(\rho_A, \rho_B, o_{\vec{t},\vec{t'}}, \vec{x}, \vec{y}, C)$

In order to define the choreography let us consider four roles: ρ_C which represents the customer behaviour, ρ_M which represents the market service, ρ_B which represents the bank service for credit card payment and ρ_S which represents the supplier service. In the following we define the templates, the operation sets, the information and the roles. In Fig. 6.2 are graphically represented the communication links among the roles.

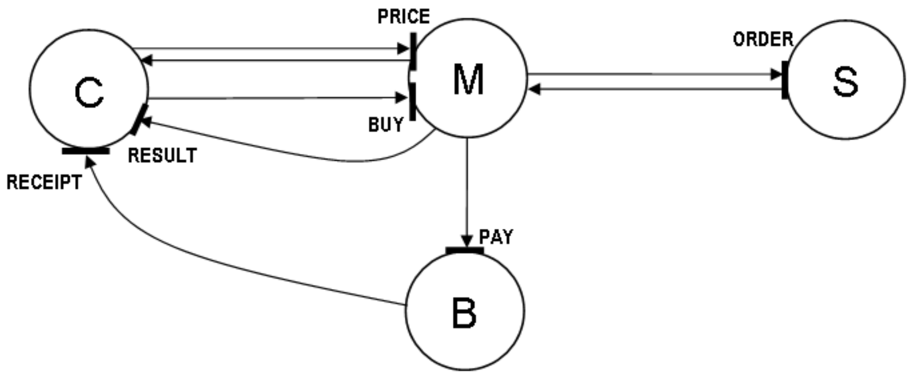

Figure 6.2: Interactions among the roles

We define the following templates:
$t_0 = t_2 = \langle Inf, Inf\rangle$
$t'_0 = t'_2 = t_3 = t_5 = \langle Inf\rangle$
$t_1 = t_4 = \langle Inf, Inf, Inf, Loc\rangle$

$\omega_C = \{\overline{PRICE}_{t_0,t'_0}, \overline{BUY}_{t_1}, RESULT_{t_3}, RECEIPT_{t_5}\}$
$\omega_M = \{PRICE_{t_0,t'_0}, BUY_{t_1}, \overline{ORDER}_{t_2,t'_2}, \overline{PAY}_{t_4}\}$
$\omega_S = \{ORDER_{t_2,t'_2}\}$
$\omega_B = \{PAY_{t_4}, \overline{RECEIPT}_{t_5}\}$.

$I_C = \{good, num, price, buy, card, ncard, outcome, receipt\}$

Let Σ be the set of roles defined in the following way:
$\Sigma = \{(\rho_C, \omega_C), (\rho_M, \omega_M), (\rho_S, \omega_S), (\rho_B, \omega_B)\}$.

We define the following knowledge $\mathcal{K} = (I, \Lambda)$:

$$
\begin{aligned}
I(\rho_C) &= \{buy, good, num, card, ncard\} \\
I(\rho_M) &= \{price\} \\
I(\rho_S) &= \{outcome\} \\
I(\rho_B) &= \{receipt\}
\end{aligned}
$$

$$
\begin{aligned}
\Lambda(\rho_C) &= \{\rho_C\} \\
\Lambda(\rho_M) &= \{\rho_S, \rho_B\} \\
\Lambda(\rho_S) &= \Lambda(\rho_B) = \emptyset
\end{aligned}
$$

Let Con be the following conversation:

$$
\begin{aligned}
Con \quad ::= \quad & RR(\rho_C, \rho_M, PRICE_{t_0, t_0'}, \langle good, num \rangle, price, \mathbf{0}); \\
& BuyChoice; \\
& OW(\rho_C, \rho_M, BUY_{t_1}, \langle buy, card, ncard, \rho_C \rangle); \\
& ExOrder;
\end{aligned}
$$

$$
\begin{aligned}
BuyChoice \quad ::= \quad & \texttt{if}\ price \geq 100_{\rho_C}\ \texttt{then} \\
& \quad buy :=_{\rho_C} cancelled \\
& \texttt{else} \\
& \quad buy :=_{\rho_C} accepted
\end{aligned}
$$

$$
\begin{array}{rcl}
\mathsf{ExOrder} & ::= & \texttt{if}\ \mathsf{buy} == \mathsf{accepted}_{\rho_M}\ \texttt{then} \\
& & \quad \mathsf{RR}(\rho_M, \rho_S, \mathrm{ORDER}_{t_2,t_2'}, \langle \mathsf{good}, \mathsf{num} \rangle, \mathsf{outcome}, \mathbf{0}); \\
& & \quad (\mathsf{OW}(\rho_M, \rho_C, \mathrm{RESULT}_{t_3}, \mathsf{outcome}) \mid \mathsf{Payment}) \\
& & \texttt{else} \\
& & \quad \mathbf{0}
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{Payment} & ::= & \texttt{if}\ \mathsf{outcome} == \mathsf{OK}_{\rho_M}\ \texttt{then} \\
& & \quad \mathsf{OW}(\rho_M, \rho_B, \mathrm{PAY}_{t_4}, \langle \mathsf{card}, \mathsf{ncard}, \mathsf{price}, \rho_C \rangle); \\
& & \quad \mathsf{OW}(\rho_B, \rho_C, \mathrm{RECEIPT}_{t_5}, \mathsf{receipt})
\end{array}
$$

Finally, we define the following initial constraints over the variables:

$$
\begin{array}{rl}
X = & \mathsf{good} \in \{\mathsf{apple}, \mathsf{banana}, \mathsf{strawberry}\} \\
\wedge & 0 \le \mathsf{num} \le 200 \\
\wedge & \mathsf{card} \in \{\mathsf{visa}, \mathsf{mastercard}\} \\
\wedge & \mathsf{ncard} = 123456789 \\
\wedge & \mathsf{buy} = \perp \\
\wedge & 50 \le \mathsf{price} \le 200 \\
\wedge & \mathsf{outcome} \in \{\mathsf{OK}, \mathsf{REJECTED}\} \\
\wedge & \mathsf{receipt} = \mathsf{receiptDoc}
\end{array}
$$

The choreography is defined by the tuple $\mathsf{Chor} = (\Sigma, \mathsf{Con}, \mathcal{K}, X)$ and it is error-free and connected. It is worth noting that within the One-Way message exchange between the customer role and the market one on the operation BUY, also the role name ρ_C is communicated. This means that a location mobility between the two roles is implicitly programmed. Indeed, the market role will communicate the location of the customer to the bank role which will be able to have a message exchange with the customer. If we do not program such a location mobility, the choreography does not satisfy the error-free definition because the bank role will never have the knowledge of the customer location because its initial knowledge is empty and it never receives such an information from the market role. Moreover, the location knowledge of the supplier is initially empty and it never receives any location from the other roles during the execution of teh choreography. Despite of this the Request-Response communication exchange between the market role and the supplier role can be performed because of the hidden location mobility within the Request-Response primitive.

6.7 Comparing C_L and WS-CDL

This section is devoted to compare the static part and the dynamic one. Since WS-CDL is not equipped with a formal sematics, we do not discuss the execution part of C_L but we will make some considerations by exploiting an example extracted from the WS-CDL specifications.

6.7.1 Static part

In C_L the static part deals with the role and the information declarations. Each role is joined with a name and a set of operations. We can compare such a part with the WS-CDL types part where roleTypes, relationshipTypes, participantType, channelType and InformationType are declared. In C_L we do not distinguish the concepts of behaviour and role but we model both within a unique concept which is that of role. Furthermore, C_L abstracts away from the relationshipType and the participantType concepts which can be indirectly obtained by considering the declared operations within each role. For example, let us consider Fig. 6.1 where there are three roles. It is possible to extract three different relationshipType: a relationshipType between A and B, a relationshipType between B and C and a relationshipType between C and A. Moreover, as far as the participantType is concerned, each participantType corresponds with a role. In WS-CDL a channel can correspond to a set of operations located at a specific role and, possibly, characterized by conversation instance information (e.g. correlation data). At the present, C_L does not deal with conversation instances and the concept of channel is related to that of operation. Furthermore, it is worth noting that WS-CDL allows for the communication of a channel between two roles. In C_L such a feature can be partially modelled with the possibility to express a location communication. In this case we do not communicate all the information related to a channel but only the role name where it is located. Such a feature is related to the fact that only location mobility is considered. Interface mobility indeed, is not taken into account because it is not actually implemented by Web Services specifications. As far as the informationType is concerned, it could be compared with the information set I_C of C_L even if the latter represent an actual information whereas the former is namely a type. The main difference is related to the fact that an information which belongs to the set I_C has the same value in every role it is known whereas the informationType describes only a variable type that can have different values on different roles.

6.7.2 Dynamic part

In Tab 6.3, 6.4 and 6.5 is reported a strict comparison betweeen the syntax constructs of the C_L language and those of WS-CDL. In the following we discuss each construct:

- *One-Way*: the One-Way is expressed in WS-CDL by exploiting the tag *interaction* where there is only a tag *exchange* defined in. Such a tag has the attribute *action* set to *request*. It is worth noting that the effect of a One-Way in C_L is to communicate the information x from the sender role to the receiver one. We have modelled such a feature by indicating x both within the tag *send* and *receive*. The informationType is set to the message template $\vec{t}$ because it deals with the representation of the exchanged data and it can be indirectly related to the type of the information. The tag *record* of the interaction is not exploited because the One-Way of C_L cannot have effects on other information except those that are communicated.

- *Request-Response*: the Request-Response is modelled in WS-CDL by exploiting two different interaction tags where the former deals with the request message whereas the latter deals with the response one. In WS-CDL indeed, as we have discussed in [GGL05] does not exist a unique construct which deals with the Request-Response.

- *Internal activity*: the internal activity is simply modelled with a silent action.

- *Assign*: the assign primitive is modelled by exploiting the WS-CDL *activity*. The syntax is self-explaining.
- *Sequence, Parallel and Choice*: each of these constructs has a corresponding one in WS-CDL. The syntax is self-explaining.
- *Deterministic choice and Iteration*: these constructs in WS-CDL are modelled by exploiting the *workunit* construct because it allows for the specification of guard conditions.

It is worth noting that C_L does not deal with the exceptionBlock and the finalizer one. Such constructs are strictly related to faults management which is a topic out of the scope of this work.

6.7.3 Modelling a WS-CDL example by using C_L

In this section we model the WS-CDL example shown in section 2.1.4.3 where we do not take into account fault messages. The system is composed by two roles whose names are ρ_B and ρ_S and which represents the roles *BuyerRole* and *SellerRole* respectively. Role ρ_S exhibits two Request-Response operations, *getQuote* and *updateQuote* and a One-Way one *order*. We define the following template and operation sets:

$\vec{t} = \langle inf \rangle$
$\omega_B = \{\overline{getQuote}_{\vec{t},\vec{t}}, \overline{order}_{\vec{t}}, \overline{updateQuote}_{\vec{t},\vec{t}}\}$
$\omega_S = \{getQuote_{\vec{t},\vec{t}}, order_{\vec{t}}, updateQuote_{\vec{t},\vec{t}}\}$
Let Σ be the set of roles defined in the following way:
$\Sigma = \{(\rho_B, \omega_B), (\rho_S, \omega_S)\}$
The information set is defined as follows where the name of the variables is the same of the example presented in the WS-CDL section:
$I_C = \{quoteRequest, quoteResponse, orderRequest, barteringDone\}$
As far as the conversation is concerned, by considering tables 6.3, 6.4 and 6.5, we can initially translate it without taking into account the knowledge and the initial constraints

	C_L	$WS - CDL$

One-Way	$\rho_A \rightharpoonup^{x}_{o_{\vec{t}}} \rho_B$	`<interaction name=” . . . ”`
		`channelVariable=” . . . ”`
		`operation=”o” >`
		` <participate relationshipType=” . . . ”`
		` fromRoleTypeRef=”rhoA”`
		` toRoleTypeRef=”rhoB”/>`
		` <exchange name=” . . . ”`
		` informationType=”t”`
		` action=”request”>`
		`  <send variable=”x” />`
		`  <receive variable=”x”/>`
		` </exchange>`
		`</interaction>`

Request-Response	$\rho_A \rightleftharpoons^{x,y}_{o_{\vec{t},\vec{t'}}} \rho_B(C)$	`<interaction name=”IntName”`
		`channelVariable=” . . . ”`
		`operation=”o” >`
		` <participate relationshipType=” . . . ”`
		` fromRoleTypeRef=”rhoA”`
		` toRoleTypeRef=”rhoB”/>`
		` <exchange name=” . . . ”`
		` informationType=”t” action=”request”>`
		`  <send variable=”x” />`
		`  <receive variable=”x”/>`
		` </exchange>`
		`</interaction>`
		`C`
		`<interaction name=”IntName”`
		`channelVariable=” . . . ”`
		`operation=”o” >`
		` <participate relationshipType=” . . . ”`
		` fromRoleTypeRef=”rhoB”`
		` toRoleTypeRef=”rhoA”/>`
		` <exchange name=” . . . ”`
		` informationType=”t ’” action=”response”>`
		`  <send variable=”y” />`
		`  <receive variable=”y”/>`
		` </exchange>`
		`</interaction>`

Table 6.3: Comparing communication primitives between C_L and WS-CDL

	C_L	$WS - CDL$
Internal activity	$\mathit{int}(\rho)$	`<silentAction roleType="rho" />`
Assign	$i :=_\rho e$	`<assign roleType="rho">`
		` <copy name="...">`
		`  <source variable="e" />`
		`  <target variable="i" />`
		` </copy>`
		`</assign>`
Sequence	$C1; C2$	`<sequence>`
		`C1`
		`C2`
		`</sequence>`
Parallel	$C1 \mid C2$	`<parallel>`
		`C1`
		`C2`
		`</parallel>`
Choice	$\sum^{+}_{i \in H} \eta_i; C_i$	`<choice>`
		`  ...`
		` <sequence>`
		`  eta_i`
		`  C_i`
		` </sequence>`
		`  ...`
		`</choice>`

Table 6.4: Comparing constructs of C_L and WS-CDL

	C_L	$WS - CDL$

Det. choice	`if` χ_ρ `then` C1 `else` C2	`<choice>`
		`<workunit`
		`name=""`
		`guard="chi" >`
		`C1`
		`</workunit>`
		`<workunit`
		`name=""`
		`guard="not␣chi" >`
		`C2`
		`</workunit>`
		`</choice>`

Iteration	`while`(χ_ρ, C)	`<workunit`
		`name=""`
		`guard="chi"`
		`repeat="true" >`
		`C`
		`</workunit>`

Table 6.5: Comparing constructs of C_L and WS-CDL

that will be discussed in the following:

$$
\begin{aligned}
\mathsf{Bartering} ::= {} & \mathsf{barteringDone} :=_{\rho_S} \mathsf{false} \\
& ; \rho_B \rightleftharpoons^{\mathsf{quoteRequest},\mathsf{quoteResponse}}_{\mathsf{getQuote}_{\vec{t},\vec{t}}} \rho_S(\mathbf{0}) \\
& ; \mathtt{while}((\mathsf{barteringDone} == \mathsf{false})_{\rho_S}, \mathsf{BodyW})
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{BodyW} ::= {} & (\rho_B \rightharpoonup^{\mathsf{orderRequest}}_{\mathsf{order}_{\vec{t}}} \rho_S; \mathsf{barteringDone} :=_{\rho_S} \mathsf{true}) \\
& + \\
& (\rho_B \rightleftharpoons^{\mathsf{quoteRequest},\mathsf{quoteResponse}}_{\mathsf{updateQuote}_{\vec{t},\vec{t}}} \rho_S(\mathbf{0}))
\end{aligned}
$$

The conversation describes a bartering process which starts with a message exchange between the buyer and the seller where the former asks for a *quoteRequest* and it receives a *quoteResponse*. The bartering process continues starting a cycle which finishes only when the buyer sends an order message on the *order* operation. If the buyer requests for an updating of the quote the cycle continues waiting for the next interaction. The information *barteringDone* maintains the state of the bartering process, if it is equal to *true* the process is finished. Such a representation strictly follows that of WS-CDL but some considerations about the non deterministic choice must be done. In C_L the non-deterministic choice models a race among the messages specified within the branches where the first interaction that completes has the effect to disable the other ones. In the example above, namely, there is not a real race between the two interactions because the buyer internally decides to ask for an updating or not and the interaction to perform is a consequence of that choice. In order to correctly model such an example in C_L, we replace the *barteringDone* information with another information, *buyerOrder*, which allows us to establish if the buyer wants to perform an order or not. In other words, we shift the knowledge of the bartering ending from the seller to the buyer. The variable *buyerOrder* will range between two values true or false within the initial constraints and it allows us to replace the non-deterministic choice with a deterministic one by means of an if then else process where the condition tests if such a variable is true or not. In the following, we present the new conversation where we do not consider the while construct which we comment later.

$$
\begin{aligned}
\mathsf{Bartering} ::= {} & \rho_B \rightleftharpoons^{\mathsf{quoteRequest},\mathsf{quoteResponse}}_{\mathsf{getQuote}_{\vec{t},\vec{t}}} \rho_S(\mathbf{0}) \\
& ;\mathsf{if}\ (\mathsf{BuyerOrder} == \mathsf{true})_{\rho_B}\ \mathsf{then} \\
& \qquad \rho_B \rightharpoonup^{\mathsf{orderRequest}}_{\mathsf{order}_{\vec{t}}} \rho_S \\
& \mathsf{else} \\
& \qquad \rho_B \rightleftharpoons^{\mathsf{quoteRequest},\mathsf{quoteResponse}}_{\mathsf{updateQuote}_{\vec{t},\vec{t}}} \rho_S(\mathbf{0})
\end{aligned}
$$

where we define the information set and the knowledge $\mathcal{K}(I,\Lambda)$ as follows:

$$I_C = \{\mathsf{quoteRequest}, \mathsf{quoteResponse}, \mathsf{orderRequest}, \mathsf{buyerOrder}\}$$

$$
\begin{aligned}
I(\rho_B) &= \{\mathsf{quoteRequest}, \mathsf{orderRequest}, \mathsf{buyerOrder}\} \\
I(\rho_S) &= \{\mathsf{quoteResponse}\}
\end{aligned}
$$

$$
\begin{aligned}
\Lambda(\rho_B) &= \{\rho_S\} \\
\Lambda(\rho_S) &= \{\emptyset\}
\end{aligned}
$$

The initial constraints definition follows where $\mathsf{msg1}$, $\mathsf{msg2}$ and $\mathsf{msg3}$ represent some kind of message:

$$
\begin{aligned}
X ::= {} & \mathsf{quoteRequest} = \mathsf{msg1} \\
& \wedge \mathsf{quoteResponse} = \mathsf{msg2} \\
& \wedge \mathsf{orderRequest} = \mathsf{msg3} \\
& \wedge \mathsf{buyerOrder} = \{\mathsf{true}, \mathsf{false}\}
\end{aligned}
$$

If we consider the semantics rule of Table 6.1 such a choreography is represented by two labelled transition systems where in the former one the variable *buyerOrder* has the initial value set to true whereas in the latter one it is set to false. This fact implies that in the former labelled transition system only the branch $\rho_B \rightharpoonup^{\mathsf{orderRequest}}_{\mathsf{order}_{\vec{t}}} \rho_S$ is executed whereas in the latter one the branch $\rho_B \rightleftharpoons^{\mathsf{quoteRequest},\mathsf{quoteResponse}}_{\mathsf{updateQuote}_{\vec{t},\vec{t}}} \rho_S(\mathbf{0})$ is executed. Now, let us introduce the while construct:

$$\begin{aligned}\mathtt{Bartering} ::= {} & \rho_B \rightleftharpoons^{quoteRequest,quoteResponse}_{getQuote_{\tilde{t},\tilde{t}}} \rho_S(\mathbf{0}) \\ & ;\mathtt{while}((buyerOrder == false)_{\rho_S}, \rho_B \rightleftharpoons^{quoteRequest,quoteResponse}_{updateQuote_{\tilde{t},\tilde{t}}} \rho_S(\mathbf{0})) \\ & ;\rho_B \rightharpoonup^{orderRequest}_{order_{\tilde{t}}} \rho_S\end{aligned}$$

The semantics is represented by two labelled transition systems also in this case. It is worth noting that, where the variable $buyerOrder$ is set to $true$ only the interaction $;\rho_B \rightharpoonup^{orderRequest}_{order_{\tilde{t}}} \rho_S$ is executed whereas where the variable $buyerOrder$ is set to $false$, it is always executed the interaction $\rho_B \rightleftharpoons^{quoteRequest,quoteResponse}_{updateQuote_{\tilde{t},\tilde{t}}} \rho_S(\mathbf{0})$.
This example shows that the semantics of the non-deterministic choice in C_L is different from that informally given to the WS-CDL choice construct. In WS-CDL indeed, the choice construct can model a race among some interactions but it can also model an intrinsic non-determinism related to the fact that the value of an internal variable of a role (e.g. the $barteringDone$ variable) is *a priori* unknown. In C_L the non-deterministic choice only models a race among some interactions and the implicit non-determinism about the value of a variables is modelled by considering all the possible values for that variable within the initial constraints. Such a fact implies that the semantics of the given choreography will be represented by more than one labelled transition system.

6.8 Related works

There are other works which deals with a formal representation of choreography. In [CHY07] Honda et al. present a typed choreography language inspired by WS-CDL called global calculus. Suck a kind of calculus is based upon the concept of service channels and sessions. Two dialoguers can start a communication by opening a session on a channel. Differently from the approach proposed within this thesis where there are no sessions and interactions are always performed by exploiting an operation that can be seen as a communication channel, in the work of Honda et al. the session is a unique name shared by two participants on which several message exchanges can be performed. In [DZD06] Decker et al. present a graphical language for choreography where interactions can be inter-related by three kind of relations: *precedes*, *inhibits* and *weak-precedes*.

The precede relationship allows for the specification of a sort of sequence between two interactions, the inhibits allows for the inhibition of an interaction and the weak-precedes expresses a weak sequence between two interactions where it is not necessary that the first interaction is completely executed for starting the second one. Moreover it is possible to express guard conditions and repetitions. The execution semantics is given in terms of pi-calculus processes. In this work the authors do not deal with any kind of communication channels on which the communication is performed. In [Fos06] gives a partial representation of choreographies, focussed on interaction modelling, in terms of labelled transition systems. In [MS06] Montagero and Semini characterize choreography as a pair formed by a logical theory, that expresses all the properties of the choreography, and a formula holding in the theory which gives an abstract view of the behaviour of the choreography. In [DD04] Dumas et al. present a choreography representation in terms of Petri Nets but they only consider interactions as transitions without differentiating on specific constructs for designing communication pattern as Request-Response or composing operators like parlallel, non-deterministic choice, etc. Similarly, in [BBM^{+}05b] Baldoni et al. describe a choreography in terms of a state finite automata where message exchanges are represented by labelled transitions, but they do not deal with language aspects for its representation.

Chapter 7

Orchestration

In this chapter we introduce a formal language for orchestration based on the language SOCK. In order to be coherent with SOCK, in the following we use the terms orchestrator and service engine, and the terms services system and orchestrated system, as synonymous. The language is structured on three layers: the service behaviour, the service engine and the services system. As we have described in section 3.1, the former deals with the design of service behaviours by supplying computation and external communication primitives where, in this case, the operation definitions are enriched by considering the template definition given in section 4.2.1. Here, we introduce templates because in the following they will be fundamental to the end of conformance where we will have to distinguish between communicated location and information. The service engine layer allows us to design a service engine that, differently from SOCK, here is limited to a service behaviour joined with a state and a location. As far as the the bipolar framework is concerned indeed, at the state of the art it does not deal with session management. Finally, the services system layer follows that defined within the SOCK calculus. It is worth noting that here we model asynchronous communication which was not considered in the previous sections. Asynchronous communications are a basic characteristic of SOC systems and, to the end of conformance, they play a fundamental role. In general indeed, the labelled transition system of an orchestrated system is different if asynchronous communications are considered or not. It is worth noting, that the semantics rules which deal with asynchronous communications can be easily applied to the SOCK calculus in order to enhance it with such a kind of communication model.

7.1 The syntax

Formally, let $\mathit{Signals}$ be a set of signal names exploited for synchronizing processes in parallel within a process. Let Var be a set of variables ranged over by x, y, z. We exploit the notations $\vec{x} = \langle x_0, x_1, ..., x_i \rangle$ for representing arrays of variables. Let k ranges over $\mathit{Var} \cup \mathrm{Loc}$ where $\mathit{Var} \cap \mathrm{Loc} = \emptyset$. The language is defined as it follows where we intend W as a finite non-empty set of indexes, P and Q as service behaviour processes, P_S as a service behaviour joined with a state, Y as a service engine and E as a services system:

$$\epsilon ::= s \mid o_{\vec{t}}(\vec{x}) \mid o_{\vec{t},\vec{t}'}(\vec{x}, \vec{y}, P)$$
$$\overline{\epsilon} ::= \bar{s} \mid \overline{o}_{\vec{t}}@k(\vec{x}) \mid \overline{o}_{\vec{t},\vec{t}'}@k(\vec{x}, \vec{y})$$

$P, Q, \ldots ::=$	processes	
$\mathbf{0}$		null process
$\overline{\epsilon}$		output
ϵ		input
$x := e$		assignment
$\chi?P : Q$		if then else
$P; P$		sequence
$P \mid P$		pararallel
$\sum^{+}_{i \in W} \epsilon_i; P_i$		non-det. choice
$\chi \rightleftharpoons P$		iteration

$P_S ::= (P, S)$

$Y ::= [P_S]@l$

$E ::= Y \mid E \parallel E$

We denote with OL the set of all possible orchestrated systems ranged over by E. $\mathbf{0}$ is the null process. Outputs can be a signal $\bar{s}$, a Notification $\bar{o}_{\vec{t}}@k(\vec{x})$ or a Solicit-Response $\bar{o}_{\vec{t},\vec{t}'}@k(\vec{x},\vec{y})$ where $s \in \mathit{Signals}$, $o \in \mathcal{O}$, $\vec{t}, \vec{t'} \in \mathcal{T}$, $k \in \mathit{Var} \cup \mathit{Loc}$ represents the receiver location which can be explicit or represented by a variable[1], $\vec{x}$ is the tuple of the variables which store the information to send and $\vec{y}$ is the tuple of variables where, in the case of the Solicit-Response, the received information will be stored. Dually, inputs can be an input signal s, a One-Way $o_{\vec{t}}(\vec{x})$ or a Request-Response $\bar{o}_{\vec{t},\vec{t}'}@k(\vec{x},\vec{y})$ where $s \in \mathit{Signals}$, $o \in \mathcal{O}$, $\vec{t}, \vec{t'} \in \mathcal{T}$, $\vec{x}$ is the tuple of variables where the received information are stored whereas $\vec{y}$ is the tuple of variables which contain the information to send in the case of the Request-Response; finally P is the process that has to be executed between the request and the response. $x := e$ assigns the result of the expression e to the variable x. For the sake of brevity, we do not present the syntax for representing expressions, we assume that they include all the arithmetic operators, values in Val and variables. $\chi?P : Q$ is the if then else process, where χ is a logic condition on variables whose syntax is:

$$\chi ::= x \leq e \mid e \leq x \mid \neg\chi \mid \chi \wedge \chi$$

It is worth noting that conditions such as $x = v$, $x \neq v$ and $v_1 \leq x \leq v_2$ can be defined as abbreviations; P is executed only if the condition χ is satisfied, otherwise Q is executed. $P;P$, $P \mid P$ represent sequential and parallel composition respectively whereas $\sum^{+}_{i \in W} \epsilon_i; P_i$ is the non-deterministic choice restricted to be guarded on inputs. $\chi \rightleftharpoons P$ is the construct to model guarded iterations. P_S represents a service behaviour P coupled with a state of variables $\mathcal{S}$ which is a function $\mathcal{S} : \mathit{Var} \rightarrow \mathit{Val} \cup \{\bot\}$ mapping variables to values. The term Y denotes a service engine which is a service behaviour coupled with a state and joined to a specific location l. Finally, an orchestrated system E consists of the parallel composition of orchestrators. Before presenting the semantics of the language we introduce the following satisfaction relation $\vdash$ between a template and a vector of values where length is a function which returns the element number of a vector:

[1]It is worth noting that, in order to fullfil location mobility, k must be used as a variable and it has to contain a location.

$\vec{v} \vdash \vec{t} \Rightarrow length(\vec{t}) = length(\vec{v}) \wedge \forall i < length(\vec{t}), t[i] = Type(v[i])$

7.2 The semantics.

The semantics of the orchestration language follows the idea of that of SOCK language. It is defined in terms of a labelled transition system structured into four layers: *service behaviour lts layer* (Tables 7.1 and 7.2), the *service engine state lts layer* (Table 7.3), the *service engine location lts layer* (Table 7.4) and the *services system lts layer*. Each lts layer catches the actions raised by the underlying one and will enable or disable them. If an action is enabled by an lts layer it will be raised to the overlying one. The service behaviour lts layer describes all the possible execution paths generated by a service process. The service engine state lts layer defines the rule for joining a service behaviour with a service engine local state, the service engine location lts layer deals with the the rules for deploying a service engine at a specific location and, finally, the services system lts layer deals with a composed service engine system. We assume as state definition that presented in Section 3.2.1. In order to deal with asynchronous communication and Request-Response message exchange, we extend the syntax by introducing the following terms:

$$P ::= \ldots \mid \langle \overline{o}_{\vec{t}}@l(\vec{v})\rangle \mid \langle \overline{o}_{\vec{t},\vec{t}'}@l(\vec{v})\rangle \mid \langle \overline{o}_{\vec{t},\vec{t}'}@l(\vec{v},\vec{y})\rangle \mid \overline{o}_{\vec{t},\vec{t}'}@l(\vec{y}) \mid o_{\vec{t},\vec{t}'}(\vec{y})$$

7.2.1 The service behaviour lts layer

We define $\rightarrow \subseteq P \times Act \times P$ as the least relation which satisfies the axioms and rules of Tables 7.1 and 7.2 closed w.r.t. $\equiv$, where $\equiv$ statisfies the structural congruence rules at the end of the table. Let $Act = In \cup Out \cup Internal$ be the set of actions ranged over by a where the set In represents the set of the external input actions, the set Out respresents the set of the exeternal output actions and $Internal$ is the set of the internal actions. As we have shown for SOCK, the service behaviour calculus does not deal with the actual

(IN)

$$s \xrightarrow{s} \mathbf{0}$$

(OUT)

$$\bar{s} \xrightarrow{\bar{s}} \mathbf{0}$$

(ONE-WAYOUTASYN)

$$\frac{\vec{v} \vdash \vec{t}}{\bar{o}_{\vec{t}}@z(\vec{x}) \xrightarrow{\bar{o}_{\vec{t}}@l/z(\vec{v}/\vec{x})} \langle \bar{o}_{\vec{t}}@l(\vec{v}) \rangle}$$

(ONE-WAYOUTLOCASYN)

$$\frac{\vec{v} \vdash \vec{t}}{\bar{o}_{\vec{t}}@l(\vec{x}) \xrightarrow{\bar{o}_{\vec{t}}@l(\vec{v}/\vec{x})} \langle \bar{o}_{\vec{t}}@l(\vec{v}) \rangle}$$

(ONE-WAYOUT)

$$\langle \bar{o}_{\vec{t}}@l(\vec{v}) \rangle \xrightarrow{\bar{o}_{\vec{t}}@l(\vec{v})} \mathbf{0}$$

(ONE-WAYIN)

$$\frac{\vec{v} \vdash \vec{t}}{o_{\vec{t}}(\vec{x}) \xrightarrow{l:o_{\vec{t}}(\vec{v}/\vec{x})} \mathbf{0}}$$

(REQ-OUTASYN)

$$\frac{\vec{v} \vdash \vec{t}}{\bar{o}_{\vec{t},\vec{t}'}@z(\vec{x},\vec{y}) \xrightarrow{l':\bar{o}_{\vec{t},\vec{t}'}@l/z(\vec{v}/\vec{x},\vec{y})} \langle \bar{o}_{\vec{t},\vec{t}'}@l(\vec{v},\vec{y}) \rangle}$$

(REQ-OUTLOCASYN)

$$\frac{\vec{v} \vdash \vec{t}}{\bar{o}_{\vec{t},\vec{t}'}@l(\vec{x},\vec{y}) \xrightarrow{l':\bar{o}_{\vec{t},\vec{t}'}@l(\vec{v}/\vec{x},\vec{y})} \langle \bar{o}_{\vec{t},\vec{t}'}@l(\vec{v},\vec{y}) \rangle}$$

(REQ-OUT)

$$\langle \bar{o}_{\vec{t},\vec{t}'}@l(\vec{v},\vec{y}) \rangle \xrightarrow{l':\bar{o}_{\vec{t},\vec{t}'}@l(\vec{v},\vec{y})} o_{\vec{t},\vec{t}'}(\vec{y})$$

(REQ-IN)

$$\frac{\vec{v} \vdash \vec{t}}{o_{\vec{t},\vec{t}'}(\vec{x},\vec{y},P) \xrightarrow{l':o_{\vec{t},\vec{t}'}(\vec{v}/\vec{x},\vec{y},P)@l} P;\bar{o}_{\vec{t},\vec{t}'}@l(\vec{y})}$$

(RESP-OUTLOCASYN)

$$\frac{\vec{v} \vdash \vec{t}}{\bar{o}_{\vec{t},\vec{t}'}@l(\vec{x}) \xrightarrow{l':\bar{o}_{\vec{t},\vec{t}'}@l(\vec{v}/\vec{x})} \langle \bar{o}_{\vec{t},\vec{t}'}@l(\vec{v}) \rangle}$$

(RESP-OUT)

$$\langle \bar{o}_{\vec{t},\vec{t}'}@l(\vec{v}) \rangle \xrightarrow{l':\bar{o}_{\vec{t},\vec{t}'}@l(\vec{v})} \mathbf{0}$$

(RESP-IN)

$$\frac{\vec{v} \vdash \vec{t}}{o_{\vec{t},\vec{t}'}(\vec{x}) \xrightarrow{l:o_{\vec{t},\vec{t}'}(\vec{v}/\vec{x})} \mathbf{0}}$$

Table 7.1: Rules for service behaviour lts layer (1)

(ASSIGN) $x := e \overset{x:=v/e}{\longrightarrow} \mathbf{0}$

(IF THEN) $\chi?P : Q \xrightarrow{\chi?} P$

(ELSE) $\chi?P : Q \xrightarrow{\neg\chi?} Q$

(ITERATION) $\chi \rightleftharpoons P \xrightarrow{\chi?} P; \chi \rightleftharpoons P$

(NOT ITERATION) $\chi \rightleftharpoons P \xrightarrow{\neg\chi?} \mathbf{0}$

(SYNCHRO) $$\frac{P \xrightarrow{s} P', Q \xrightarrow{\bar{s}} Q'}{P \mid Q \xrightarrow{\pi} P' \mid Q'}$$

(SEQUENCE) $$\frac{P \xrightarrow{a} P'}{P; Q \xrightarrow{a} P'; Q}$$

(PARALLEL) $$\frac{P \xrightarrow{a} P'}{P \mid Q \xrightarrow{a} P' \mid Q}$$

(CHOICE) $$\frac{\epsilon_i \xrightarrow{a} \mathbf{0} \quad i \in W}{\sum^{+}_{i \in W} \epsilon_i; P_i \xrightarrow{a} P_i}$$

STRUCTURAL CONGRUENCE

$$P \mid Q \equiv Q \mid P \qquad P \mid \mathbf{0} \equiv \mathbf{0} \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R \qquad \mathbf{0}; P \equiv P$$

$$\langle \bar{o}_{\vec{t}} @ l(\vec{v}) \rangle ; Q \equiv \langle \bar{o}_{\vec{t}} @ l(\vec{v}) \rangle \,| Q$$

$$\langle \bar{o}_{\vec{t},\vec{t}'} @ l(\vec{v}, \vec{y}) \rangle ; Q \equiv \langle \bar{o}_{\vec{t},\vec{t}'} @ l(\vec{v}, \vec{y})) \rangle \,| Q \qquad \langle \bar{o}_{\vec{t},\vec{t}'} @ l(\vec{v}) \rangle ; Q \equiv \langle \bar{o}_{\vec{t},\vec{t}'} @ l(\vec{v}) \rangle \,| Q$$

Table 7.2: Rules for service behaviour lts layer2

values of variables and locations but it models all the possible execution paths for all the possible variable values and locations. The action sets are represented as follows:

$$In = \{l{:}\, o_{\vec{t}}(\vec{v}/\vec{x}), l{:}\, o_{\vec{t},\vec{t}'}(\vec{v}/\vec{x}), l'{:}\, o_{\vec{t},\vec{t}'}(\vec{v}/\vec{x}, \vec{y}, P)@l, l'{:}\, o_{\vec{t},\vec{t}'}(\vec{v}/\vec{x})\}$$

$$\begin{aligned} Out = \{&\bar{o}_{\vec{t}}@l/z(\vec{v}/\vec{x}), \bar{o}_{\vec{t}}@l(\vec{v}/\vec{x}), \bar{o}_{\vec{t}}@l(\vec{v}),\\ &l'{:}\,\bar{o}_{\vec{t},\vec{t}'}@l/z(\vec{v}/\vec{x}, \vec{y}), l'{:}\,\bar{o}_{\vec{t},\vec{t}'}@l(\vec{v}/\vec{x}, \vec{y}), l'{:}\,\bar{o}_{\vec{t},\vec{t}'}@l(\vec{v}, \vec{y}),\\ &l'{:}\,\bar{o}_{\vec{t},\vec{t}'}@l(\vec{v}/\vec{x}), l'{:}\,\bar{o}_{\vec{t},\vec{t}'}@l(\vec{v})\}\end{aligned}$$

$$Internal = \{s, \bar{s}, x := v/e, \chi?, \neg\chi?, \pi\}$$

Rules IN, OUT and SYNCHRO deal with internal synchronizations among parallel processes. It is worth noting that, differently from SOCK, here we label a synchronization between signals with a π action which will be useful, to the end of conformance, for distinguishing internal synchronizations from computational steps represented by the action τ [2]. In particular, the action π plays a fundamental role in the case of the non-deterministic choice where, by syntax definition, some branches could be selected by internal synchronizations; this fact becomes important if we consider the conformance with a choreography where it is possible to design a non-deterministic choice among interactions and internal actions. To this end, in the service engine location lts layer, the action π will be joined with the location of the orchestrator and in Chapter 8, within the conformance notion, it will be related with the choreography internal action ($int(\rho)$). Rules ONE-WAYOUTASYN and ONE-WAYOUTLOCASYN deal with the asynchronous sending of a message on the Notification operation $\bar{o}_{\vec{t}}$ where, in the former case, the location is stored within a variable (z) whereas in the latter one the location is statically explicited. Both rules produce the term $\langle \bar{o}_{\vec{t}}@l(\vec{v})\rangle$ which freezes the actual values of the variables and the location. Such a term allows us to describe asynchronous communication (rule ONE-WAYOUT) because, by means of the structural congruence rules, it is always executed in parallel. Rule ONE-WAYIN deals with the reception of a message on a One-Way opera-

[2] The τ actions will be considered only at the level of the service engine.

tion where the location l within the action represents the location of the receiver; such a location will be joined within the service engine location lts layer and it will be exploited within the services system lts layer for allowing communication among different service engines. Rules REQ-OUTASYN, REQ-OUTLOCASYN and REQ-OUT follow the same idea of the previous ones but they deal with a Solicit-Response operation $\overline{o}_{\vec{t},\vec{t}'}$ instead of a Notification one. It is worth noting that in rule REQ-OUT, after the message sent, it is produced the term $o_{\vec{t},\vec{t}'}(\vec{y})$ which allows for the reception of the reply message on the same operation. Rule REQ-IN describes the Request-Response operation behaviour: when the message is received the process P is executed and the term $\overline{o}_{\vec{t},\vec{t}'}@l(\vec{y})$ allows for the sending of the reply message. The location l' within the produced actions, as well as the location of the action produced in the ONE-WAYIN rule, represents the location of the service engine which executes the primitive. Rules RESP-OUTLOCASYN and RESP-OUT deal with the semantics of the sending of the reply message in a Request-Response message exchange. It is worth noting that the location is always statically defined because it is received within the request message of the Request-Response. Rule RESP-IN deals with the reception of a reply message. Rule ASSIGN deals with the assignment whereas rules IF THEN, ELSE, ITERATION and NOT ITERATION deal with *if then else* process and iteration respectively. Finally, SEQUENCE, PARALLEL and CHOICE describe the sequence, the parallel and the non-deterministic composition operators respectively.

7.2.2 The service engine state lts layer.

We define $\rightarrow_S \subseteq P_S \times Act_S \times P_S$ as the least relation which satisfies the rules of Table 7.3 where the set of actions $Act_S = In_S \cup Out_S \cup Internal_S$ is defined as follows:

$$In_S = \{l{:}\, o_{\vec{t}}(\vec{v}),\, l{:}\, o_{\vec{t},\vec{t}'}(\vec{v}),\, l'{:}\, o_{\vec{t},\vec{t}'}(\vec{v},\vec{y},P)@l\}$$

$$Out_S = \{\overline{o}_{\vec{t}}@l(\vec{v}),\, l'{:}\,\overline{o}_{\vec{t},\vec{t}'}@l(\vec{v},\vec{y}),\, l'{:}\,\overline{o}_{\vec{t},\vec{t}'}@l(\vec{v})\}$$

(SYNCHRO)
$$\frac{P \xrightarrow{\pi} P'}{(P, \mathcal{S}) \xrightarrow{\pi}_S (P', \mathcal{S})}$$

(ONE-WAYOUTASYN)
$$\frac{P \xrightarrow{\bar{o}_{\vec{t}}@l/z(\vec{v}/\vec{x})} P', \mathcal{S}(z) = l, \mathcal{S}(\vec{x}) = \vec{v}}{(P, \mathcal{S}) \xrightarrow{\tau}_S (P', \mathcal{S})}$$

(ONE-WAYOUTLOCASYN)
$$\frac{P \xrightarrow{\bar{o}_{\vec{t}}@l(\vec{v}/\vec{x})} P', \mathcal{S}(\vec{x}) = \vec{v}}{(P, \mathcal{S}) \xrightarrow{\tau}_S (P', \mathcal{S})}$$

(ONE-WAYOUT)
$$\frac{P \xrightarrow{\bar{o}_{\vec{t}}@l(\vec{v})} P'}{(P, \mathcal{S}) \xrightarrow{\bar{o}_{\vec{t}}@l(\vec{v})}_S (P', \mathcal{S})}$$

(ONE-WAYIN)
$$\frac{P \xrightarrow{l:o_{\vec{t}}(\vec{v}/\vec{x})} P'}{(P, \mathcal{S}) \xrightarrow{l:o_{\vec{t}}(\vec{v})}_S (P', \mathcal{S}[\vec{v}/\vec{x}])}$$

(REQ-OUTASYN)
$$\frac{P \xrightarrow{l':\bar{o}_{\vec{t},\vec{t}'}@l/z(\vec{v}/\vec{x},\vec{y})} P', \mathcal{S}(z) = l, \mathcal{S}(\vec{x}) = \vec{v}}{(P, \mathcal{S}) \xrightarrow{\tau}_S (P', \mathcal{S})}$$

(REQ-OUTLOCASYN)
$$\frac{P \xrightarrow{l':\bar{o}_{\vec{t},\vec{t}'}@l(\vec{v}/\vec{x},\vec{y})} P', \mathcal{S}(\vec{x}) = \vec{v}}{(P, \mathcal{S}) \xrightarrow{\tau}_S (P', \mathcal{S})}$$

(REQ-OUT)
$$\frac{P \xrightarrow{l':\bar{o}_{\vec{t},\vec{t}'}@l(\vec{v},\vec{y})} P'}{(P, \mathcal{S}) \xrightarrow{l':\bar{o}_{\vec{t},\vec{t}'}@l(\vec{v},\vec{y})}_S (P', \mathcal{S})}$$

(REQ-IN)
$$\frac{P \xrightarrow{l':o_{\vec{t},\vec{t}'}(\vec{v}/\vec{x},\vec{y},P)@l} P'}{(P, \mathcal{S}) \xrightarrow{l':o_{\vec{t},\vec{t}'}(\vec{v},\vec{y},P)@l}_S (P', \mathcal{S}[\vec{v}/\vec{x}])}$$

(RESP-OUTLOCASYN)
$$\frac{P \xrightarrow{l':\bar{o}_{\vec{t},\vec{t}'}@l(\vec{v}/\vec{x})} P', \mathcal{S}(\vec{x}) = \vec{v}}{(P, \mathcal{S}) \xrightarrow{\tau}_S (P', \mathcal{S})}$$

(RESP-OUT)
$$\frac{P \xrightarrow{l':\bar{o}_{\vec{t},\vec{t}'}@l(\vec{v})} P'}{(P, \mathcal{S}) \xrightarrow{l':\bar{o}_{\vec{t},\vec{t}'}@l(\vec{v})}_S (P', \mathcal{S})}$$

(RESP-IN)
$$\frac{P \xrightarrow{l:o_{\vec{t},\vec{t}'}(\vec{v}/\vec{x})} P'}{(P, \mathcal{S}) \xrightarrow{l:o_{\vec{t},\vec{t}'}(\vec{v})}_S (P', \mathcal{S}[\vec{v}/\vec{x}])}$$

(ASSIGN)
$$\frac{P \xrightarrow{x:=v/e} P', e \hookrightarrow_{\mathcal{S}} v}{(P, \mathcal{S}) \xrightarrow{\tau}_S (P', \mathcal{S}[v/x])}$$

(SATISFACTION)
$$\frac{P \xrightarrow{\chi?} P', \chi \vdash \mathcal{S}}{(P, \mathcal{S}) \xrightarrow{\tau}_S (P', \mathcal{S})}$$

(NOT SATISFACTION)
$$\frac{P \xrightarrow{\neg\chi?} P', \chi \nvdash \mathcal{S}}{(P, \mathcal{S}) \xrightarrow{\tau}_S (P', \mathcal{S})}$$

Table 7.3: Rules for service engine state lts layer

$\mathtt{Internal}_S = \{\tau, \pi\}$

Rule SYNCHRO deals with internal synchronization. Rules ONE-WAYOUTASYN and ONE-WAYOUTLOCASYN enable the actions where the state contains variable whose values correspond to the values and the locations raised by the service behaviour lts layer action. Such a kind of actions are replaced with τ actions at the level of the service engine state lts layer because they are not relevant at the end of the communication. On the contrary, rule ONE-WAYOUT defines the sending of a message and its action is raised also at the overlying lts layer. All the other rules follow the same idea to enable only the actions which have the values and the locations that correspond to those stored into the state and to replace with τ actions all the actions which are not relevant to the end of the communication. It is worth noting that symbols $\chi \vdash \mathcal{S}$ and $\chi \nvdash \mathcal{S}$ denotes that the condition χ is satisfied and not satisfied within the state $\mathcal{S}$ respectively.

7.2.3 The service engine location lts layer.

We define $\rightarrow_L \subseteq Y \times Act_Y \times Y$ as the least relation which satisfies the rules of Table 7.4 where the set of actions $Act_Y = In_S \cup Out_S \cup \{\tau, \pi(l)\}$. All the rules of Table 7.4 allows us to enable to the overlying lts layer only the actions where the primitive execution location corresponds to that where the service engine is actually deployed.

7.2.4 The services system lts layer.

We define $\rightarrow_E \subseteq E \times Act_E \times E$ as the least relation which satisfies the rules of Table 7.5 closed w.r.t. the structural congruence relation $\equiv$ where the set of actions is $Act_E = Act_\sigma \cup Act_L$ and is ranged over by γ. The set Act_σ is defined as follows:

$Act_\sigma = \{(l, l', o_{\vec{t}}, \vec{v}), (l, l', o_{\vec{t},\vec{t}'}, \vec{v}, \uparrow), (l, l', o_{\vec{t},\vec{t}'}, \vec{v}, \downarrow)\}$

(SYNCHRO)
$$\frac{P_S \xrightarrow{\pi}_S P'_S}{[P_S]@l \xrightarrow{\pi(l)}_L [P'_S]@l}$$

(ONE-WAYIN)
$$\frac{P_S \xrightarrow{l:o_{\vec{t}}(\vec{v})}_S P'_S}{[P_S]@l \xrightarrow{l:o_{\vec{t}}(\vec{v})}_L [P'_S]@l}$$

(RESP-IN)
$$\frac{P_S \xrightarrow{l:o_{\vec{t},\vec{t}'}@l(\vec{v})}_S P'_S}{[P_S]@l \xrightarrow{l:o_{\vec{t},\vec{t}'}@l(\vec{v})}_L [P'_S]@l}$$

(REQ-IN)
$$\frac{P_S \xrightarrow{l':o_{\vec{t},\vec{t}'}(\vec{v},\vec{y},P)@l}_S P'_S}{[P_S]@l' \xrightarrow{l':o_{\vec{t},\vec{t}'}(\vec{v},\vec{y},P)@l}_L [P'_S]@l'}$$

(RESP-OUT)
$$\frac{P_S \xrightarrow{l':\overline{o}_{\vec{t},\vec{t}'}@l(\vec{v})}_S P'_S}{[P_S]@l' \xrightarrow{l':\overline{o}_{\vec{t},\vec{t}'}@l(\vec{v})}_L [P'_S]@l'}$$

(REQ-OUT)
$$\frac{P_S \xrightarrow{l':\overline{o}_{\vec{t},\vec{t}'}@l(\vec{v},\vec{y})}_S P'_S}{[P_S]@l' \xrightarrow{l':\overline{o}_{\vec{t},\vec{t}'}@l(\vec{v},\vec{y})}_L [P'_S]@l'}$$

(ONE-WAYOUT)
$$\frac{P_S \xrightarrow{\overline{o}_{\vec{t}}@l'(\vec{v})}_S P'_S}{[P_S]@l \xrightarrow{\overline{o}_{\vec{t}}@l'(\vec{v})}_L [P'_S]@l}$$

(SILENT)
$$\frac{P_S \xrightarrow{\tau}_S P'_S}{[P_S]@l \xrightarrow{\tau}_L [P'_S]@l}$$

Table 7.4: Rules for service engine location lts layer

$(l, l', o_{\vec{t}}, \vec{v})$, $(l, l', o_{\vec{t},\vec{t}'}, \vec{v}, \uparrow)$ and $(l, l', o_{\vec{t},\vec{t}'}, \vec{v}, \downarrow)$ are parameterized actions where l, l' are locations, o is an operation name, $\vec{t}$ and $\vec{t}'$ are operation templates, $\vec{v}$ is a vector of values and $\uparrow$ and $\downarrow$ represent the communication direction in a Request-Response message exchange: $\uparrow$ represents the request message whereas $\downarrow$ represents the response one. Rule ONE-WAYSYNC deals with the synchronization on a One-Way operation between two orchestrators whereas the rules REQ-SYNC and RESP-SYNC deal with that on a Request-Response one.

7.3 Orchestrator abstract process

This section is devoted to introduce the concept of *orchestrator abstract process*. An orchestrator abstract process is a process which describes the orchestrator without expliciting the initial values for some of the variables involved within it and where some computational aspects are omitted. In particular, an abstract process orchestrator is able to well describe the behaviour of the orchestrator as far as the communication primitives are

(ONE-WAYSYNC)

$$\frac{Y \xrightarrow{\bar{o}_{\vec{t}}@l'(\vec{v})}_{L} Y' \ , \ Z \xrightarrow{l':o_{\vec{t}}(\vec{v})}_{L} Z'}{Y \parallel Z \xrightarrow{\sigma}_{E} Y' \parallel Z'} \ , \sigma = (l, l', o_{\vec{t}}, \vec{v})$$

(REQ-SYNC)

$$\frac{Y \xrightarrow{l:\bar{o}_{\vec{t},\vec{t}'}@l'(\vec{v},\vec{y})}_{L} Y' \ , \ Z \xrightarrow{l':o_{\vec{t},\vec{t}'}(\vec{v},\vec{y},P)@l}_{L} Z'}{Y \parallel Z \xrightarrow{\sigma}_{E} Y' \parallel Z'} \ , \sigma = (l, l', o_{\vec{t},\vec{t}'}, \vec{v}, \uparrow)$$

(RESP-SYNC)

$$\frac{Y \xrightarrow{l:o_{\vec{t},\vec{t}'}@l'(\vec{v})}_{L} Y' \ , \ Z \xrightarrow{l':\bar{o}_{\vec{t},\vec{t}'}@l(\vec{v})}_{L} Z'}{Y \parallel Z \xrightarrow{\sigma}_{E} Y' \parallel Z'} \ , \sigma = (l, l', o_{\vec{t},\vec{t}'}, \vec{v}, \downarrow)$$

(INTERNAL) (SILENT)

$$\frac{Y \xrightarrow{\pi(l)}_{L} Y'}{Y \xrightarrow{\pi(l)}_{E} Y'} \qquad \frac{Y \xrightarrow{\tau}_{L} Y'}{Y \xrightarrow{\tau}_{E} Y'}$$

(PAR-EXT) (CONGRE)

$$\frac{E_1 \xrightarrow{\gamma}_{E} E_1'}{E_1 \parallel E_2 \xrightarrow{\gamma}_{E} E_1' \parallel E_2} \qquad \frac{E_1 \equiv E_1' \ , \ E_1' \xrightarrow{\gamma}_{E} E_2', \ E_2' \equiv E_2}{E_1 \xrightarrow{\gamma}_{E} E_2}$$

(STRUCTURAL CONGRUENCE OVER E)

$$E_1 \parallel E_2 \equiv E_2 \parallel E_1 \qquad E_1 \parallel (E_2 \parallel E_3) \equiv (E_1 \parallel E_2) \parallel E_3$$

Table 7.5: Rules for services sytem lts layer

concerned, but it lacks in details if we consider a specific configuration for a specific execution of that orchestrator. In particular, let us consider the following example of a supplier orchestrator which is implemented by means of a Request-Response operation:

$$\begin{aligned} S_1 \quad ::= \quad & [(ORDER_{t_2,t_2'}(\langle good_S, num_S\rangle, outcome_S, \\ & \quad outcome_S := \texttt{execute}(good_S, num_S) \\ &), \mathcal{S}_S[\perp/outcome_S])]_S \end{aligned}$$

$ORDER_{t_2,t_2'}$ is a Request-Response operation which receives values for the variables $good_S$ and num_S that represent some kind of input for the supplier orchestrator. Between the request and the response, the supplier orchestrator will calculate the result by means of the function `execute`[3] and it stores it within the variable $outocome_S$. $outocome_S$ is the variable which will be contained within the response and it is initialized to the value $\perp$. Now, let us consider the same orchestrator modified as follows:

$$\begin{aligned} S_2 \quad ::= \quad & [(ORDER_{t_2,t_2'}(\langle good_S, num_S\rangle, outcome_S, \\ & \quad \mathbf{0} \\ &), \mathcal{S}_S[\perp/outcome_S])]_S \end{aligned}$$

Differently from S_1, S_2 is not able to compute a value for the variable $outcome_S$ which is initialized to $\perp$. For this reason S_2 always replies with a message whose content is $\perp$. From an implementation point of view, although S_2 is syntactically and semantically correct, it does not represent a reasonable orchestrator to execute. But, if we consider S_2 from a system design point of view it can be considered as the abstract representation of the orchestrator S_1. S_2 indeed, models an orchestrator which receives two inputs and then replies with an output variable even if we do not know nothing about the way such a variable is calculated. S_2 could be exploited at the design time within a complete system by simply assuming some initial values for the variable $outcome_S$. For example, we can put S_2 within a system in the following way:

$$\begin{aligned} S_2 \quad ::= \quad & [(ORDER_{t_2,t_2'}(\langle good_S, num_S\rangle, outcome_S, \\ & \quad \mathbf{0} \\ &), \mathcal{S}_S[OK/outcome_S])]_S \end{aligned}$$

[3]For the sake of this example, we suppose the function `execute` is a total function whose co-domain is represented by the set $\{OK, REJECTED\}$.

In this case S_2 is modelled in a way that its answer is always OK and we can test the system where we assume a positive answer replied by the supplier orchestrator. In the case we would to test a negative reply from the supplier we can initialize the $\mathsf{outcome}_S$ variable of S_2 with the value $\mathsf{REJECTED}$.

Since the orchestrator abstract process does not introduce any syntactic or semantic modification in the orchestration language, in the following we do not formalize such a concept but we will use the term *abstract process orchestrator* for identifying those orchestrators whose behaviour is partially defined. In particular, we will use the notation $\mathcal{S}_\perp$ for referring to a state where all the variables are initialized with the value $\perp$. The orchestrator abstract processes will be exploited when we will consider conformance; in that case indeed, in order to test the conformance between an orchestrated system and a choreography, we will join choreography information with orchestrators variable and, for each possible initial state of the choreography, we will verify the conformance with the orchestrated system. Finally, a comparison between the orchestrator abstract process and the WS-BPEL abstract process (Section 2.1.3.2) can be traced. The WS-BPEL abstract process models an orchestrator that cannot be executed and it allows for the definition of a sort of skeleton of the orchestrator where internal details such as, for example, some variable values are considered *opaque* that means they can assume any value. The orchestrator abstract process is similar even if, here, we do not have any specific syntactic construct, such as the WS-BPEL opaque one, for expressing not defined parts of the orchestrator but we simply do not specify them.

7.4 Orchestration example

In the following we present an orchestrated system where there are four orchestrators which enrole the roles of the choreography presented in section 6.6: C, M, S and B where C represents the orchestrator of the customer, M the orchestrator of the market, S the orchestrator of the supplier and B that of the bank. In the following, for each orchestrator we declare its own variables and we present its code in terms of an orchestrator abstract process.

7.4.1 The customer

7.4.1.1 Customer variables

$$\mathtt{good}_C, \mathtt{num}_C, \mathtt{price}_C, \mathtt{buy}_C, \mathtt{card}_C, \mathtt{ncard}_C, \mathtt{receipt}_C, \mathtt{loc}_C, \mathtt{result}_C$$

The variable $\mathtt{good}_C$ and $\mathtt{num}_C$ contain the type of the good and the quantity to purchase respectively. For the sake of this example we will not initialize them in order to present a general orchestrator able to manage each kind of request. The initialization can be easily done by assigning some values to the variables by means of an assign primitive. The variable $\mathtt{price}_C$ will store the price purchase order price received from the market whereas the variable $\mathtt{buy}_C$ will be assigned with the values $\mathtt{accepted}$ or $\mathtt{cancelled}$ depending on the fact that the customer confirm the order or not. Variables $\mathtt{card}_C$ and $\mathtt{ncard}_C$ model the credit card data of the customer. Variable $\mathtt{receipt}_C$ models the receipt received from the bank whereas the variable $\mathtt{loc}_C$ contains the actual location of the customer. Finally, within variable $\mathtt{result}_C$ will be stored the result message that will be sent from the market for committing.

7.4.1.2 Customer orchestrator

$$\begin{array}{lll}
C & ::= & [(\mathtt{loc}_c := C; (\overline{\mathrm{PRICE}}_{t_0,t_0'}@M(\langle \mathtt{good}_C, \mathtt{num}_C\rangle, \mathtt{price}_C); \\
 & ; & (\mathtt{price}_C \geq 100 \\
 & & ?\mathtt{buy}_C := \mathtt{cancelled} \\
 & & : \mathtt{buy}_C := \mathtt{accepted}) \\
 & ; & \overline{\mathrm{BUY}}_{t_1}@M(\langle \mathtt{buy}_C, \mathtt{card}_C, \mathtt{ncard}_C, \mathtt{loc}_C\rangle)) \\
 & | & \mathrm{RESULT}_{t_3}(\mathtt{result}_C) \\
 & | & \mathrm{RECEIPT}_{t_5}(\mathtt{receipt}_C), \mathcal{S}_\perp)]_C
\end{array}$$

Initially, the customer sends a purchase order request to the market by exploiting the Solicit-Response operation $\overline{\mathrm{PRICE}}_{t_0,t_0'}@M$ and it waits for the price. If the price is greater that 100 it will not accept the order, on the contrary it will accept it. If it accepts the order, the customer will send a request to buy by exploiting the Notification $\overline{\mathrm{BUY}}_{t_1}@\mathtt{M}$ where it sends also its location ($\mathtt{loc}_C$) for allowing the bank service to send the receipt. Finally, the customer will wait for the bank receipt and the market commit message on the One-Way operations RESULT_{t_3} and $\mathrm{RECEIPT}_{t_5}$, respectively.

7.4.2 The market

7.4.2.1 Market variables

$$\mathtt{good}_M, \mathtt{num}_M, \mathtt{price}_M, \mathtt{buy}_M, \mathtt{card}_M, \mathtt{ncard}_M, \mathtt{receipt}_M, \mathtt{loc}_M, \mathtt{outcome}_M$$

The variables $\mathtt{good}_M$, $\mathtt{num}_M$, $\mathtt{buy}_M$, $\mathtt{card}_M$, $\mathtt{ncard}_M$, $\mathtt{receipt}_M$ and $\mathtt{loc}_M$ will contain the values communicated by the customer and stored within the customer variables that have the same names. The variable $\mathtt{price}_M$ models the price fro the requested order. The variable $\mathtt{outcome}_M$ will store the result message from the supplier and it abstractly models the fact that the supplier is able or not to fulfill the order.

7.4.2.2 Market orchestrator

$$\begin{array}{lll}
\mathtt{M} & ::= & [(\mathrm{PRICE}_{t_0,t'_0}(\langle \mathtt{good}_M, \mathtt{num}_M\rangle, \mathtt{price}_M, \mathbf{0}) \\
 & \mid & (\mathrm{BUY}_{t_1}(\langle \mathtt{buy}_M, \mathtt{card}_M, \mathtt{ncard}_M, \mathtt{loc}_M\rangle); \mathtt{buy}_M == \mathtt{accepted}?\mathtt{Ord} : \mathbf{0}), \mathcal{S}_\perp)]_M
\end{array}$$

$$\begin{array}{lll}
\mathtt{Ord} & ::= & \overline{\mathrm{ORDER}}_{t_2,t'_2}@\mathtt{S}(\langle \mathtt{good}_M, \mathtt{num}_M\rangle, \mathtt{outcome}_M) \\
 & ; & (\overline{\mathrm{RESULT}}_{t_3}@\mathtt{loc}_M(\mathtt{outcome}_M) \\
 & & \;\mid \mathtt{outcome}_M == \mathtt{OK}?\overline{\mathrm{PAY}}_{t_4}@\mathtt{B}(\langle \mathtt{card}_M, \mathtt{ncard}_M, \mathtt{price}_M, \mathtt{loc}_M\rangle) : \mathbf{0} \\
 & &)
\end{array}$$

Initially, the market is waiting for a price request from a customer on the Request-Response operation $\mathrm{PRICE}_{t_0,t'_0}$. When invoked, the orchestrator replies by sending the price for the given order. After that, it waits for a confirmation or a cancellation of the order on the One-Way BUY_{t_1}. The customer response is stored within the variable $\mathtt{buy}_M$ and if it is equal to $\mathtt{accept}$ the process $\mathtt{Ord}$ is executed. Such a kind of process forwards the order to the supplier by exploiting the Solicit-Response $\overline{\mathrm{ORDER}}_{t_2,t'_2}@\mathtt{S}$ and waits for a response from the supplier which will be stored within the variable $\mathtt{outcome}_M$. If $\mathtt{outcome}_M$ is equal to OK, that models a successful termination of the order, the market will invoke the bank by means of the Notification $\overline{\mathrm{PAY}}_{t_4}@\mathrm{B}$ in order to start a financial transaction, otherwise it terminates its execution. Moreover, the market sends the value of the variable $\mathtt{outcome}_M$ to the customer by means of the Notification $\overline{\mathrm{RESULT}}_{t_3}@\mathtt{loc}_M$. It is worth noting that the location of the customer is contained within the variable $\mathtt{loc}_M$ whose value is received during the execution of the One-Way BUY_{t_1}.

7.4.3 The supplier

7.4.3.1 Supplier variable

$$\mathtt{good}_S, \mathtt{num}_S, \mathtt{outcome}_S$$

The variables $\mathtt{good}_S$ and $\mathtt{num}_S$ model the data related to the order whereas the variable $\mathtt{outcome}_S$ models the state of the order that is, if the order can be fulfilled or not.

7.4.3.2 Supplier orchestrator

$$S \quad ::= \quad [(\mathrm{ORDER}_{t_2,t_2'}(\langle \mathtt{good}_S, \mathtt{num}_S\rangle, \mathtt{outcome}_S, \mathbf{0}), \mathcal{S}_\perp)]_S$$

The code of the supplier is very simple and it is represented by the Request-Response $\mathrm{ORDER}_{t_2,t_2'}$ which receives the order data and, as a response, sends the variable $\mathtt{outcome}_S$.

7.4.4 The bank

7.4.4.1 Bank variables

$$\mathtt{card}_B, \mathtt{ncard}_B, \mathtt{price}_B, \mathtt{loc}_B, \mathtt{receipt}_B$$

The variables $\mathtt{card}_B$ and $\mathtt{ncard}_B$ model the credit card data of the bank account of the customer on which the financial transaction must be performed. The variable $\mathtt{price}_B$ contains the amount of money whereas the variable $\mathtt{loc}_B$ contains the location of the customer to which the receipt must be sent. Finally, the variable $\mathtt{receipt}_B$ abstractly models the receipt message.

7.4.4.2 Bank orchestrator

$$\begin{array}{lll} B & ::= & [(\mathrm{PAY}_{t_4}(\langle \mathtt{card}_B, \mathtt{ncard}_B, \mathtt{price}_B, \mathtt{loc}_B \rangle) \\ & ; & \overline{\mathrm{RECEIPT}}_{t_5}@\mathtt{loc}_B(\mathtt{receipt}_B), \mathcal{S}_\perp)]_B \end{array}$$

Initially, the bank orchestrator waits for an invocation on the One-Way operation PAY_{t_4}, after that it sends the message receipt, to the customer located at the received location stored within the variable $\mathtt{loc}_B$, by means of the Notification $\overline{\mathrm{RECEIPT}}_{t_5}@\mathtt{loc}_B$.

7.4.5 The system

The system is described by the following process E where the four orchestrators are composed in parallel:

$$E ::= C \parallel M \parallel S \parallel B$$

Chapter 8

Conformance

The conformance is a mathematical relation which allows us to formally relate the choreography semantics with the orchestration one. As we have shown in the previous chapters, the choreography and orchestration semantics are expressed in terms of labelled transition systems where transitions represent a message exchange between two roles or some internal or silent actions. The conformance aims at stating if all the interactions and the internal actions expressed by a choreography are actually performed by the orchestrated system. The following concepts are fundamental to the end of the conformance between a choreography and an orchestrated system:

- *A role in choreography can be enroled by one or more orchestrators into the orchestrated system.* We join each orchestrator of an orchestrated system with one or any role of a choreography by means of a *joining function*, named Ψ, which allows us to associate the orchestrators of an orchestrated system E to the roles of a choreography $\mathcal{C}$. Such a fact implies that more than one orchestrator can enrole a choreography role and that there could be orchestrators that are not joined with any role, we call this kind of orchestrators *coordinators*. As an example, let us remind the case of the choreography reported in section 6.6 where there are four roles: the customer, the market, the supplier and the bank, and, furthermore, let us consider the orchestration example of section 7.4 where we have designed an orchestrated system where there is an orchestrator for each role. In the following, we will show that there exists a joining function which join each orchestrator to each choreography role so that the orchestrated system is conformant to the choreography. Depending on the joining function, different orchestrated systems can be conformant to the same choreography. For instance, the orchestrated system of the example above, could be modified by introducing some new orchestrators. In Fig. 8.1 we abstractly show the communication links of an orchestrated system where there are more than one orchestrator which enrole both the supplier and the bank roles. Also in this case, it is possible to construct a joining function in a way that the orchestrated system is conformant to the choreography. Such a joining function joins the orchestrator C with the customer role, the orchestrator M with the market role, the orchestrators S_1, S_2 and S_3 with the supplier role and the orchestrators B_1, B_2 and B_3 with the bank role. This example will be deeply discussed in the following.

- *The conformance is always tested up to a joining function.*

- *The conformance is based upon the conformability relation.* The *conformability* is a relation, which resembles a bisimulation one, between a labelled transition system of the choreography and a *joined labelled transition system* of the orchestrated system. The joined labelled transition system is the labelled transition system of the orchestrated system where the joining function is applied in order to rename all the interactions among the orchestrators. In particular, the renaming distinguishes among the following interactions types:

  – The *observable interactions*: they represent all the message exchanges among orchestrators joined with different choreography roles.

  – The *internal interactions*: they represent both the message exchanges between orchestrators joined with the same role and internal synchronizations within an orchestrator. They will be labelled with π actions.

  – The *coordinating interactions*: they represent all the interactions which are not considered in the choreography and are exploited only for coordinating the orchestrators joined with a role. For example all the interactions where at least one of the involved orchestrators is a coordinator. To the end of conformance they will be labelled with τ actions as the internal computational actions.

- *The conformance consists of a conformability relation test for each different initial state of the choreography.* In other words, we will generate a labelled transition system for each initial state of the choreography coherent with its initial constraints and, for each of them, we will consider the correspondant joined labelled transition system then, we will test the conformability relation for each lts couple. More formally, given a choreography $\mathcal{C} = (\Sigma, C, \mathcal{K}, X)$ an orchestrated system E and a joining function Ψ, the idea is to consider all the possible choreography states which satisfy the initial constraint X and for each of them test, up to Ψ, the conformability ($\rhd_\Psi$) between the labelled transition system of the choreography and the *joined labelled transition system* of the orchestrated system. Such a condition implies that the initial values of the choreography information must be joined with the initial values of the variables of the orchestrators. We will exploit the joining function both for joining orchestrators to roles and orchestrator variables to choreography information.

8.1 The joining function

In the following we present the definition of the joining function which is composed by two components where the former associates orchestrators (identified by their location) to roles and the latter associates the orchestrator variables to the choreography information.

Definition 8.1 (joining functions) *A joining function is an element of the set*

$$\{\Psi \mid \Psi : \mathsf{Loc} \rightarrow \mathsf{RName} \cup \{\perp\} \times (\mathsf{Var} \rightarrow I_C \cup \{\perp\})\}$$

containing functions which associate to each orchestrator location a pair composed of a choreography role (or the $\perp$ value in case no role is associated) and a function from orchestrator variables

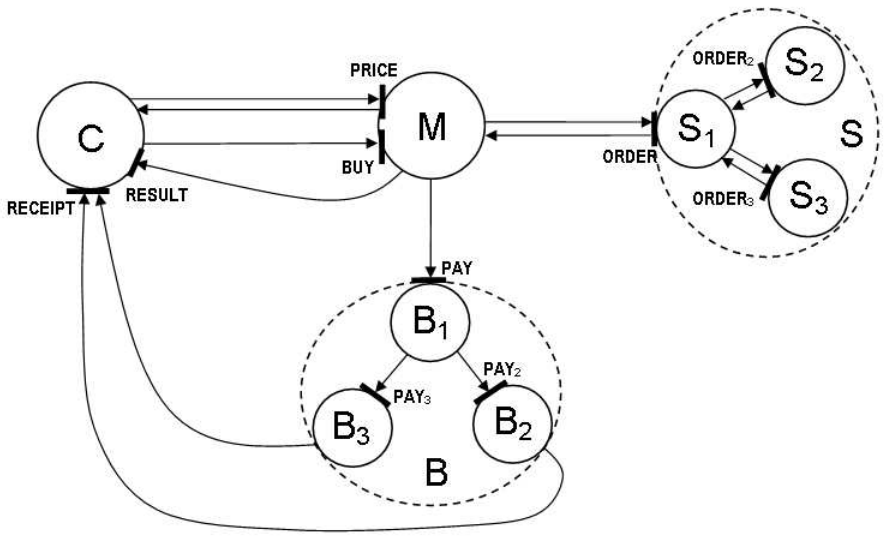

Figure 8.1: More orchestrators can enrole a choreography role

to choreography information (or the $\perp$ value in case no variable is associated). We denote with Ψ^1 the projection of Ψ on the first element of the pair (the associated role) and with Ψ^2 the projection on the second element (the variable mapping function).

Given a joining function Ψ and an action $\sigma = (l, l', o_{\vec{t}}, \vec{v})$ or an action $\sigma' = (l, l', o_{\vec{t},\vec{t}'}, \vec{v}, dir)$ of a given orchestrated system where l and l' are orchestrator locations, o is an operation name, $\vec{t}$ and $\vec{t}'$ are templates, $\vec{v}$ is a vector of values and locations and $dir \in \{\uparrow, \downarrow\}$, we denote with

$$\Psi[\sigma] = (\Psi^1(l), \Psi^1(l'), o_{\vec{t}}, \vec{v})$$

$$\Psi[\sigma'] = (\Psi^1(l), \Psi^1(l'), o_{\vec{t},\vec{t}'}, \vec{v}, dir)$$

the renaming of the orchestrator locations with the joined roles. The projection Ψ^2 will be exploited for joining the initial values of the choreography information to the related ones of the orchestrated system. Given a choreography lts label μ, an orchestrated system lts label σ and a joining function Ψ we say that $\Psi[\sigma] \geq \mu$ if the following conditiond hold:

- $\mu = (\rho_A, \rho_B, o_{\vec{t}}, \vec{w}) \wedge \sigma = (l, l', r_{\vec{u}}, \vec{v})$
  - $\Psi^1(l) = \rho_A \wedge \Psi^1(l') = \rho_B$

  - $o = r$
  - $\vec{t} = \vec{u}$
  - $\forall t_i, ((t_i = inf \Rightarrow w_i = v_i) \vee (t_i = loc \Rightarrow \Psi^1(v_i) = w_i))$

- $\mu = (\rho_A, \rho_B, o_{\vec{t},\vec{t}'}, \vec{w}, dir) \wedge \sigma = (l, l', r_{\vec{u},\vec{u}'}, \vec{v}, dir')$
  - $\Psi^1(l) = \rho_A \wedge \Psi^1(l') = \rho_B$
  - $o = r$
  - $\vec{t} = \vec{u} \wedge \vec{t}' = \vec{u}'$
  - $\forall t_i, ((t_i = inf \Rightarrow w_i = v_i) \vee (t_i = loc \Rightarrow \Psi^1(v_i) = w_i))$
  - $dir = dir'$

It is worth noting that the communicated values in a choreography interaction must be equal to those communicated into the related interaction in the orchestrated system where locations must correspond to role names.

8.2 The joined labelled transition system

In the following we present the definition of the joined labelled transition system where, in this new transition system, the initial values of the variables of the orchestrated system are joined with the choreography ones up to Ψ^2. Furthermore, some hiding operators are applied to the orchestrated system in order to observe only those interactions which are relevant for the choreography. Hiding consists of replacing labels with τ or with a label $\pi(l)$ and it is applied to the following actions:

- Actions related to internal interactions are replaced with the label $\pi(l)$ where l is the location of the sender.
  - the interactions which are performed between orchestrators joined with the same role.
- Actions related to coordinating interactions are replaced with the label τ:

- the interactions that involve an orchestrator not joined with any role
- the interactions performed on operations not declared in the choreography. It is also the case of interactions performed within two orchestrators joined with different roles but on an operation not considered in choreography.

Definition 8.2 (Joined labelled transition system) *Given a choreography $\mathcal{C} = (\Sigma, C, \mathcal{K}_0, X)$, an orchestrated system E and a joining function Ψ such that $Im^1(\Psi) = \Sigma \cup \{\perp\}$*[1]*, let $\omega_{\mathcal{C}}$ be the set of operations involved within the choreography $\mathcal{C}$, let ω_E be the set of operations exhibited by the processes of E and let $E_{OP} = \omega_E / \omega_{\mathcal{C}}$ be the set of operations exhibited by E and which do not appear within the roles of $\mathcal{C}$. Let $E_\perp$ be the set of orchestrator locations l of E for which $\Psi^1(l) = \perp$. Let $\gamma \in \gamma_X$. We denote the joined labelled transition system with:*

$$E^\frown\Psi^2[\gamma]/E_\rho//E_\perp///E_{OP}$$

where:

- $E^\frown\Psi^2[\gamma]$ *is an operator which associates the values of the choreography information in γ to the corresponding variables in the states of E up to the joining function Ψ^2. Formally let $\vec{x}_l$ and $\vec{y}_l$ be the vectors of variables in Var which belong to the state of the orchestrator located at l and for which the following conditions hold respectively: $\Psi^2(l)(\vec{x}_l) \neq \perp$, $\Psi^2(l)(\vec{y}_l) = \perp$ and let $\vec{v}_l$ be the vector of values of the choreography information joined with the variables $\vec{x}_l$ that is $\vec{v}_l = \gamma(\Psi^2(l)(\vec{x}_l))$. We have that the $E^\frown\Psi^2[\gamma]$ is inductively defined as follows:*

  - $[P, \mathcal{S}]_l^\frown\Psi^2[\gamma] = [P, \mathcal{S}[\vec{v}_l/\vec{x}_l, \perp/\vec{y}_l]]_l$
  - $E^\frown\Psi^2[\gamma] = [P, \mathcal{S}[\vec{v}_l/\vec{x}_l, \perp/\vec{y}_l]]_l \parallel E'^\frown\Psi^2[\gamma]$

- $/E_\rho$ *hides, replacing with $\pi(l)$ moves, all the interactions between the same role (the $\Psi^1(l)$ of the sender located at a location l corresponds to the role $\Psi^1(l')$ joined with the receiver located at a location l').*

[1] $Im^1(\Psi) = \{\Psi^1(l) \mid l \in Loc\}$

- *$//E_{\perp}$ is a hiding operator which hides, replacing with τ moves, all the transitions which contain orchestrators not joined with any role.*
- *$///E_{OP}$ is a hiding operator which hides, replacing with τ moves, all the transitions which contain operations contained in E_{OP}*

8.3 Conformability

In the following we present the conformability relation between a labelled transition system of a choreography and a joined labelled transition system of an orchestrated system. Conformability is inspired by bisimulation [Mil89] but some differences exist. In particular, the conformability states if for each interaction μ within the choreography there exists a corresponding one (σ) into the orchestrated one. Furthermore, it states if for each internal action in the choreography $\pi(\rho)$ there is an internal action into the orchestrated system ($\pi(l)$) even if, such a condition is not symmetric in the sense that it is not relevant to the end of conformability that, for each internal action performed at the level of the orchestrated system, there exists an internal action at the level of choreography. Such a condition allows us, on the one hand, to test if the internal actions declared within the choreography are performed within the orchestrated system and, on the other hand, to avoid a strict limit on the performed internal actions on the orchestration side. Indeed, in a system with several orchestrators joined with the same roles, the internal actions could be more than those defined within the choreography. Here, it is important to verify that the orchestrated system performs at least the internal actions declared within the choreography. Finally, the τ actions are not relevant to the end of the conformability and they are ignored. It is worth noting that here, we are not interested to distinguish between deadlock and termination states both in choreography and orchestration which are related in conformability by introducing the set of states $C_\delta(\mathcal{K}, \gamma)$ and E_δ defined in the following.

Definition 8.3 (Conformability) *Let Ψ be a joining function. A relation $\mathcal{R}_\Psi \subseteq ((C_L, K_C, \Gamma_C) \times OL)$ is a conformability relation if $((C, \mathcal{K}, \gamma), E) \in \mathcal{R}_\Psi$ implies that $C \in C_\delta(\mathcal{K}, \gamma)$ and $E \in E_\delta$ or,* *for all* $\mu \in \mathrm{Act}_C$ *and for all* $\sigma \in \mathrm{Act}_E$*, the following conditions hold:*

1. $(C, \mathcal{K}, \gamma) \overset{\mu}{\Rightarrow} (C', \mathcal{K}', \gamma') \Rightarrow E \overset{\sigma}{\Rightarrow} E' \wedge ((C', \mathcal{K}', \gamma'), E') \in \mathcal{R}_\Psi \wedge \Psi^1[\sigma] \geq \mu$

2. $(C, \mathcal{K}, \gamma) \overset{\pi(\rho)}{\Rightarrow} (C', \mathcal{K}', \gamma') \Rightarrow E \overset{\pi(l)}{\Rightarrow} E' \wedge \Psi^1(l) = \rho \wedge ((C', \mathcal{K}', \gamma'), E') \in \mathcal{R}_\Psi$

3. $E \overset{\sigma}{\Rightarrow} E' \Rightarrow (C, \mathcal{K}, \gamma) \overset{\mu}{\Rightarrow} (C', \mathcal{K}', \gamma') \wedge ((C', \mathcal{K}', \gamma'), E') \in \mathcal{R}_\Psi \wedge \Psi^1[\sigma] \geq \mu$

4. $E \overset{\pi(l)}{\Rightarrow} E' \Rightarrow ((C, \mathcal{K}, \gamma), E') \in \mathcal{R}_\Psi \vee ((C, \mathcal{K}, \gamma) \overset{\pi(\rho)}{\Rightarrow} (C', \mathcal{K}', \gamma') \wedge ((C', \mathcal{K}', \gamma'), E') \in \mathcal{R}_\Psi \wedge \Psi^1(l) = \rho)$

where the arrow $\overset{\gamma}{\Rightarrow}$ *means the concatenation of the following transitions:* $\overset{\tau^*}{\rightarrow}\overset{\gamma}{\rightarrow}\overset{\tau^*}{\rightarrow}$ *and* $C_\delta(\mathcal{K}, \gamma)$ *and* E_δ *are defined as follows:*

- $C_\delta(\mathcal{K}, \gamma) = \{C \in C_L \mid \forall C' \in C_L \nexists \nu, \mathcal{K}', \gamma'.(C, \mathcal{K}, \gamma) \overset{\nu}{\rightarrow} (C', \mathcal{K}', \gamma')\}$
- $E_\delta = \{E \in OL \mid \nexists E' \in OL, l \in \mathrm{Loc}\ .E \overset{\sigma}{\rightarrow} E' \vee E \overset{\pi(l)}{\rightarrow} E' \vee E \overset{\tau}{\rightarrow} E'\}$

We write $(C, \mathcal{K}, \gamma) \triangleright_\Psi E$ *if there exists a conformability relation* $\mathcal{R}_\Psi$ *such that* $((C, \mathcal{K}, \gamma), E) \in \mathcal{R}_\Psi$*.*

8.4 Conformance

In this section we present the formal definition of the conformance between a choreography and an orchestrated system. Given a choreography $\mathcal{C} = (\Sigma, C, \mathcal{K}, X)$ an orchestrated system E and a joining function Ψ, the idea is to consider all the possible choreography states which satisfy the initial constraint X and for each of them test, up to Ψ, the conformability ($\triangleright_\Psi$) between the labelled transition system of the choreography and the *joined labelled transition system* of the orchestrated system.

Definition 8.4 (Conformance) *Given a choreography $\mathcal{C} = (\Sigma, C, \mathcal{K}_0, X)$, an orchestrated system $E \in OL$ and a joining function Ψ such that $Im^1(\Psi) = \Sigma \cup \{\perp\}$, let ω_C be the set of operations involved within the choreography $\mathcal{C}$, let ω_E be the set of operations exhibited by the processes of E and let $E_{OP} = \omega_E/\omega_C$ be the set of operations exhibited by E and which do not appear within the roles of $\mathcal{C}$. Let $E_\perp$ be the set of orchestrator identifiers id of E for which $\Psi^1(l) = \perp$. We say that E is conformant to $\mathcal{C}$ if the following condition holds:*

$$\forall \gamma \in \gamma_X, (C, \mathcal{K}, \gamma) \rhd_\Psi E^\frown \Psi^2[\gamma]/E_\rho /\!/ E_\perp /\!/\!/ E_{OP}$$

Observe that on the right hand side of $\rhd_\Psi$ the joined labelled transition system of the orchestrated system defined in Definition 8.2 is considered.

8.5 Examples

In this section we report four examples, A, B, C and D, in order to show how conformance works. As far as the examples A, B and C are concerned, we always consider the same choreography $Chor$ of Section 6.6 in order to show how different orchestrated systems can be conformant to the same choreography depending on the selected joining function, whereas we will exploit example D for showing how non-deterministic choice in orchestration does not strictly correspond to the non-deterministic choice in choreography.

8.5.1 Example A

Here we consider the choreography presented in Section 6.6 and the orchestrated system presented in Section 7.4 where in the former there are four roles defined, the customer (ρ_C), the market (ρ_M), the supplier (ρ_S) and the bank (ρ_B) whereas in the latter, there are four orchestrators defined, C, M, S and B. In order to test the conformance between that choreography and that orchestrated system, we consider a joining function Ψ_A whose projection Ψ_A^1 allows us to join the orchestrators C, M, S and B to the roles ρ_C, ρ_M, ρ_S and ρ_B, respectively. Ψ_A^1 is defined as it follows:

$\Psi^1_A(C) = \rho_C,\ \Psi^1_A(M) = \rho_M,\ \Psi^1_A(S) = \rho_S,\ \Psi^1_A(B) = \rho_B,$

$\Psi^1_A(l) = \perp$ for $l \notin \{C, M, S, B\}$

We remind that the information set defined within the choreography is:

$$I_C = \{good, num, price, buy, card, ncard, outcome, receipt\}$$

and that the initial constraints are defined in the following way:

$$\begin{array}{rcl} X & = & good \in \{apple, banana, strawberry\} \\ & \wedge & 0 \leq num \leq 200 \\ & \wedge & card \in \{visa, mastercard\} \\ & \wedge & ncard = 123456789 \\ & \wedge & buy = \perp \\ & \wedge & 50 \leq price \leq 200 \\ & \wedge & outcome \in \{OK, REJECTED\} \\ & \wedge & receipt = receiptDoc \end{array}$$

To the end of conformance, we also define the joining function projection Ψ^2_A which joins the orchestrated system variables to the choreography information as it follows:

$\Psi^2_A(C)(good_C) = good$

$\Psi^2_A(C)(num_C) = num$

$\Psi^2_A(M)(price_M) = price$

$\Psi^2_A(C)(buy_C) = buy$

$\Psi^2_A(C)(card_C) = card$

$\Psi^2_A(C)(ncard_C) = ncard$

$\Psi^2_A(S)(outcome_S) = outcome$

$\Psi^2_A(B)(receipt_B) = receipt$

In Fig. 8.2 and 8.3 we have reported the labelled transition system of the choreography and the orchestrated system respectively, generated starting from an initial state where the information have the following values:

$good = apple$

$num = 10$

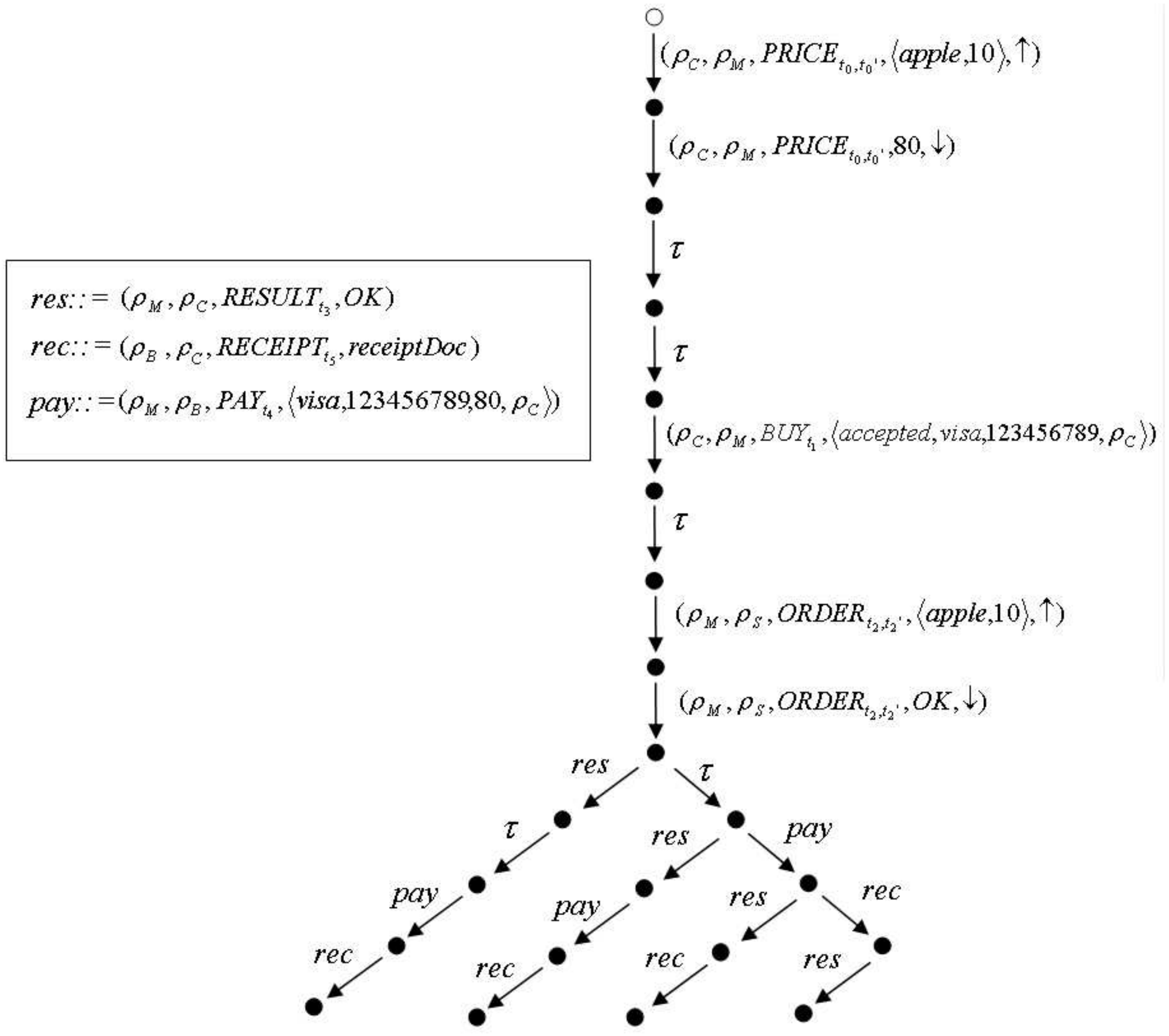
$res ::= (\rho_M, \rho_C, RESULT_{t_3}, OK)$
$rec ::= (\rho_B, \rho_C, RECEIPT_{t_5}, receiptDoc)$
$pay ::= (\rho_M, \rho_B, PAY_{t_4}, \langle visa, 123456789, 80, \rho_C \rangle)$
$(\rho_C, \rho_M, PRICE_{t_0, t_0'}, \langle apple, 10 \rangle, \uparrow)$
$(\rho_C, \rho_M, PRICE_{t_0, t_0'}, 80, \downarrow)$
τ
τ
$(\rho_C, \rho_M, BUY_{t_1}, \langle accepted, visa, 123456789, \rho_C \rangle)$
τ
$(\rho_M, \rho_S, ORDER_{t_2, t_2'}, \langle apple, 10 \rangle, \uparrow)$
$(\rho_M, \rho_S, ORDER_{t_2, t_2'}, OK, \downarrow)$
res
τ
τ
res
pay
pay
pay
res
rec
rec
rec
rec
res

Figure 8.2: Choreography labelled transition system

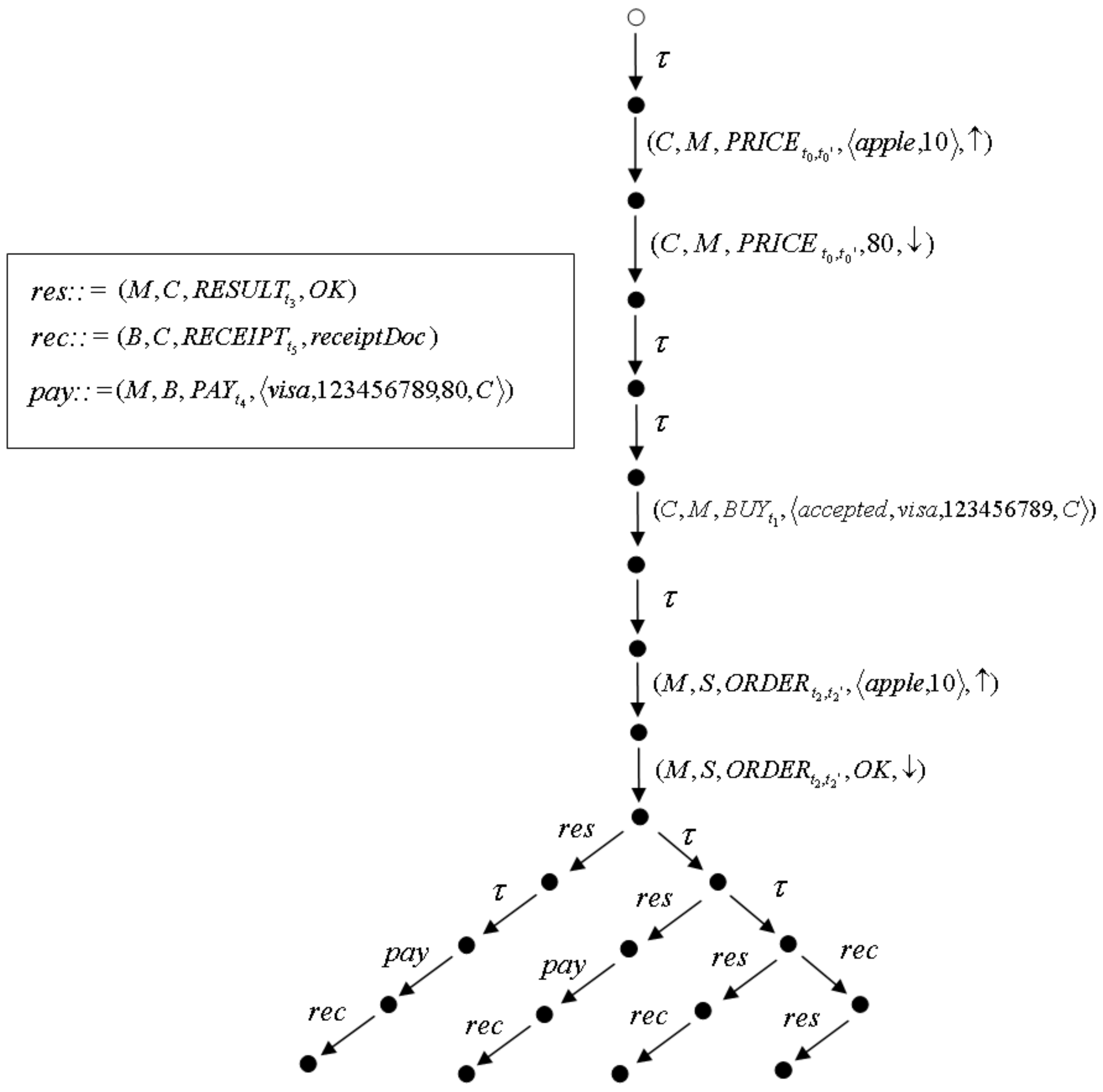
$res ::= (M, C, RESULT_{t_3}, OK)$
$rec ::= (B, C, RECEIPT_{t_5}, receiptDoc)$
$pay ::= (M, B, PAY_{t_4}, \langle visa, 123456789, 80, C \rangle)$
τ
$(C, M, PRICE_{t_0, t_0'}, \langle apple, 10 \rangle, \uparrow)$
$(C, M, PRICE_{t_0, t_0'}, 80, \downarrow)$
τ
τ
$(C, M, BUY_{t_1}, \langle accepted, visa, 123456789, C \rangle)$
τ
$(M, S, ORDER_{t_2, t_2'}, \langle apple, 10 \rangle, \uparrow)$
$(M, S, ORDER_{t_2, t_2'}, OK, \downarrow)$
res
τ
τ
res
τ
pay
pay
res
rec
rec
rec
rec
res

Figure 8.3: Orchestrated system labelled transition system

$price = 80$

$buy = \perp$

$card = visa$

$ncard = 123456789$

$outcome = accepted$

$receipt = receiptDoc$

The two labelled transition systems satisfy the conformability relation. Furthermore, for each initial state of the choreography the labelled transition systems of the choreography and those of the orchestrated system satisfy the conformability relation thus allowing us to state that the orchestrated system of Section 7.4 is conformant to the choreography of Section 6.6 up to the considered joining function Ψ_A. It is worth noting that an explicit location mobility is exploited within the example. The bank indeed, needs to know the location of the customer in order to send the receipt. Here, we want to notice that:

$$\Psi[(C, M, BUY_{t_1}, \langle accepted, visa, 1223456789, C\rangle)] \geq \\ (\rho_C, \rho_M, BUY_{t_1}, \langle accepted, visa, 1223456789, \rho_C\rangle)$$

Roles ρ_C and ρ_M indeed, are joined with orchestrators C and M and the operation BUY_{t_1} and the exchanged values are the same in both interactions. It is worth noting that the fourth exchanged value is a location ρ_C and, by the definition we gave of $\Psi[\sigma] \geq \mu$ in Section 8.1, it must be compared with the role joined with orchestrator C which is ρ_C.

8.5.2 Example B

Here we present an orchestrated system where some roles are joined with more than one orchestrator. In particular, we consider an orchestrated system which follows the communication links represented in Fig. 8.1 where there are three orchestrators which enrole the supplier role (S_1, S_2 and S_3) and three orchestrators which enrole the bank one (B_1, B_2 and B_3). The orchestrator S_1 receive an order request on the operation $ORDER_{t_2,t_2'}$ and then, depending on the good, it forwards the request to S_2 or S_3. The orchestrator B_1 receives a request payment on the operation PAY_{t_4} and then, depending on the card type, it forwards the request to the orchestrator B_2 or B_3. B_2 and B_3 will send the receipt to the customer. In the following we report the orchestrated system code:

$$
\begin{array}{rcl}
E_B & ::= & C \parallel M \parallel S_1 \parallel S_2 \parallel S_3 \parallel B_1 \parallel B_2 \parallel B_3 \\
\\
C & ::= & [(loc_c := C; (\overline{PRICE}_{t_0,t_0'}@M(\langle good_C, num_C\rangle, price_C); \\
; & & (price_C \geq 100 \\
& & ?buy_C := cancelled \\
& & : buy_C := accepted) \\
; & & \overline{BUY}_{t_1}@M(\langle buy_C, card_C, ncard_C, loc_C\rangle)) \\
\mid & & RESULT_{t_3}(result_C) \\
\mid & & RECEIPT_{t_5}(receipt_C), \mathcal{S}_\perp)]_C \\
\\
M & ::= & [(PRICE_{t_0,t_0'}(\langle good_M, num_M\rangle, price_M, \mathbf{0}) \\
\mid & & (BUY_{t_1}(\langle buy_M, card_M, ncard_M, loc_M\rangle) \\
; & & buy_M == accepted?Ord : \mathbf{0}), \mathcal{S}_\perp)]_M \\
Ord & ::= & \overline{ORDER}_{t_2,t_2'}@S(\langle good_M, num_M\rangle, outcome_M) \\
; & & (\overline{RESULT}_{t_3}@loc_M(outcome_M) \\
& & \mid outcome_M == OK \\
& & ?\ \overline{PAY}_{t_4}@B(\langle card_M, ncard_M, price_M, loc_M\rangle) : \mathbf{0} \\
& &) \\
\\
S_1 & ::= & [(ORDER_{t_2,t_2'}(\langle good_{S1}, num_{S1}\rangle, outcome_{S1}, SelS), \mathcal{S}_\perp)]_{S1} \\
SelS & ::= & good_{S1} == apple \\
& & ?\ \overline{ORDER2}_{t_2,t_2'}@S_2(\langle good_{S1}, num_{S1}\rangle, outcome_{S1}) \\
& & :\ \overline{ORDER3}_{t_2,t_2'}@S_3(\langle good_{S1}, num_{S1}\rangle, outcome_{S1}) \\
\\
S_2 & ::= & [(ORDER2_{t_2,t_2'}(\langle good_{S2}, num_{S2}\rangle, outcome_{S2}, \mathbf{0}), \mathcal{S}_\perp)]_{S2} \\
S_3 & ::= & [(ORDER3_{t_2,t_2'}(\langle good_{S3}, num_{S3}\rangle, outcome_{S3}, \mathbf{0}), \mathcal{S}_\perp)]_{S2}
\end{array}
$$

$$\begin{array}{lll}
B_1 & ::= & [(PAY_{t_4}(\langle card_{B1}, ncard_{B1}, price_{B1}, loc_{B1}\rangle); \\
 & ; & (card_{B1} = visa \\
 & & \;?\; \overline{PAY2}_{t_4}@B_2(\langle card_{B1}, ncard_{B1}, price_{B1}, loc_{B1}\rangle) : \mathbf{0} \\
 & ; & card_{B1} = mastercard \\
 & & \;?\; \overline{PAY3}_{t_4}@B_3(\langle card_{B1}, ncard_{B1}, price_{B1}, loc_{B1}\rangle))), \mathcal{S}_\perp)]_{B1}
\end{array}$$

$$\begin{array}{lll}
B_2 & ::= & [(PAY2_{t_4}(\langle card_{B2}, ncard_{B2}, price_{B2}, loc_{B2}\rangle) \\
 & ; & \overline{RECEIPT}_{t_5}@loc_{B2}(receipt_{B2}), \mathcal{S}_\perp)]_{B2} \\
B_3 & ::= & [(PAY3_{t_4}(\langle card_{B3}, ncard_{B3}, price_{B3}, loc_{B3}\rangle) \\
 & ; & \overline{RECEIPT}_{t_5}@loc_{B3}(receipt_{B3}), \mathcal{S}_\perp)]_{B3}
\end{array}$$

Now, let us test the conformance between the choreography of Section 6.6 and the orchestrated system above, by exploiting the joining function Ψ_A defined in the example A (8.5.1). The conformance test states that this orchestrated system is not conformant to the choreography up to the joining function Ψ_A. Indeed, in Ψ_A the orchestrators S_1, S_2, B_1 and B_2 are not joined with any role ($\Psi^1_A(S_1) = \Psi^1_A(S_2) = \Psi^1_A(B_1) = \Psi^1_A(B_2) = \perp$) thus, the joined labelled transition system does not include the interactions between the orchestrators of the bank and the customer which are all replaced with τ actions. In this orchestrated system indeed, the interactions between the bank role and the customer role are performed by the orchestrators B_2 and B_3 on the operation RECEIPT but they are not joined with any role and the interactions they performed are hidden by the operator $//E_\perp$ so the orchestrated system E_B is not conformant to the choreography up to Ψ_A because the choreography interaction $(\rho_B, \rho_C, RECEIPT_{t_5}, receipt)$ never matches with any orchestrated interaction. Now, let us consider the following joining function Ψ_B where orchestrators S_2 and S_3 are joined with role S and orchestrators B_2 and B_3 are joined with role B:

$\Psi^1_B(C) = \rho_C$
$\Psi^1_B(M) = \rho_M$
$\Psi^1_B(S_1) = \rho_S, \Psi^1_B(S_2) = \rho_S, \Psi^1_B(S_3) = \rho_S$

$\Psi^1_B(B_1) = \rho_B, \Psi^1_B(B_2) = \rho_B, \Psi^1_B(B_3) = \rho_B$

$\Psi^1_B(l) = \bot$ for $l \notin \{C, M, S_1, S_2, S_3, B_1, B_2, B_3\}$.

$\Psi^2_B(C)(buy_C) = buy$

$\Psi^2_B(C)(good_C) = good$

$\Psi^2_B(C)(num_C) = num$

$\Psi^2_B(C)(card_C) = card$

$\Psi^2_B(C)(ncard_C) = ncard$

$\Psi^2_B(M)(price_M) = price$

$\Psi^2_B(S_2)(outcome_{S2}) = outcome$

$\Psi^2_B(S_3)(outcome_{S3}) = outcome$

$\Psi^2_B(B_2)(receipt_{B2}) = receipt$

$\Psi^2_B(B_3)(receipt_{B3}) = receipt$

It descends that the orchestrated system E_B is conformant to the choreography $Chor$ up to Ψ_B. In this case indeed, the interactions between B_2 (or B_3) and the customer are not hidden because the bank orchestrators are joined with the same role S. It is worth noting that the message exchanges on the operations ORDER2$_{t_2,t_2'}$ and ORDER3$_{t_2,t_2'}$, in the joined labelled transition system, are renamed with the label $\pi(S)$ by applying the operator $/E_\rho$. Analagously, the message exchanges on the operations PAY2$_{t_4}$ and PAY3$_{t_4}$ are renamed with the label $\pi(B)$. These interactions indeed, are performed among orchestrators joined with the same roles and they must be considered as internal actions.

8.5.3 Example C

In this example we consider an orchestrated system where the market service queries a register service for obtaining the location of the supplier service and the bank one. Depending on the goods to purchase the register service will answer with the location of a supplier and, in the same way, depending on the card type it will answer with the location of a bank service. Differently from the previous example where, for the supplier and the bank, there is a central service that forwards the requests to the other ones, here

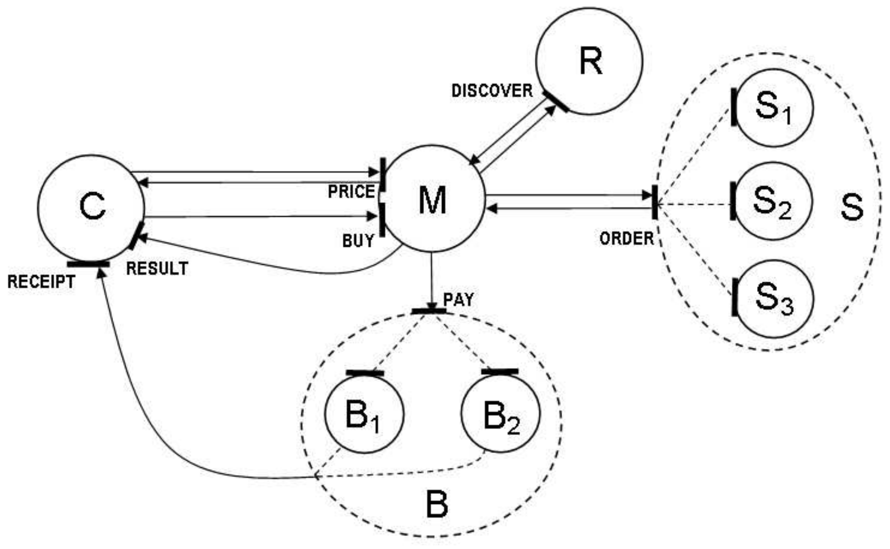

Figure 8.4: Orchestration system E_C

we model a typical SOC scenario where the service to invoke is retrieved by quierying a service register. This fact allows us to model all the supplier services and all the bank services by exploiting the same interface. All of them indeed, will exhibit the operation $\text{ORDER}_{t_2,t_2'}$, as far as the supplier services are concerned, and the operation PAY_{t_4}, as far as the bank services are concerned. In particular, here we have considered three supplier services and two bank services. The register service is modelled with a simple Request-Response operation (DISCOVER) where we abstract away from the retrieving procedure by representing it with the function $\texttt{queryDB}(query)$. Let us consider that supplier S_1 will be retrieved for apples, supplier S_2 for bananas, supplier S_3 for the strawberries, bank B_1 for visa cards and bank B_2 for mastercard cards. In Fig. 8.4 we present the communication links of this orchestrated system where dotted circles contain services that have all the same interface. In the following we present the code of the orchestrated system:

$t_6 = \langle inf \rangle$
$t_6' = \langle loc \rangle$

$$E_C \; ::= \; C \parallel M \parallel R \parallel S_1 \parallel S_2 \parallel S_3 \parallel B_1 \parallel B_2$$

$$\begin{array}{rcl}
C & ::= & [(loc_c := C; (\overline{PRICE}_{t_0,t_0'}@M(\langle good_C, num_C\rangle\,, price_C); \\
 & ; & (price_C \geq 100 \\
 & & ?buy_C := cancelled \\
 & & : buy_C := accepted) \\
 & ; & \overline{BUY}_{t_1}@M(\langle buy_C, card_C, ncard_C, loc_C\rangle)) \\
 & | & RESULT_{t_3}(result_C) \\
 & | & RECEIPT_{t_5}(receipt_C), \mathcal{S}_C)]_C
\end{array}$$

$$\begin{array}{rcl}
M & ::= & [(PRICE_{t_0,t_0'}(\langle good_M, num_M\rangle\,, price_M, \mathbf{0}) \mid \\
 & | & ((BUY_{t_1}(\langle buy_M, card_M, ncard_M, client\rangle) \\
 & ; & buy_M = accepted?Ord : \mathbf{0})), \mathcal{S}_M)]_M \\
Ord & ::= & \overline{DISCOVER}_{t_6,t_6'}@R(good_M, ordLoc) \\
 & ; & \overline{ORDER}_{t_2,t_2'}@ordLoc(\langle good_M, num_M\rangle\,, outcome_M) \\
 & ; & (\overline{RESULT}_{t_3}@client(outcome_M) \\
 & | & outcome_M = OK \\
 & & \;?\,\overline{DISCOVER}t_6, t_6'@R(card_M, payid); \\
 & & \;;\,\overline{PAY}_{t_4}@payid(\langle card_M, ncard_M\, price_M, client\rangle) \\
 & & \;:\;\mathbf{0}
\end{array}$$

$$\begin{array}{rcl}
S_1 & ::= & [(ORDER_{t_2,t_2'}(\langle good_{S1}, num_{S1}\rangle\,, outcome_{S1}, \mathbf{0}), \mathcal{S}_{S1})]_{S1} \\
S_2 & ::= & [(ORDER_{t_2,t_2'}(\langle good_{S2}, num_{S2}\rangle\,, outcome_{S2}, \mathbf{0}), \mathcal{S}_{S2})]_{S2} \\
S_3 & ::= & [(ORDER_{t_2,t_2'}(\langle good_{S3}, num_{S3}\rangle\,, outcome_{S3}, \mathbf{0}), \mathcal{S}_{S3})]_{S3}
\end{array}$$

$$\begin{array}{rcl}
B_1 & ::= & [(PAY_{t_4}(\langle card_{B1}, ncard_{B1}, price_{B1}, cid_{B1}\rangle) \\
 & ; & \overline{RECEIPT}_{t_5}@cid_{B1}(receipt_{B1}), \mathcal{S}_{B1})]_{B1} \\
B_2 & ::= & [(PAY_{t_4}(\langle card_{B2}, ncard_{B2}, price_{B2}, cid_{B2}\rangle) \\
 & ; & \overline{RECEIPT}_{t_5}@cid_{B2}(receipt_{B2}), \mathcal{S}_{B2})]_{B2}
\end{array}$$

$$R \quad ::= \quad [(tt \rightleftharpoons \mathrm{DISCOVER}_{t_6,t_6'}(query, l, l := \mathtt{queryDB}(query)), \mathcal{S}_R)]_{B1}$$

If we consider the joining function Ψ_B the orchestrated system E_C is not conformant to $Chor$ because the variable $outcome_{S_1}$ is not joint to any choreography information and it is undefined ($\perp$). Considering all the labelled transition system generated for each state of the choreography, there will be a transition system where the transition between the supplier (S_1) and the market will contain the value of the variable $outcome_{S_1}$ (undefined) that does not correspond to that of the choreograhy (contained within the information $outcome$). For this reason E_C is not conformant to $Chor$ up to Ψ_B. Now, let us consider the following joining function Ψ_C:

$\Psi^1_C(C) = \rho_C$
$\Psi^1_C(M) = \rho_M$
$\Psi^1_C(S_1) = \rho_S, \Psi^1_C(S_2) = \rho_S, \Psi^1_C(S_3) = \rho_S$
$\Psi^1_C(B_1) = \rho_B, \Psi^1_C(B_2) = \rho_B$
$\Psi^1_C(R) = \perp$
$\Psi^1_C(l) = \perp \text{ for } l \notin \{C, M, S_1, S_2, S_3, B_1, B_2\}$

$\Psi^2_C(C)(buy_C) = buy$
$\Psi^2_C(C)(good_C) = good$
$\Psi^2_C(C)(num_C) = num$
$\Psi^2_C(C)(card_C) = card$
$\Psi^2_C(C)(ncard_C) = ncard$
$\Psi^2_C(M)(price_M) = price$
$\Psi^2_C(S_1)(outcome_{S1}) = outcome$
$\Psi^2_C(S_2)(outcome_{S2}) = outcome$
$\Psi^2_C(S_3)(outcome_{S3}) = outcome$
$\Psi^2_C(B_2)(receipt_{B2}) = receipt$
$\Psi^2_C(B_3)(receipt_{B3}) = receipt$

The orchestrated system E_C is conformant to the choreography $Chor$ up to the joining function Ψ_C. It is worth noting that the register is not joined with any role but it colud be joined with role M without compromising the conformance relation.

8.5.4 Example D

In this example we consider a choreography where a non-deterministic choice is designed and we show a conformant orchestrated system where two non-deterministic choices are programmed. We will show that the former choice corresponds with that designed within the choreography whereas the latter one is not related to a non-determinism at the level of the choreography. The choreography models a simple question system. A player requests for a question to a questioner that replies with a question and then waits for an answer. The only admitted answers are yes or not. Once received the answer, a role that manages the question statistics receives the answer from the questioner. It is worth noting that the questioner can be stopped by a master role before receiving the question request from the player and, for the sake of simplicity, the choreography describes only the evolution of the system for a question. The choreography is formed by four roles: the Master (ρ_M), the Questioner (ρ_Q), the Player (ρ_P) and the Statistic role (ρ_S). In Fig. 8.5 we report the communication links of the system whereas in the following we define the templates and the operations supplied by the different roles:

$t_0 = \langle\rangle$
$t1 = \langle Inf\rangle$

$\omega_M = \{\overline{\mathrm{STOP}}_{t_0}\}$
$\omega_Q = \{\mathrm{STOP}_{t_0}, \mathrm{QUESTION}_{t_0,t_1}, \mathrm{ANSWER}_{t_1}, \overline{\mathrm{YES}}_{t_1}, \overline{\mathrm{NO}}_{t_1}\}$
$\omega_P = \{\overline{\mathrm{QUESTION}}_{t_0,t_1}, \overline{\mathrm{ANSWER}}_{t_1}\}$
$\omega_S = \{\mathrm{YES}_{t_1}, \mathrm{NO}_{t_1}\}$

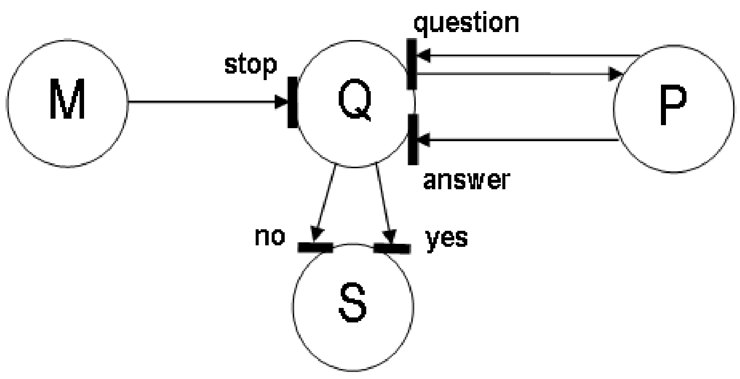

Figure 8.5: Choreography communication links of the example D

The information set contains the following information:

$$I_C = \{question, answer\}$$

where $question$ abstractly models a question and $answer$ is the information which contains the answer. Let Σ be the set of roles defined in the following way:

$$\Sigma = \{(\rho_M, \omega_M), (\rho_Q, \omega_Q), (\rho_P, \omega_P), (\rho_S, \omega_S)\}$$

The knowledge ($\mathcal{K} = (I, \Lambda)$) is defined as it follows:

$I(\rho_M) = \{\}$
$I(\rho_Q) = \{question\}$
$I(\rho_P) = \{answer\}$
$I(\rho_S) = \{\}$

$\Lambda(\rho_M) = \{\rho_Q\}$
$\Lambda(\rho_Q) = \{\rho_S\}$
$\Lambda(\rho_P) = \{\rho_Q\}$
$\Lambda(\rho_S) = \emptyset$

In the following we present the conversation Con of the choreography where, for the sake of clarity, we use the following notation for denoting One-Way and Request-Response interactions:

$\rho_A \rightharpoonup^{\vec{x}}_{o_{\vec{t}}} \rho_B \equiv OW(\rho_A, \rho_B, o_{\vec{t}}, \vec{x})$

$\rho_A \rightleftharpoons^{\vec{x},\vec{y}}_{o_{\vec{t},\vec{t}'}} \rho_B(C) \equiv RR(\rho_A, \rho_B, o_{\vec{t},\vec{t}'}, \vec{x}, \vec{y}, C)$

$$Con \ ::= \ OW(\rho_M, \rho_S, \text{STOP}_{t_0}, \langle\rangle) + (RR(\rho_P, \rho_S, \text{QUESTION}_{t_0,t_1}, \langle\rangle, question, \mathbf{0}); Ans)$$

$$\begin{aligned} Ans \ ::= \ & OW(\rho_P, \rho_S, \text{ANSWER}_{t_1}, answer); \\ & \texttt{if } answer == yes_{\rho_Q} \texttt{ then} \\ & \quad OW(\rho_Q, \rho_S, \text{YES}_{t_1}, question) \\ & \texttt{else} \\ & \quad OW(\rho_Q, \rho_S, \text{NO}_{t_1}, question) \end{aligned}$$

The initial constraints are defined as it follows:

$$\begin{aligned} X \ = \ & question = Aquestion \\ \wedge \ & answer \in \{yes, no\} \end{aligned}$$

The choreography is defined by the tuple $Chor2 = (\Sigma, Con, \mathcal{K}, X)$. It is worth noting that the choreography starts with a non-deterministic choice between the interactions $OW(\rho_M, \rho_S, \text{STOP}_{t_0}, \langle\rangle)$ and $RR(\rho_P, \rho_S, \text{QUESTION}_{t_0,t_1}, \langle\rangle, question, \mathbf{0})$. In the following, we present an orchestrated system for which it will be possible to define a joining function in order to be conformant to the choreography $Chor2$. The orchestrated system has an orchestrator for each role of the choreography.

$$E_D ::= M \parallel Q \parallel P \parallel S$$

$$M \ ::= \ [(\overline{\text{STOP}}_{t_0}@Q(\langle\rangle), \mathcal{S}_\perp)]_M$$

$$\begin{aligned} Q \ ::= \ & [(\text{STOP}_{t_0}(\langle\rangle) + (\text{QUESTION}_{t_0,t_1}(\langle\rangle, question_Q), \mathbf{0}; Answ), \mathcal{S}_\perp)]_Q \\ Answ \ ::= \ & \text{ANSWER}_{t_1}(answer_Q); \\ & answer_Q == yes? \\ & \quad \overline{\text{YES}}_{t_1}@S(question_Q) \\ & : \\ & \quad \overline{\text{NO}}_{t_1}@S(question_Q) \end{aligned}$$

$$P \quad ::= \quad [(\overline{\text{QUESTION}}_{t_0,t_1}@Q(\langle\rangle\,, \mathit{question}_P); \overline{\text{ANSWER}}_{t_1}@Q(\mathit{answer}_P), \mathcal{S}_\perp)]_P$$

$$S \quad ::= \quad [(\text{YES}_{t_1}(\mathit{question}_S) + \text{NO}_{t_1}(\mathit{question}_S), \mathcal{S}_\perp)]_S$$

If we consider the following joining function Ψ_D we obtain that the orchestrated system E_D is conformant to the choreography $Chor2$.

$\Psi^1_D(M) = \rho_M$
$\Psi^1_D(Q) = \rho_Q$
$\Psi^1_D(P) = \rho_P$
$\Psi^1_D(S) = \rho_S$

$\Psi^2_D(Q)(\mathit{question}_Q) = \mathit{question}$
$\Psi^2_D(P)(\mathit{answer}_P) = \mathit{answer}$

It is worth noting that two non-deterministic choices has been designed within the orchestrated system E_D. The former choice is programmed within the orchestrator Q whereas the latter one is programmed within the orchestrator S. From an orchestration point of view both the non-deterministic choices are reasonable to be designed because both the orchestrator Q and the orchestrator S are not able to predict which message they will receive first. But from a choreography view point only the former is a non-deterministic choice whereas the latter is deterministically selected at the beginning of the choreography by setting the variable answer to yes or to no. The non-deterministic choice within the orchestrator S indeed, it is strictly related to the value of the variable answer that, from a choreography view point, is deterministically set at the beginning of each execution of the choreography. This example shows as the non-deterministic choice at the level of the orchestration does not always correspond to a non-deterministic choice at the level of the choreography. Such a difference can be observed if we consider all the possible labelled transition systems for the choreography. There are two labelled transition systems: the former (Fig. 8.6) is obtained by considering the initial state $\mathit{question} = \mathit{Aquestion}$ and $\mathit{answer} = \mathit{yes}$ whereas the latter (Fig. 8.7) is obtained by considering the initial state $\mathit{question} = \mathit{Aquestion}$ and $\mathit{answer} = \mathit{no}$. It is possible

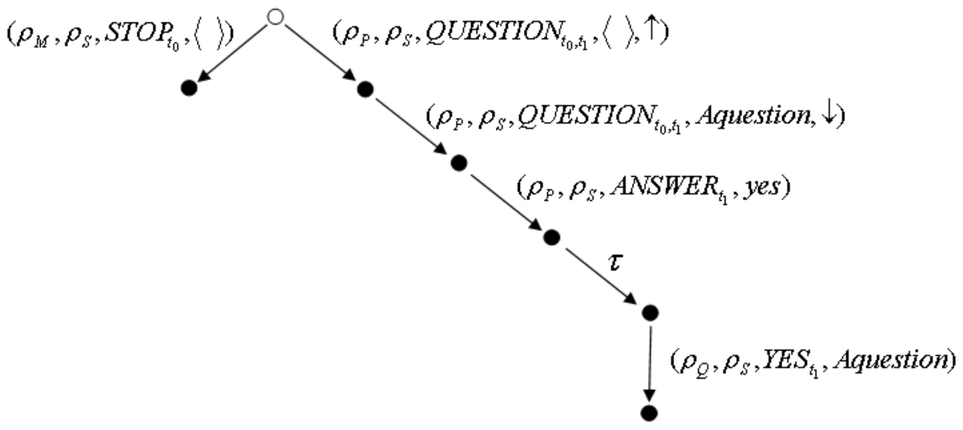
$(\rho_M, \rho_S, STOP_{t_0}, \langle\ \rangle)$
$(\rho_P, \rho_S, QUESTION_{t_0,t_1}, \langle\ \rangle, \uparrow)$
$(\rho_P, \rho_S, QUESTION_{t_0,t_1}, Aquestion, \downarrow)$
$(\rho_P, \rho_S, ANSWER_{t_1}, yes)$
τ
$(\rho_Q, \rho_S, YES_{t_1}, Aquestion)$

Figure 8.6: Choreography labelled transition system: $answer = yes$

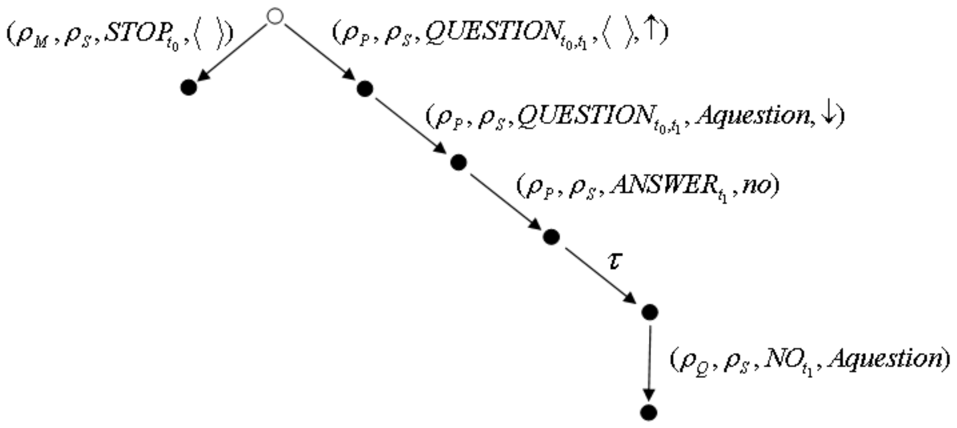
$(\rho_M, \rho_S, STOP_{t_0}, \langle\ \rangle)$
$(\rho_P, \rho_S, QUESTION_{t_0,t_1}, \langle\ \rangle, \uparrow)$
$(\rho_P, \rho_S, QUESTION_{t_0,t_1}, Aquestion, \downarrow)$
$(\rho_P, \rho_S, ANSWER_{t_1}, no)$
τ
$(\rho_Q, \rho_S, NO_{t_1}, Aquestion)$

Figure 8.7: Choreography labelled transition system: $answer = no$

to notice that the non-deterministic choice designed within the orchestrator Q is always modelled, within the choreography labelled transition systems, by two transitions raised from the initial state, whereas the non-deterministic choice, designed within the orchestrator S, is modelled by two different interactions of the two labelled transition system: $(\rho_Q, \rho_S, YES_{t_1}, Aquestion)$ and $(\rho_Q, \rho_S, NO_{t_1}, Aquestion)$.

8.6 Related works

Other works which deal with a conformance between a global approach, as the choreography language, and a local view point, as the orchestration, exist. In [CHY07] the authors propose a typed language, called global calculus, for describing a system in a top view manner then, they present a precise machinery for making the so-called Endpoint Projection. The Endpoint Projection extracts, from a global view description, the behaviour of each participant in a way that it satisfies the global specification. In [BBM$^+$05a] Baldoni et al. present a conformance notion between a choreography description and an orchestration ones in terms of a similation between state finite automata but they limited their analysis only to a choreography with two participants furthermore, in [BBM$^+$05b], the same authors extend their proposal to more participant choreography and they present a notion of conformance based on two different simulations where they distinguish between incoming messages and outcoming messages. In [KP06] the authors extend our previous work [BGG$^+$05b] without providing two formal languages for choregraphy and orchestration but formalizing the semantics of WS-CDL and WS-BPEL in terms of labelled transition systems. In [DD04] Dijkman and Dumas exploit Petri nets for describing choreography, orchestration and service interface behaviours focusing on the relationship between a single orchestrator w.r.t. a given choreography. Finally, in [HM05] Heckel et al. indirectly deal with a conformance notion, intended as a relationship between a global description wr.t. a local one, by focusing on automated testing of behavioural contracts provided by a service.

Chapter 9

System design with a bipolar framework

In chapter 3, we have proposed a unique language, SOCK, for addressing both *service design* and *service composition* issues. Such a language could be enough for dealing with a services system design because it allows for the internal programming of a service and, concurrently, it allows for its composition within a system. Although that, if we consider a system where there are a lot of services, a designer could encounter difficulties in managing both service design and composition with a unique approach. Let us consider, for example, the case of the system presented in Section 3.4 which is formed of seven services; a designer could start its design by programming the market service and, step by step, it can modify the other involved services trying to approach the system design by considering all the details of each service. When there is a great number of services, such an activity could be very difficult with a high probability to make errors. In this context, tools for managing a great number of services are needed. But, which kind of facilities do they have to provide? How can they actually supply a more intuitive way for approaching a complex system design?

In this chapter, we want to address the system design issue by considering the so-called *bipolar approach* where the choreography and the orchestration languages we have discussed in Chapters 6 and 7, are exploited for dealing with different aspects of a services system. The choreography language supplies a view of the system from a global perspective whereas the orchestration deals with a local view point. The bipolar approach is based upon a mathematical *liaison* between the semantics representation of the two languages called *conformance* which allows us to verify if a system described with the orchestration language is *conformant* to that described with a choreography. The idea the bipolar approach is based upon is that *a difficult thing in orchestration is an easy thing in choreography and vice versa*. In order to give the intuition we can trace a comparison with signal analysis. A signal can be processed in the time domain or in the frequency one and the Fourier transform allows for the change from one domain to the other one. It is well known that some things are easy in the frequency domain (e.g. filter design) and other things are easy in the time domain (e.g. signal sampling). In the same way, orchestration and choreography languages supply different domains for representing composed systems whereas the conformance relation plays the role of the Fourier transform. Such a kind of framework could be exploited as it follows: a first coarse system can be designed as a choreography from which it is possible to extract a conformant orchestrated system skeleton that, subsequently, can be enhanced by adding other services or by enriching the behaviour of the existing ones. Afterward, it is possible to rebuild a conformant choreography from the previous system and then adjust or enhance it for introducing more details; then, from the new choreography it will be possible to come back to the orchestrated system and so on. It is reasonable to suppose that, as a final result, the design steps must achieve a connected error-free choreography. We can describe the bipolar approach by means of Fig 9.1 where the relation between the orchestration domain and the choreography one is given by the conformance notion and two different algorithms, the Extracting Choreography and the Extracting an Orchestration, allow for the generation of a choreography starting from an orchestrated system and the generation of an orchestrated system starting from a choreography respectively. At the present, we are focusing on the orchestration and choreography domains development and on the conformance relation between them. As far as the two algorithms are concerned, we have started to analyze them even if, so far, we cannot present results related to them.

In the remainder of this chapter, we aim at providing two examples where two services system are designed by exploiting a bipolar approach. It is worth noting that there is not a predetermined procedure to follow for achieving the design of a system by exploiting the bipolar approach, but the designer is free to choose the view he prefers as he prefers

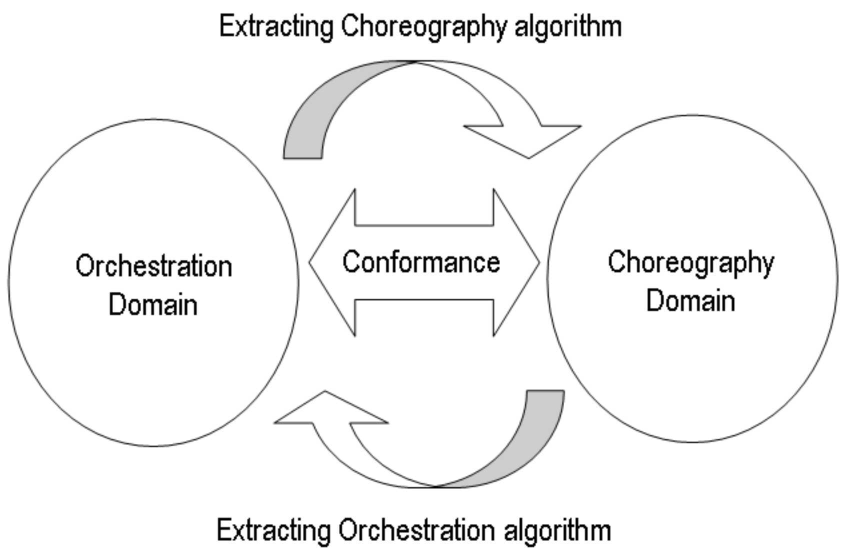
Extracting Choreography algorithm
Orchestration Domain
Conformance
Choreography Domain
Extracting Orchestration algorithm

Figure 9.1: The bipolar framework

and when he prefers. The conformance notion supplies the mathematical machinery for interpreting a view into the other one and the designer can exploit it as he wants. In this context, the following examples do not show the best procedure for achieving a system design but only some possible approaches that can be followed. As we have done within the previous chapter, here, for the sake of clarity, we exploit the following notation for representing the choreography One-Way interactions and the Request-Response ones:

$$\rho_A \rightharpoonup^{\vec{x}}_{o_{\vec{t}}} \rho_B \equiv OW(\rho_A, \rho_B, o_{\vec{t}}, \vec{x})$$
$$\rho_A \rightleftharpoons^{\vec{x},\vec{y}}_{o_{\vec{t},\vec{t}'}} \rho_B(C) \equiv RR(\rho_A, \rho_B, o_{\vec{t},\vec{t}'}, \vec{x}, \vec{y}, C)$$

9.1 Hospital reservation example

In this example a hospital reservation exams service is designed where choreography and orchestration views are alternatively used for adding new orchestrators or modifying the system at the design time. The Hospital Reservation Exams Service, HRES for short, receives a request for an exam reservation from a client service and it makes the reservation. The HRES will exploit two other services: the *reservation service* which performs the reservation and the archive service which stores all the personal data of the clients.

9.1.1 Step 1 (a first choreography).

The design could start by programming a simple choreography with two roles: the *client* (ρ_C) and the *HRES* (ρ_H). Thus, the set of the role names is:

$$RName = \{\rho_C, \rho_H\}$$

The communication links of the choreography could be formed by a Request-Response operation exhibited by ρ_H for receiving a request from the client and then sending the response back to it. The operation set follows:

$$Op := \{(res, rr, \vec{t}_r, \vec{t}'_r), (res, sr, \vec{t}_r, \vec{t}'_r)\}$$

where $\vec{t}_r$ and $\vec{t}'_r$ are operation templates which we define after we will design the exchanged information. Once defined the operations, the set of roles follows:

$$Role := \{(\rho_C, \overline{res}_{\vec{t}_r, \vec{t}'_r}), (\rho_H, res_{\vec{t}_r, \vec{t}'_r})\}$$

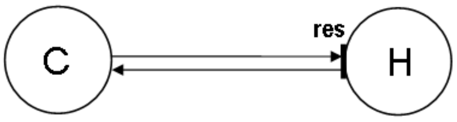

Figure 9.2: Hospital reservation example communication links Step 1

In Fig. 9.2 the communication links are represented. Now, we consider the exchanged information. To this end, let us consider that the client name ($name$), the client surname ($surname$) and the exam type ($type$) are sufficient both for identifying the client and reserving the exam. Furthermore, let the reservation data be identified by the unique information $rdata$. The information set follows:

$$I_C = \{name, surname, type, rdata\}$$

Since we suppose that the client sends to the HRES the information $name$, $surname$ and $type$ and receives as a response the $rdata$ information, we define the operation templates as follows:

$$\vec{t}_r = \langle inf, inf, inf \rangle \quad \vec{t}'_r = \langle inf \rangle$$

Moreover, it descends that the initial distribution of knowledge is:

$\mathcal{K} = \{I, \Lambda\}$
$I(\rho_C) = \{name, surname, type\} \quad I(\rho_H) = \{rdata\}$
$\Lambda(\rho_C) = \Lambda(\rho_H) = \emptyset$

As initial constraints we exploit the following ones:

$X ::= name = Aname \wedge surname = Asurname \wedge type = Atype \wedge rdata = Somedata$

The conversation is formed by a Request-Response interaction between the client and the HRES:

$$Con := RR(\rho_C, \rho_H, res_{\vec{t}_r, \vec{t'}_r}, \langle name, surname, type \rangle, rdata, \mathbf{0})$$

From this choreography we can extract a conformant orchestrated system[1]. A conformant orchestrated system could be the following one where we join the role ρ_C to the orchestrator C and the role ρ_H to the orchestrator H:

$C := [\overline{res}_{\vec{t}_r, \vec{t}'_r}@Hloc(\langle name_C, surname_C, type_C \rangle, rdata_C), \mathcal{S}_C]@Cloc$
$H := [res_{\vec{t}_r, \vec{t}'_r}(\langle name_H, surname_H, type_H \rangle, rdata_H, \mathbf{0}), \mathcal{S}_H]@Hloc$

The joining function we exploit in order to test the conformance follows:

[1] At the present we are developing algorithms for doing this step in an automatic way.

$\Psi^1(C) = \rho_C \quad \Psi^1(H) = \rho_H$

$\Psi^2(C)(name_C) = name \quad \Psi^2(C)(surname_C) = surname$

$\Psi^2(C)(type_C) = type \qquad \Psi^2(H)(rdata_H) = rdata$

9.1.2 Step 2 (introducing a reservation service).

Now, we decide to introduce a reservation service (R) which actually performs the reservation. The HRES will invoke the service R forwarding the data received by the client and waiting for the reservation data where we abstract away from the internal computation steps of R. Finally, the reservation data will be forwarded to the client.

$C := [\overline{res}_{\vec{t}_r,\vec{t}'_r}@Hloc(\langle name_C, surname_C, type_C\rangle, rdata_C), \mathcal{S}_\perp]@Cloc$

$H := [res_{\vec{t}_r,\vec{t}'_r}(\langle name_H, surname_H, type_H\rangle, rdata_H, ReqRes), \mathcal{S}_\perp]@Hloc$
$ReqRes := \overline{reqres}_{\vec{t}_r,\vec{t}'_r}@Rloc(\langle name_H, surname_H, type_H\rangle, rdata_H)$

$R := [reqres_{\vec{t}_r,\vec{t}'_r}(\langle name_R, surname_R, type_R\rangle, rdata_R, \mathbf{0}), \mathcal{S}_\perp]$

Since at this step we consider the reservation service as a refinement of the HRES one, we consider the service R joined with the same role ρ_H and we modify the joining function as it follows:
$\Psi^1(C) = \rho_C \quad \Psi^1(H) = \rho_H \quad \Psi^1(R) = \rho_H$
$\Psi^2(C)(name_C) = name \quad \Psi^2(C)(surname_C) = surname$
$\Psi^2(C)(type_C) = type \qquad \Psi^2(R)(rdata_R) = rdata$
The orchestrated system is conformant to the previous choreography because the interactions between the service engine H and the service engine R are internal actions and they do not alter the conformance. At this point, since we consider that the registration service could be relevant from a global view point, we modify the choreography in order to take into account the new service enigne R by introducing a new role ρ_R. Thus, the set of the role names and the operation set are enriched as follows:

$RName = \{\rho_C, \rho_H, \rho_R\}$

$$\begin{aligned} Op := \{&(res, rr, \vec{t}_r, \vec{t}'_r), (res, sr, \vec{t}_r, \vec{t}'_r),\\ &(reqres, rr, \vec{t}_r, \vec{t}'_r), (reqres, sr, \vec{t}_r, \vec{t}'_r)\} \end{aligned}$$

The set of roles follows:

$$Role := \{(\rho_C, \overline{res}_{\vec{t}_r, \vec{t}'_r}), (\rho_H, \{res_{\vec{t}_r, \vec{t}'_r}, \overline{reqres}_{\vec{t}_r, \vec{t}'_r}\}), (\rho_R, reqres_{\vec{t}_r, \vec{t}'_r})\}$$

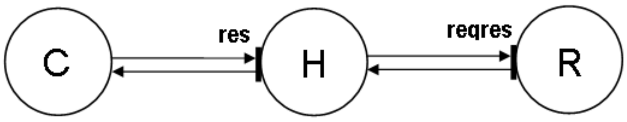

Figure 9.3: Hospital reservation example communication links Step 2

It is worth noting that the initial distribution of knowledge is modified because in this new choreography the $rdata$ is initially known by the role ρ_R and not by the role ρ_H:

$\mathcal{K} = \{I, \Lambda\}$
$I(\rho_C) = \{name, surname, type\} \quad I(\rho_H) = \emptyset \quad I(\rho_R) = \{rdata\}$
$\Lambda(\rho_C) = \Lambda(\rho_H) = \Lambda(\rho_R) = \emptyset$

The conversation is:

$Con := RR(\rho_C, \rho_H, res_{\vec{t}_r, \vec{t'}_r}, \langle name, surname, type\rangle, rdata, RqRs)$
$Rqrs := RR(\rho_H, \rho_R, reqres_{\vec{t}_r, \vec{t'}_r}, \langle name, surname, type\rangle, rdata, \mathbf{0})$

We modify the joining function as it follows:

$\Psi^1(C) = \rho_C \quad \Psi^1(H) = \rho_H \quad \Psi^1(R) = \rho_R$
$\Psi^2(C)(name_C) = name \quad \Psi^2(C)(surname_C) = surname$
$\Psi^2(C)(type_C) = type \qquad \Psi^2(R)(rdata_R) = rdata$

9.1.3 Step 3 (Interaction modification).

Now, by reasoning about the choreography, we want to modify the previous choreography in order to allow the reservation service to send the reservation data directly to the client without communicating it to the role ρ_H. In this case all the operations will be changed into One-Way operations. The set of the role names is not modified whereas the operation set is defined as follows:

$$\begin{aligned}Op := \{&(res, ow, \vec{t}_r), (res, n, \vec{t}_r),\\ &(reqres, ow, \vec{t}_r), (reqres, n, \vec{t}_r)\\ &(rd, ow, \vec{t}'_r), (rd, n, \vec{t}'_r)\}\end{aligned}$$

where the templates now are defined as:

$$\vec{t}_r = \langle inf, inf, inf\rangle \quad \vec{t}'_r = \langle inf\rangle$$

The communication links are represented in Fig. 9.4 and the set of roles follows:

$$Role := \{(\rho_C, \{\overline{res}_{\vec{t}_r}, rd_{\vec{t}'_r}\}), (\rho_H, \{res_{\vec{t}_r}, \overline{reqres}_{\vec{t}_r}\}), (\rho_R, \{reqres_{\vec{t}_r}, \overline{rd}_{\vec{t}'_r}\})\}$$

It is worth noting that the client has to exhibit a One-Way operation (rd) in order to re-

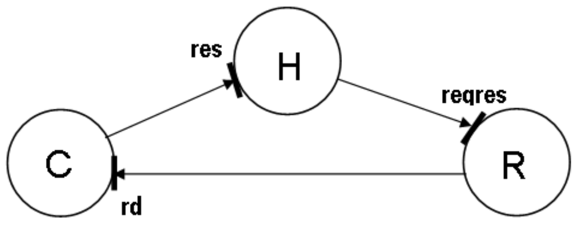

Figure 9.4: Hospital reservation example communication links Step 3

ceive the $rdata$ information from ρ_R. The initial distribution of knowledge is unchanged whereas the conversation is modified in order to take into account three One-Way interactions:

$\mathsf{Con} := \mathsf{Res}; \mathsf{RqRs}; \mathsf{Rd}$

$\mathsf{Res} := \mathsf{OW}(\rho_C, \rho_H, \mathsf{res}_{\vec{t}_r}, \langle \mathsf{name}, \mathsf{surname}, \mathsf{type} \rangle)$
$\mathsf{Rqrs} := \mathsf{OW}(\rho_H, \rho_R, \mathsf{reqres}_{\vec{t}_r}, \langle \mathsf{name}, \mathsf{surname}, \mathsf{type} \rangle)$
$\mathsf{Rd} := \mathsf{OW}(\rho_R, \rho_C, \mathsf{rd}_{\vec{t}'_r}, \mathsf{rdata})$

Such a choreography seems to satisfy our purposes but it is not error-free. Indeed, when the role ρ_R try to send the rdata to role ρ_C, it does not know its location because it never receives it from role ρ_H and its initial knowledge does not contain such an information. In this case, we have to introduce a location mobility between ρ_H and ρ_R. To this end, we modify the operation template $\vec{t}_r$ and the conversations as it follows:

$\vec{t}_r = \langle \mathsf{inf}, \mathsf{inf}, \mathsf{inf}, \mathsf{loc} \rangle$

$\mathsf{Con} := \mathsf{Res}; \mathsf{RqRs}; \mathsf{Rd}$

$\mathsf{Res} := \mathsf{OW}(\rho_C, \rho_H, \mathsf{res}_{\vec{t}_r}, \langle \mathsf{name}, \mathsf{surname}, \mathsf{type}, \rho_C \rangle)$
$\mathsf{Rqrs} := \mathsf{OW}(\rho_H, \rho_R, \mathsf{reqres}_{\vec{t}_r}, \langle \mathsf{name}, \mathsf{surname}, \mathsf{type}, \rho_C \rangle)$
$\mathsf{Rd} := \mathsf{OW}(\rho_R, \rho_C, \mathsf{rd}_{\vec{t}'_r}, \mathsf{rdata})$

Now, we consider a new orchestration system where each orchestrator is joined to a role in choreography:

$\mathsf{C} := [\mathsf{l} := \mathsf{Cloc}; \overline{\mathsf{res}}_{\vec{t}_r}@\mathsf{Hloc}(\langle \mathsf{name}_C, \mathsf{surname}_C, \mathsf{type}_C, \mathsf{l} \rangle)$
$\quad ; \mathsf{rd}_{\vec{t}'_r}(\mathsf{rdata}_C), \mathcal{S}_\perp]@\mathsf{Cloc}$

$\mathsf{H} := [\mathsf{res}_{\vec{t}_r}(\langle \mathsf{name}_H, \mathsf{surname}_H, \mathsf{type}_H, \mathsf{loc}_H \rangle)$
$\quad ; \overline{\mathsf{reqres}}_{\vec{t}_r}@\mathsf{Rloc}(\langle \mathsf{name}_H, \mathsf{surname}_H, \mathsf{type}_H, \mathsf{loc}_H \rangle), \mathcal{S}_\perp]@\mathsf{Hloc}$

$$R := [reqres_{\vec{t}_r}(\langle name_R, surname_R, type_R, loc_R\rangle); \overline{rd}_{\vec{t}'_r}@loc_R(rdata_R), \mathcal{S}_\perp]$$

The joining function is not altered and we continue to exploit that of step 2.

9.1.4 Step 4 (introducing an archive service)

Now, we decide to introduce an archive service (A) which stores the personal data of the clients. The reservation service will need the ID of the client in order to perform the reservation and it will request for it forwarding the name and the surname to the archive service. Since we consider the service A as a refinement of the service R, the orchestrated system will be modified as it follows where we consider the service A joined with the role ρ_R:

$$C := [l := Cloc; \overline{res}_{\vec{t}_r}@Hloc(\langle name_C, surname_C, type_C, l\rangle); rd_{\vec{t}'_r}(rdata_C), \mathcal{S}_\perp]@Cloc$$

$$\begin{aligned} H := [&res_{\vec{t}_r}(\langle name_H, surname_H, type_H, loc_H\rangle) \\ &; \overline{reqres}_{\vec{t}_r}@Rloc(\langle name_H, surname_H, type_H, loc_H\rangle), \mathcal{S}_\perp]@Hloc \end{aligned}$$

$$\begin{aligned} R := [&reqres_{\vec{t}_r}(\langle name_R, surname_R, type_R, loc_R\rangle) \\ &; \overline{id}_{\vec{t}_a, \vec{t}'_a}@Aloc(\langle name_R, surname_R\rangle, id_R) \\ &; \overline{rd}_{\vec{t}'_r}@loc_R(rdata_R), \mathcal{S}_\perp] \end{aligned}$$

$$A := [id_{\vec{t}_a, \vec{t}'_a}(\langle name_A, surname_A\rangle, id_A, \mathbf{0}), \mathcal{S}_\perp]@Aloc$$

where

$$\vec{t}_a = \langle inf, inf\rangle \quad \vec{t}'_a = \vec{t}'_r = \langle inf\rangle$$

Also in this case, since we consider that the service A is relevant from a global view point, we modify the choreography in order to take into account the new service A. The set of the role names and the operation set are enriched as follows:

$$RName = \{\rho_C, \rho_H, \rho_R, \rho_A\}$$

$$\begin{aligned} Op := \{&(res, ow, \vec{t}_r), (res, n, \vec{t}_r),\\ &(reqres, ow, \vec{t}_r), (reqres, n, \vec{t}_r)\\ &(id, rr, \vec{t}_a, \vec{t}'_a), (id, sr, \vec{t}_a, \vec{t}'_a)\\ &(rd, ow, \vec{t}'_r), (rd, n, \vec{t}'_r)\} \end{aligned}$$

The set of roles follows:

$$\begin{aligned} Role := \{&(\rho_C, \overline{res}_{\vec{t}_r,\vec{t}'_r}), (\rho_H, \{res_{\vec{t}_r,\vec{t}'_r}, \overline{reqres}_{\vec{t}_r,\vec{t}'_r}\}),\\ &(\rho_R, \{reqres_{\vec{t}_r,\vec{t}'_r}, \overline{id}_{\vec{t}_a,\vec{t}'_a}, \overline{rd}_{\vec{t}'_r}\}), (\rho_A, id_{\vec{t}_a,\vec{t}'_a})\} \end{aligned}$$

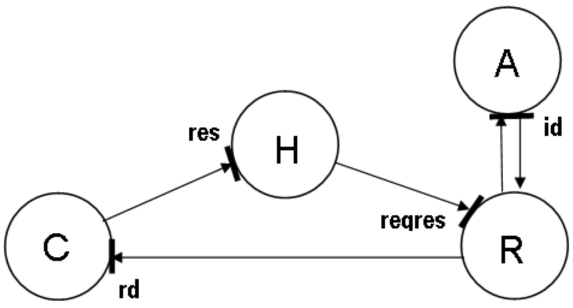

Figure 9.5: Hospital reservation example communication links Step 4

We need to introduce a new information which models the ID of the client known by the archive service. Thus, the information set and the initial distribution of knowledge are:

$$I_C = \{name, surname, type, rdata, id\}$$

$\mathcal{K} = \{I, \Lambda\}$

$I(\rho_C) = \{name, surname, type\} \quad I(\rho_H) = \emptyset \quad I(\rho_R) = \{rdata\} \quad I(A) = \{id\}$

$\Lambda(\rho_C) = \{\rho_C\} \quad \Lambda(\rho_H) = \Lambda(A) = \emptyset \quad \Lambda(\rho_R) = \{\rho_A\}$

We suppose to enrich the initial constraints with the condition $id = AnID$. The conversation must take into account the Request-Response interaction between the reservation service and the archive one:

$Con := Res; RqRs; Arch; Rd$

$Res := OW(\rho_C, \rho_H, res_{\vec{t}_r}, \langle name, surname, type, \rho_C\rangle)$
$Rqrs := OW(\rho_H, \rho_R, reqres_{\vec{t}_r}, \langle name, surname, type, \rho_C\rangle)$
$Arch := RR(\rho_R, \rho_A, id_{\vec{t}_a, \vec{t}'_a}, \langle name, surname\rangle, id, \mathbf{0})$
$Rd := OW(\rho_R, \rho_C, rd_{\vec{t}'_r}, rdata)$

The joining function is modified as it follows:
$\Psi^1(C) = \rho_C \quad \Psi^1(H) = \rho_H \quad \Psi^1(R) = \rho_R \quad \Psi^1(A) = \rho_A$
$\Psi^2(C)(name_C) = name \quad \Psi^2(C)(surname_C) = surname$
$\Psi^2(C)(type_C) = type \qquad \Psi^2(R)(rdata_R) = rdata \qquad \Psi^2(A)(id_A) = id$

9.1.5 Step 5 (interaction changing).

Now we want to modify the interactions in order to increase the efficiency of the previous system. The requests for the ID indeed, can be performed directly by the HRES as it is shown in the communication links picture reported in Fig. 9.6. In this case the archive service receives the notification from the HRES service and then forwards the client id directly to the registration service. The operation set and the role set are modified as it follows where the operation id is that exploited by the HRES service for notifying the archive service whereas the operation id_2 is exhibited by the reservation service for receiving the client id from the archive one:

$Op := \{(res, ow, \vec{t}_r), (res, n, \vec{t}_r),$
$\qquad (reqres, ow, \vec{t}_{res}), (reqres, n, \vec{t}_{res})$

$$(id, ow, \vec{t}_a), (id, n, \vec{t}_a)$$
$$(id2, ow, \vec{t}'_a), (id2, n, \vec{t}'_a)$$
$$(rd, ow, \vec{t}'_r), (rd, n, \vec{t}'_r)\}$$

$$Role := \{(\rho_C, \overline{res}_{\vec{t}_r, \vec{t}'_r}), (\rho_H, \{res_{\vec{t}_r, \vec{t}'_r}, \overline{reqres}_{\vec{t}_r, \vec{t}'_r}, \overline{id}_{\vec{t}_a}\}), (\rho_R, \{reqres_{\vec{t}_r, \vec{t}'_r}, id2_{\vec{t}'_a}, \overline{rd}_{\vec{t}'_r}\}), (\rho_A, \{id_{\vec{t}_a}, \overline{id2}_{\vec{t}'_a}\})\}$$

where:

$$\vec{t}_{res} = \langle inf, loc \rangle \quad \vec{t}_a = \langle inf, inf \rangle \quad \vec{t}'_a = \langle inf \rangle$$

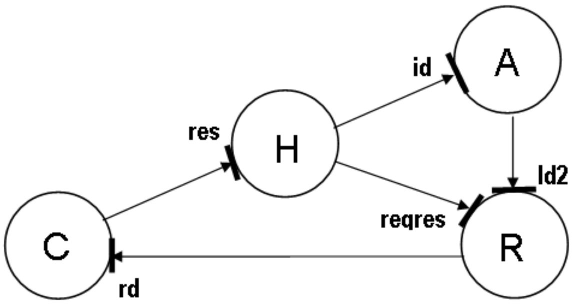

Figure 9.6: Hospital reservation example communication links Step 5

The initial distribution of the knowledge does not change w.r.t. step 4 whereas the conversation is changed as it follows:

$$Con := Res; (RqRs \mid (Arch1; Arch2)); Rd$$

$$Res := OW(\rho_C, \rho_H, res_{\vec{t}_r}, \langle name, surname, type, \rho_C \rangle)$$
$$Rqrs := OW(\rho_H, \rho_R, reqres_{\vec{t}_{res}}, \langle type, \rho_C \rangle)$$
$$Arch1 := OW(\rho_H, \rho_A, id_{\vec{t}_a}, \langle name, surname \rangle)$$
$$Arch2 := OW(\rho_A, \rho_R, id2_{\vec{t}'_a}, id)$$

$Rd := OW(\rho_R, \rho_C, rd_{\vec{t}'_r}, rdata)$

It is worth noting that conversations $Arch1$ and $Arch2$ are performed in parallel with the conversation $Rqrs$ and only when all the parallel composition is finished the conversation Rd can be performed. The orchestrated system is modified as it follows:

$$\begin{aligned} C := [&l := Cloc; \overline{res}_{\vec{t}_r}@Hloc(\langle name_C, surname_C, type_C, l\rangle) \\ &; rd_{\vec{t}'_r}(rdata_C), \mathcal{S}_\perp]@Cloc \end{aligned}$$

$$\begin{aligned} H := [&res_{\vec{t}_r}(\langle name_H, surname_H, type_H, loc_H\rangle) \\ &; (\overline{reqres}_{\vec{t}_r}@Rloc(\langle type_H, loc_H\rangle) \\ &\mid \overline{id}_{\vec{t}_a}@Aloc(\langle name_H, surname_H\rangle)), \mathcal{S}_\perp]@Hloc \end{aligned}$$

$$\begin{aligned} R := [&reqres_{\vec{t}_r}(\langle name_R, surname_R, type_R, loc_R\rangle) \\ &; \overline{id}_{\vec{t}_a, \vec{t}'_a}@Aloc(\langle name_R, surname_R\rangle, id_R) \\ &; \overline{rd}_{\vec{t}'_r}@loc_R(rdata_R), \mathcal{S}_\perp] \end{aligned}$$

$$\begin{aligned} A := [&id_{\vec{t}_a}(\langle name_A, surname_A\rangle) \\ &; \overline{id2}_{\vec{t}'_a}@Rloc(id_A), \mathcal{S}_\perp]@Aloc \end{aligned}$$

We exploit the same joining function of step 5 and we test that such an orchestrated system is conformant to the given choreography. The choreography is error-free and connected and it satisfies our purposes. At this point, we can consider finished our design phase and we can add to the orchestration abstract processes some constructs for specifying some internal computational steps as it follows:

$$\begin{aligned} C := [&l := Cloc; \overline{res}_{\vec{t}_r}@Hloc(\langle name_C, surname_C, type_C, l\rangle) \\ &; rd_{\vec{t}'_r}(rdata_C), \mathcal{S}_\perp]@Cloc \end{aligned}$$

$$\begin{aligned} H := [&res_{\vec{t}_r}(\langle name_H, surname_H, type_H, loc_H\rangle) \\ &; (\overline{reqres}_{\vec{t}_r}@Rloc(\langle type_H, loc_H\rangle) \end{aligned}$$

$\quad\quad | \ \overline{id}_{\vec{t}_a}@Aloc(\langle name_H, surname_H\rangle)), \mathcal{S}_{\perp}]@Hloc$

$R := [reqres_{\vec{t}_r}(\langle name_R, surname_R, type_R, loc_R\rangle)$
$\quad\quad ; \overline{id}_{\vec{t}_a, \vec{t}'_a}@Aloc(\langle name_R, surname_R\rangle, id_R)$
$\quad\quad ; rdata_R := \mathtt{reservation}(\langle type_R, id_R\rangle)$
$\quad\quad ; \overline{rd}_{\vec{t}'_r}@loc_R(rdata_R), \mathcal{S}_{\perp}]$

$A := [id_{\vec{t}_a}(\langle name_A, surname_A\rangle); id_A := \mathtt{query}(\langle name_A, surname_A\rangle)$
$\quad\quad ; \overline{id2}_{\vec{t}'_a}@Rloc(id_A), \mathcal{S}_{\perp}]@Aloc$
$query ::= id_A := \mathtt{query}(\langle name_A, surname_A\rangle)$

where the functions $\mathtt{reservation}(\langle type_R, id_R\rangle)$ and $\mathtt{query}(\langle name_A, surname_A\rangle)$ model the reservation procedure and the database query respectively.

9.2 Market example

In this example we model a business scenario similar to that proposed in the example of Section 6.6. In particular, a customer sends a request to a market service in order to receive the price of a product. The market will forwards the request to the supplier of the product and then replies to the customer with the price. Depending on the price the market will decide to buy or not the product. If the customer decides to accept, it will send an order request to the market. The market will initiate the order by sending a request to the supplier and it will also initiate a financial transaction by invoking the bank service. Then, the bank service will request for the bank account data by invoking both the customer and the supplier. At the end, the bank service will send commit messages to the market, to the customer and to the supplier. The supplier will finish by sending a commit message to the customer. In order to show that the system design can be achieved by exploiting the bipolar approach differently from the previous example, in this case we will start by designing a not connected choreography by focusing before on the interactions that involve the customer and then to those that involve the other roles.

9.2.1 Step 1 (the customer)

As a first step we design a not connected choreography which focuses only on the interactions which deal with the customer. In particular, we consider four different roles: the customer (ρ_C), the market (ρ_M), the supplier (ρ_S) and the bank (ρ_B). The set of role names follows:

$$RName ::= \{\rho_C, \rho_M, \rho_S, \rho_B\}$$

The customer will interact with the market for requesting an order, it will interact with the bank role for initiating the financial transaction and it will wait for a supplier commit message. The set of the operations follows and the communication links are reported in Fig. 9.7:

$$\begin{aligned} Op := \{&(price, rr, \vec{t_1}, \vec{t_1}'), (price, sr, \vec{t_1}, \vec{t_1}')\\ &(buy, ow, \vec{t_2}), (buy, n, \vec{t_2})\\ &(bankData, rr, \vec{t_3}, \vec{t_3}'), (bankData, sr, \vec{t_3}, \vec{t_3}')\\ &(bankCommit, ow, \vec{t_4}), (bankCommit, n, \vec{t_4})\\ &(suppCommit, ow, \vec{t_5}), (suppCommit, n, \vec{t_5})\} \end{aligned}$$

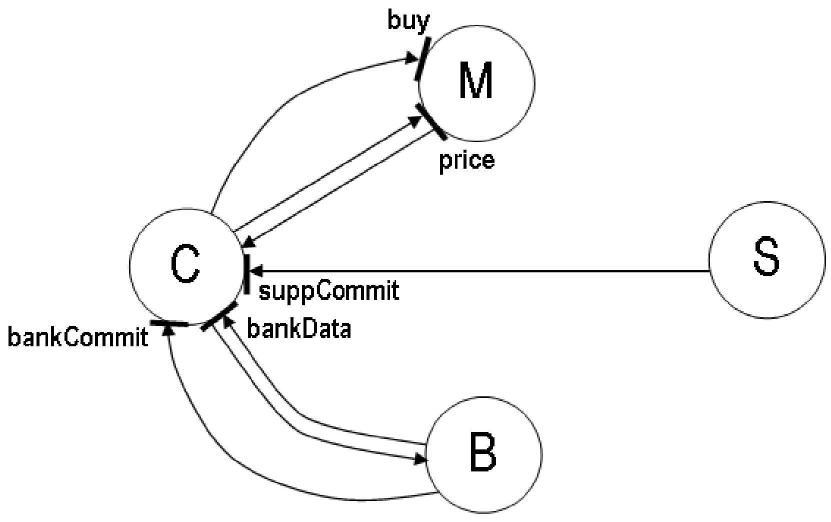

Figure 9.7: Market Example communication links Step 1

where the operation $\mathtt{price}$ allows the customer to request for a product order by sending a good type and a quantity. The operation $\mathtt{buy}$ is exploited for confirming or cancelling the order and, if the order is confirmed, the customer will send its location in order to allow the supplier and the bank roles to interact with it. The operation $\mathtt{bankData}$ allows the bank role to retrieve all the necessary data for performing the financial transaction, such as the credit card type and the credit card number. The operation $\mathtt{bankCommit}$ allows the customer to receive a commit message from the bank role whereas the operation $\mathtt{suppCommit}$ allows it to receive a commit message from the supplier role. The set of role is defined as it follows:

$$\begin{aligned}Role := \{&(\rho_C, \{\overline{price}_{\vec{t_1},\vec{t_1}'}, \overline{buy}_{\vec{t_2}}, bankData_{\vec{t_3},\vec{t_3}'}, bankCommit_{\vec{t_4}}, suppCommit_{\vec{t_5}}\}),\\ &(\rho_M, \{price_{\vec{t_1},\vec{t_1}'}, buy_{\vec{t_2}}\})\\ &(\rho_B, \{\overline{bankData}_{\vec{t_3},\vec{t_3}'}, \overline{bankCommit}_{\vec{t_4}}\})\\ &(\rho_S, \{\overline{suppCommit}_{\vec{t_5}}\})\}\end{aligned}$$

In the following we list the information exploited within the choreography:

$$I_C := \{good, quantity, price, confirm, cctype, ccnumber, bcommit, scommit\}$$

where $\mathtt{good}$ and $\mathtt{quantity}$ represent the good type and the quantity to purchase whereas $\mathtt{confirm}$ represents the information which contains a customer confirmation or not. $\mathtt{price}$ represents the order price. $\mathtt{cctype}$ and $\mathtt{ccnumber}$ represent the customer card credit type and number respectively whereas $\mathtt{bcommit}$ and $\mathtt{scommit}$ represent the commit messages from the bank and the supplier respectively. The operation templates follows:

$\vec{t_1} = \vec{t_3}' = \langle inf, inf\rangle \quad \vec{t_1}' = \vec{t_4} = \vec{t_5} = \langle inf\rangle \quad \vec{t_2} = \langle inf, loc\rangle \quad \vec{t_3} = \langle\rangle$

The conversation follows:

$$\begin{aligned}Con ::= &RR(\rho_C, \rho_M, price_{\vec{t_1},\vec{t_1}'}, \langle good, quantity\rangle, price, \mathbf{0})\\ &;OW(\rho_C, \rho_M, buy_{\vec{t_2}}, \langle confirm, \rho_C\rangle)\\ &;RR(\rho_B, \rho_C, bankData_{\vec{t_3},\vec{t_3}'}, \langle\rangle, \langle cctype, ccnumber\rangle, \mathbf{0})\\ &;(OW(\rho_B, \rho_C, bankCommit_{\vec{t_4}}, bcommit) \mid OW(\rho_S, \rho_C, suppCommit_{\vec{t_5}}, scommit)\end{aligned}$$

It is worth noting that, as we have said, the conversation is not connected but we accept this fact because we are focusing on the customer. In the following we report the initial knowledge:

$\mathcal{K} = \{I, \Lambda\}$

$I(\rho_C) = \{good, quantity, confirm, cctype, ccnumber\} \quad I(\rho_M) = \{price\}$

$I(\rho_S) = \{scommit\} \quad I(\rho_B) = \{bcommit\}$

$\Lambda(\rho_C) = \{\rho_C, \rho_M\} \quad \Lambda(\rho_M) = \emptyset$

$\Lambda(\rho_S) = \{\rho_C\} \quad \Lambda(\rho_B) = \{\rho_C\}$

It is worth noting that the location knowledge of the bank role and the supplier one contains the location of the customer. This is due to the fact that, at the present, we have designed only the interactions of the customer thus the bank and the supplier, which will receive the location of the customer from the market, for the sake of this step, must be aware of its location from the beginning. The initial constraints follows:

$$\begin{aligned} X := {} & good \in \{apple, banana\} \wedge 0 < quantity \leq 100 \wedge confirm \in \{yes, no\} \\ & \wedge cctype \in \{visa, mcard\} \wedge ccnumber = 1234 \wedge 50 < price < 200 \\ & \wedge scommit = SupMsg \wedge bcommit = BnkMsg \end{aligned}$$

Now, before designing the orchestrated system we design the joining function where four orchestrators are considered and some orchestrator variables are introduced:

$\Psi^1(C) = \rho_C \quad \Psi^1(M) = \rho_M$

$\Psi^1(S) = \rho_S \quad \Psi^1(B) = \rho_B$

$\Psi^2(C)(good_C) = good \quad \Psi^2(C)(quantity_C) = quantity$

$\Psi^2(C)(confirm_C) = confirm \qquad \Psi^2(C)(cctype_C) = cctype$

$\Psi^2(C)(ccnumber_C) = ccnumber \qquad \Psi^2(M)(price_M) = price$

$\Psi^2(B)(bcommit_B) = bcommit \qquad \Psi^2(S)(scommit_S) = scommit$

In order to be conformant the orchested system must introduce some coordinating interactions. We can choose to introduce a new orchestrator not joined to any role or we can introduce, for the sake of this step, some synchornization operations in order to follow the choreography conversation. Here, we choose the latter solution because, in the next steps we intend to exploit the synchronizing operations as a sort of roadmap which will allow us to replace each coordinating interactions with actual message exchanges among the roles. The orchestrated system follows:

$$
\begin{aligned}
\mathrm{C} ::= [(&\mathsf{myloc} := \mathrm{C}; \overline{\mathsf{price}}_{\vec{t_1},\vec{t_1}'}@\mathrm{M}(\langle \mathsf{good}_C, \mathsf{quantity}_C\rangle, \mathsf{price}_C)\\
&;\overline{\mathsf{buy}}_{\vec{t_2}}@\mathrm{M}(\langle \mathsf{confirm}_C, \mathsf{myloc}\rangle)\\
&;\mathsf{bankData}_{\vec{t_3},\vec{t_3}'}(\langle\rangle, \langle \mathsf{cctype}_C, \mathsf{ccnumber}_C\rangle, \mathbf{0})\\
&;(\mathsf{bankCommit}_{\vec{t_4}}(\mathsf{bcommit}_C) \mid \mathsf{suppCommit}_{\vec{t_5}}(\mathsf{scommit}_C)), \mathcal{S}_\perp)]_C
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{M} ::= [(&\mathsf{price}_{\vec{t_1},\vec{t_1}'}(\langle \mathsf{good}_M, \mathsf{quantity}_M\rangle, \mathsf{price}_M, \mathbf{0})\\
&;\mathsf{buy}_{\vec{t_2}}(\mathsf{confirm}_M, \mathsf{loc}_M)\\
&;\overline{\mathsf{s1}}_{t_0}@\mathrm{B}(\langle\rangle), \mathcal{S}_\perp)]_M
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{B} ::= [(&\mathsf{s1}_{t_0}(\langle\rangle); \overline{\mathsf{bankData}}_{\vec{t_3},\vec{t_3}'}@\mathrm{C}(\langle\rangle, \langle \mathsf{cctype}_B, \mathsf{ccnumber}_B\rangle)\\
&;\overline{\mathsf{s2}}_{t_0}@\mathrm{S}(\langle\rangle); \overline{\mathsf{bankCommit}}_{\vec{t_4}}@\mathrm{C}(\mathsf{bcommit}_B), \mathcal{S}_\perp)]_B
\end{aligned}
$$

$$
\mathrm{S} ::= [(\mathsf{s2}_{t_0}(\langle\rangle); \overline{\mathsf{suppCommit}}_{t_5}@\mathrm{C}(\mathsf{scommit}_S), \mathcal{S}_\perp)]_S
$$

where s1 and s2 are the operation we exploit for performing the coordinating interactions and $t_0 = \langle\rangle$. At this point, by observing the orchestrated system, we can decide to modify it, e.g. by introducing some new orchestrators as we have done in the previous example, or if we are satisfied, we can decide to wnrich the choreography and continue the design. For the sake of this example, we do not alter the orchestrated system, but we enhance the choreography by analyzing all the interactions of the market role.

9.2.2 Step 2 (the market)

Within this step, we want to introduce all the interactions of the market role. In particular, the market will ask for the price to the supplier and then forward it to the customer. Furthermore, if the customer will accept, the market will start the order by notifying the supplier and it will initiate the financial transaction by notifying the bank. Finally it will wait for a bank commit message from the bank role. We introduce the following operations:

$$\begin{aligned} Op := Op \cup \{ & (pricereq, rr, \vec{t_1}, \vec{t_1}'), (pricereq, sr, \vec{t_1}, \vec{t_1}') \\ & (order, ow, \vec{t_2}), (order, n, \vec{t_4}) \\ & (pay, ow, \vec{t_6}), (pay, n, \vec{t_6})\} \end{aligned}$$

where $t_6 = \langle inf, loc, loc \rangle$ is the template of the operation pay that will be exhibited by the bank role and which will receive the price and the locations of the customer and the market. The two locations will be exploited by the bank role for sending the commit messages. In Fig. 9.8 we report the communication links enriched with the new operations. It is worth noting that the market will receive the bank commit message on the operation $bankCommit$ previously defined.

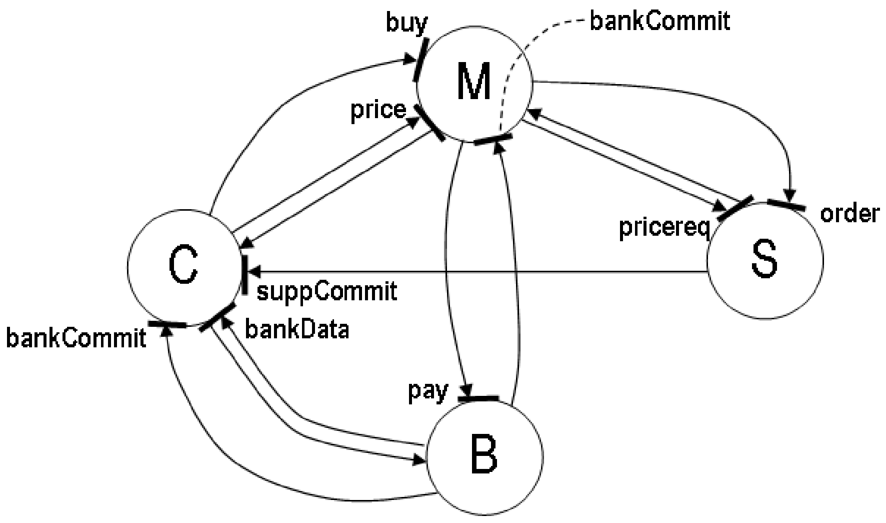

Figure 9.8: Market Example communication links Step 2

The role definition changes as it follows:

$$\begin{aligned} Role := \{ & (\rho_C, \{\overline{price}_{\vec{t_1}, \vec{t_1}'}, \overline{buy}_{\vec{t_2}}, bankData_{\vec{t_3}, \vec{t_3}'}, bankCommit_{\vec{t_4}}, suppCommit_{\vec{t_5}}\}), \\ & (\rho_M, \{\overline{pricereq}_{\vec{t_1}, \vec{t_1}'}, \overline{order}_{t_2}, price_{\vec{t_1}, \vec{t_1}'}, buy_{\vec{t_2}}, \overline{pay}_{t_6}, bankCommit_{\vec{t_4}}\}) \end{aligned}$$

$(\rho_B, \{\overline{bankData}_{\vec{t_3}, \vec{t_3}'}, \overline{bankCommit}_{\vec{t_4}}, pay_{t_6}\})$

$(\rho_S, \{pricereq_{\vec{t_1}, \vec{t_1}'}, order_{t_2}, buy_{\vec{t_2}}, price_{\vec{t_1}, \vec{t_1}'}, \overline{suppCommit}_{\vec{t_5}}\})\}$

Now we introduce the interactions related to the market within the conversation:

```
$Con ::= RR(\rho_C, \rho_M, price_{\vec{t_1}, \vec{t_1}'}, \langle good, quantity\rangle, price, PriceRequest)$
    $; OW(\rho_C, \rho_M, buy_{\vec{t_2}}, \langle confirm, \rho_C\rangle)$
    $; (OW(\rho_M, \rho_S, order_{\vec{t_2}}, \langle confirm, \rho_C\rangle)$
    $|$
      if $confirm == yes_{\rho_M}$ then
        $OW(\rho_M, \rho_B, pay_{\vec{t_6}}, \langle price, \rho_C, \rho_M\rangle)$
        $; RR(\rho_B, \rho_C, bankData_{\vec{t_3}, \vec{t_3}'}, \langle\rangle, \langle cctype, ccnumber\rangle, \mathbf{0}))$
        $; (OW(\rho_B, \rho_C, bankCommit_{\vec{t_4}}, bcommit)$
         $| OW(\rho_S, \rho_C, suppCommit_{\vec{t_5}}, scommit)$
         $| OW(\rho_B, \rho_M, bankCommit_{\vec{t_4}}, bcommit))$
      else
        $\mathbf{0}$
      )
```

$PriceRequest ::= RR(\rho_M, \rho_S, pricereq_{\vec{t_1}, \vec{t_1}'}, \langle good, quantity\rangle, price, \mathbf{0})$

The Request-Response interaction between the market and the supplier on the operation $pricereq$ is performed within the first Request-Response interaction between the customer and the market, and it is defined by means of the conversation $PriceRequest$. Furthermore, the market notifies the confirmation message to the supplier and it will initiate the financial transaction with the bank only if the message contains the value yes. If the financial transaction is initiated, the market will notifies the bank by means of the operation pay and it will wait for the commint on the $bankCommit$ operation. It is worth noting that the conversation is still not connected because, after the interaction on the pay operation, it starts the following conversation:

$; RR(\rho_B, \rho_C, bankData_{\vec{t_3}, \vec{t_3}'}, \langle\rangle, \langle cctype, ccnumber\rangle, \mathbf{0}))$

$; (OW(\rho_B, \rho_C, bankCommit_{\vec{t_4}}, bcommit)$

$\mid OW(\rho_S, \rho_C, \mathit{suppCommit}_{\vec{t_5}}, \mathit{scommit})$
$\mid OW(\rho_B, \rho_M, \mathit{bankCommit}_{\vec{t_4}}, \mathit{bcommit}))$

where the parallel composition of three interactions is enabled after the execution of the Request-Response one. Within the parallel composition, the sender of the One-Way $OW(\rho_S, \rho_C, \mathit{suppCommit}_{\vec{t_5}}, \mathit{scommit})$ is different from the last receiver (ρ_B) of the Request-Response interaction peformed on the operation $\mathit{bankData}$, thus introducing the not connection. This is due to the fact that, so far, we have not dealt with the interactions between the bank role and the supplier one that remain unspecified.

The initial knowledge is modified in order to take into account the fact that, now, the price is communicated by the supplier and the market only forwards it to the customer, futrthermore as far as the location knowledge is concerned, the market must be aware of the location of the supplier in order to interact with it, and the bank can be initialized with an empty set because it receives both the location of the customer and the market from the market itself by means of the notification pay. It is worth noting that, since the market has to communicate its own location, it must initially know it. Finally, the location knowledge of the supplier is initialized to the empty set because it will receive the customer location from the market by means of the operation order:

$\mathcal{K} = \{I, \Lambda\}$
$I(\rho_C) = \{\mathit{good}, \mathit{quantity}, \mathit{confirm}, \mathit{cctype}, \mathit{ccnumber}\} \quad I(\rho_M) = \emptyset$
$I(\rho_S) = \{\mathit{scommit}, \mathit{price}\} \quad I(\rho_B) = \{\mathit{bcommit}\}$
$\Lambda(\rho_C) = \{\rho_C, \rho_M\} \quad \Lambda(\rho_M) = \{\rho_S, \rho_M\}$
$\Lambda(\rho_S) = \emptyset \quad \Lambda(\rho_B) = \emptyset$

The initial constraints remain unchanged whereas the joining function must be modified in order to change the association of the information price. Now indeed, the price information is contained within the supplier and the customer only receives it by means of the operation $\mathit{pricereq}$, thus we join such an information with the variable price_S of the supplier instead of the variable price_C of the customer. For this reason, we modify the joining function definition of the previous step, by replacing the equation $\Psi^2(M)(\mathit{price}_M) =$

$\mathtt{price}$ with the following one: $\Psi^2(M)(\mathtt{price}_S) = \mathtt{price}$. Now, we can extract the orchestration system where we have introduced all the interactions of the market and we have replaced the coordinating interaction on $s1$ with the One-Way interaction on the operation $\mathtt{pay}$ whereas the interaction on the operation $s2$ still remains.

$$
\begin{aligned}
C ::= [&(\mathtt{myloc} := C; \overline{\mathtt{price}}_{\vec{t_1},\vec{t_1}'}@M(\langle \mathtt{good}_C, \mathtt{quantity}_C\rangle, \mathtt{price}_C)\\
&;\overline{\mathtt{buy}}_{\vec{t_2}}@M(\langle \mathtt{confirm}_C, \mathtt{myloc}\rangle)\\
&;\mathtt{bankData}_{\vec{t_3},\vec{t_3}'}(\langle\rangle, \langle \mathtt{cctype}_C, \mathtt{ccnumber}_C\rangle, \mathbf{0})\\
&;(\mathtt{bankCommit}_{\vec{t_4}}(\mathtt{bcommit}_C) \mid \mathtt{suppCommit}_{\vec{t_5}}(\mathtt{scommit}_C)), \mathcal{S}_\perp)]_C
\end{aligned}
$$

$$
\begin{aligned}
M ::= [&\mathtt{myloc} := M; \mathtt{supLoc} := S\\
&;(\mathtt{price}_{\vec{t_1},\vec{t_1}'}(\langle \mathtt{good}_M, \mathtt{quantity}_M\rangle, \mathtt{price}_M, \mathtt{PriceRequest}_M)\\
&;\mathtt{buy}_{\vec{t_2}}(\mathtt{confirm}_M, \mathtt{loc}_M)\\
&;(\overline{\mathtt{order}}_{\vec{t_2}}@\mathtt{supLoc}(\mathtt{confirm}_M, \mathtt{loc}_M)\\
&\quad\mid\\
&\quad\ \mathtt{confirm}_M == \mathtt{yes}?\\
&\quad\quad \overline{\mathtt{pay}}_{t_6}@B(\langle \mathtt{price}_M, \mathtt{loc}_M, \mathtt{myloc}\rangle)\\
&\quad\quad ;\mathtt{bankCommit}_{\vec{t_4}}(\mathtt{bcommit}_M))\\
&\quad\ : \mathbf{0}\\
&\ ), \mathcal{S}_\perp)]_M
\end{aligned}
$$

$$
\mathtt{PriceRequest}_M ::= (\overline{\mathtt{pricereq}}_{\vec{t_1},\vec{t_1}'}@\mathtt{supLoc}(\langle \mathtt{good}_M, \mathtt{quantity}_M\rangle, \mathtt{price}_M, \mathbf{0})
$$

$$
\begin{aligned}
B ::= [&\mathtt{pay}_{t_6}(\langle \mathtt{price}_B, \mathtt{loc}_1, \mathtt{loc}_2\rangle); \overline{\mathtt{bankData}}_{\vec{t_3},\vec{t_3}'}@\mathtt{loc}_1(\langle\rangle, \langle \mathtt{cctype}_B, \mathtt{ccnumber}_B\rangle)\\
&;\overline{\mathtt{s2}}_{t_0}@S(\langle\rangle)\\
&;(\overline{\mathtt{bankCommit}}_{\vec{t_4}}@\mathtt{loc}_1(\mathtt{bcommit}_B)\\
&\mid \overline{\mathtt{bankCommit}}_{\vec{t_4}}@\mathtt{loc}_2(\mathtt{bcommit}_B)), \mathcal{S}_\perp)]_B
\end{aligned}
$$

$$
\begin{aligned}
S ::= [&\mathtt{pricereq}_{\vec{t_1},\vec{t_1}'}(\langle \mathtt{good}_S, \mathtt{quantity}_S\rangle, \mathtt{price}_S); \mathtt{order}_{\vec{t_2}}(\mathtt{confirm}_S, \mathtt{loc}_S)\\
&;\mathtt{confirm}_S == \mathtt{yes}?\\
&\quad \mathtt{s2}_{t_0}(\langle\rangle); \overline{\mathtt{suppCommit}}_{t_5}@\mathtt{loc}_S(\mathtt{scommit}_S)\\
&: \mathbf{0}, \mathcal{S}_\perp)]_S
\end{aligned}
$$

As for the step 1, here we do not modify the orchestrated system but we decide to model also the bank and the supplier interactions within the choreography.

9.2.3 Step 3 (the bank and the supplier)

Within this step, we introduce all the interactions that deal with the bank and the supplier roles. In particular, we add the interaction between the bank and the supplier for retrieving the bank data of the latter and, moreover, we introduce the bank commit interaction between the bank and the supplier. In order to this, we do not need more operations because it sufficient to exploit the existing ones. Indeed, the supplier must exhibit the operations $\mathsf{bankData}$ and $\mathsf{bankCommit}$ as it is shown within the following definition of the roles:

$$
\begin{aligned}
\mathrm{Role} := \{&(\rho_C, \{\overline{\mathrm{price}}_{\vec{t_1}, \vec{t_1}'}, \overline{\mathrm{buy}}_{\vec{t_2}}, \mathrm{bankData}_{\vec{t_3}, \vec{t_3}'}, \mathrm{bankCommit}_{\vec{t_4}}, \mathrm{suppCommit}_{\vec{t_5}}\}),\\
&(\rho_M, \{\overline{\mathrm{pricereq}}_{\vec{t_1}, \vec{t_1}'}, \overline{\mathrm{order}}_{t_2}, \mathrm{price}_{\vec{t_1}, \vec{t_1}'}, \mathrm{buy}_{\vec{t_2}}, \overline{\mathrm{pay}}_{t_6}, \mathrm{bankCommit}_{\vec{t_4}}\})\\
&(\rho_B, \{\overline{\mathrm{bankData}}_{\vec{t_3}, \vec{t_3}'}, \overline{\mathrm{bankCommit}}_{\vec{t_4}}, \mathrm{pay}_{t_6}\})\\
&(\rho_S, \{\mathrm{pricereq}_{\vec{t_1}, \vec{t_1}'}, \mathrm{order}_{t_2}, \mathrm{buy}_{\vec{t_2}}, \mathrm{price}_{\vec{t_1}, \vec{t_1}'}, \overline{\mathrm{suppCommit}}_{\vec{t_5}}\\
&\quad, \mathrm{bankCommit}_{\vec{t_4}}, \mathrm{bankData}_{\vec{t_3}, \vec{t_3}'}\})\}
\end{aligned}
$$

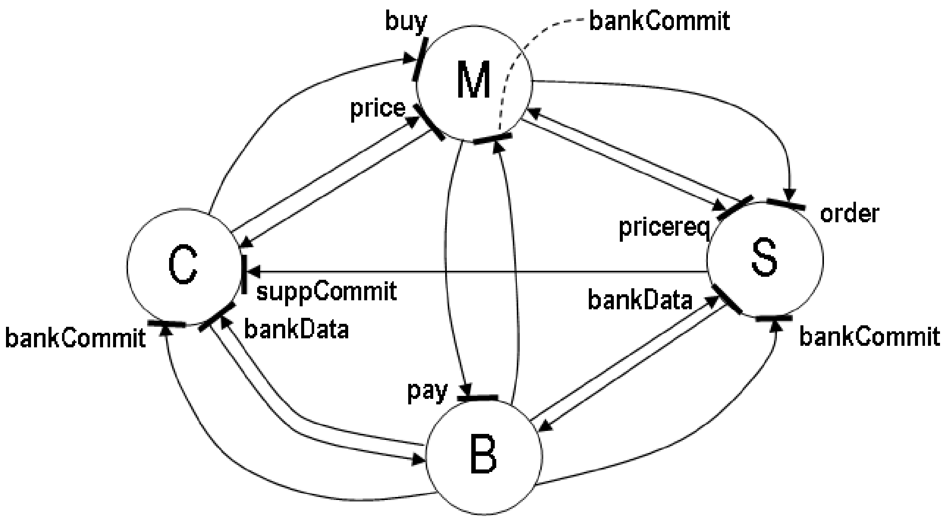

Figure 9.9: Market Example communication links Step 3

The communication links are represented within Fig. 9.9. It is worth noting that we only modify the template t_6 as it follows $t_6 = \langle inf, loc, loc, loc \rangle$ in order to consider the fact that, now, the bank requires also the supplier location in order to interact with it. Now, we introduce two new information that are related to the bank data of the supplier by redefining the information set as it follows:

$$\begin{aligned} I_C := \{&good, quantity, price, confirm, cctype, ccnumber \\ &, cctype2, ccnumber2, bcommit, scommit\} \end{aligned}$$

where $cctype2$ and $ccnumber2$ represent the credit card type and number of teh supplier respectively. The final conversation, for the business scenario we are modeling, follows:

```
$Con ::= RR(\rho_C, \rho_M, price_{\vec{t_1}, \vec{t_1}'}, \langle good, quantity \rangle, price, PriceRequest)$
      $; OW(\rho_C, \rho_M, buy_{\vec{t_2}}, \langle confirm, \rho_C \rangle)$
      $; (OW(\rho_M, \rho_S, order_{\vec{t_2}}, \langle confirm, \rho_C \rangle)$
       $|$
        if $confirm == yes_{\rho_M}$ then
           $OW(\rho_M, \rho_B, pay_{\vec{t_6}}, \langle price, \rho_C, \rho_M, \rho_S \rangle)$
           $; (RR(\rho_B, \rho_C, bankData_{\vec{t_3}, \vec{t_3}'}, \langle\rangle, \langle cctype, ccnumber \rangle, \mathbf{0})$
             $|$
            $RR(\rho_B, \rho_S, bankData_{\vec{t_3}, \vec{t_3}'}, \langle\rangle, \langle cctype2, ccnumber2 \rangle, \mathbf{0}))$
            $)$
           $; (OW(\rho_B, \rho_C, bankCommit_{\vec{t_4}}, bcommit)$
            $| (OW(\rho_B, \rho_S, bankCommit_{\vec{t_4}}, bcommit)$
                $; OW(\rho_S, \rho_C, suppCommit_{\vec{t_5}}, scommit))$
            $| OW(\rho_B, \rho_M, bankCommit_{\vec{t_4}}, bcommit)$
            $)$
        else
           $\mathbf{0}$
        $)$
```

$PriceRequest ::= RR(\rho_M, \rho_S, pricereq_{\vec{t_1}, \vec{t_1}'}, \langle good, quantity \rangle, price, \mathbf{0})$

The conversation is connected. The initial knowledge and the initial constraints are modified as it follows in order to take into account the supplier bank information:

$\mathcal{K} = \{I, \Lambda\}$
$I(\rho_C) = \{good, quantity, confirm, cctype, ccnumber\} \quad I(\rho_M) = \emptyset$
$I(\rho_S) = \{scommit, price, cctype2, ccnumber2\} \quad I(\rho_B) = \{bcommit\}$
$\Lambda(\rho_C) = \{\rho_C, \rho_M\} \quad \Lambda(\rho_M) = \{\rho_S, \rho_M\}$
$\Lambda(\rho_S) = \emptyset \quad \Lambda(\rho_B) = \emptyset$

$X := X \wedge cctype2 \in \{visa, mcard\} \wedge ccnumber2 = 5678$

As far as the joining funtion is considered, we must add the following definitions in order to consider the new information intriduce within this step:

$\Psi^2(S)(cctype_S) = cctype2 \qquad \Psi^2(S)(ccnumber_S) = ccnumber2$

The orchestrated system without coordinating interactions follows:

$C ::= [(myloc := C; \overline{price}_{\vec{t_1}, \vec{t_1}'}@M(\langle good_C, quantity_C\rangle, price_C)$
$\quad ; \overline{buy}_{\vec{t_2}}@M(\langle confirm_C, myloc\rangle)$
$\quad ; bankData_{\vec{t_3}, \vec{t_3}'}(\langle\rangle, \langle cctype_C, ccnumber_C\rangle, \mathbf{0})$
$\quad ; (bankCommit_{\vec{t_4}}(bcommit_C) \mid suppCommit_{\vec{t_5}}(scommit_C)), \mathcal{S}_\perp)]_C$

$M ::= [myloc := M; supLoc := S$
$\quad ; (price_{\vec{t_1}, \vec{t_1}'}(\langle good_M, quantity_M\rangle, price_M, PriceRequest_M)$
$\quad ; buy_{\vec{t_2}}(confirm_M, loc_M)$
$\quad ; (\overline{order}_{\vec{t_2}}@supLoc(confirm_M, loc_M)$
$\qquad \mid$
$\qquad confirm_M == yes?$
$\qquad\quad \overline{pay}_{t_6}@B(\langle price_M, loc_M, myLoc, suploc\rangle)$
$\qquad\quad ; bankCommit_{\vec{t_4}}(bcommit_M))$
$\qquad : \mathbf{0}$

$\qquad), \mathcal{S}_\perp)]_M$

$\mathit{PriceRequest}_M ::= (\overline{\mathit{pricereq}}_{\vec{t_1},\vec{t_1}'}@\mathit{supLoc}(\langle \mathit{good}_M, \mathit{quantity}_M \rangle, \mathit{price}_M, \mathbf{0})$

$B ::= [\mathit{pay}_{t_6}(\langle \mathit{price}_B, \mathit{loc}_1, \mathit{loc}_2, \mathit{loc}_3 \rangle);$

$\qquad (\overline{\mathit{bankData}}_{\vec{t_3},\vec{t_3}'}@\mathit{loc}_1(\langle\rangle, \langle \mathit{cctype}_B, \mathit{ccnumber}_B \rangle)$

$\qquad \mid \overline{\mathit{bankData}}_{\vec{t_3},\vec{t_3}'}@\mathit{loc}_3(\langle\rangle, \langle \mathit{cctype2}_B, \mathit{ccnumber2}_B \rangle)$

$\qquad)$

$\qquad ; (\overline{\mathit{bankCommit}}_{\vec{t_4}}@\mathit{loc}_1(\mathit{bcommit}_B)$

$\qquad \mid \overline{\mathit{bankCommit}}_{\vec{t_4}}@\mathit{loc}_2(\mathit{bcommit}_B)$

$\qquad \mid \overline{\mathit{bankCommit}}_{\vec{t_4}}@\mathit{loc}_3(\mathit{bcommit}_B)), \mathcal{S}_\perp)]_B$

$S ::= [\mathit{pricereq}_{\vec{t_1},\vec{t_1}'}(\langle \mathit{good}_S, \mathit{quantity}_S \rangle, \mathit{price}_S); \mathit{order}_{\vec{t_2}}(\mathit{confirm}_S, \mathit{loc}_S)$

$\qquad ; \mathit{confirm}_S == \mathit{yes}?$

$\qquad\quad \mathit{bankData}_{\vec{t_3},\vec{t_3}'}(\langle\rangle, \langle \mathit{cctype}_S, \mathit{ccnumber}_S \rangle)$

$\qquad\quad ; \mathit{bankCommit}_{\vec{t_4}}(\mathit{bcommit}_S); \overline{\mathit{suppCommit}}_{t_5}@\mathit{loc}_S(\mathit{scommit}_S)$

$\qquad : \mathbf{0}, \mathcal{S}_\perp)]_S$

At this point, the abstract processes of the orchestrated system are completed and conformant w.r.t. the choreography. Other steps can be done, for example we can introduce a register service joint with the market which can be exploited for retrieving different supplier locations. We can imagine indeed, that there exist different suppliers for different kind of goods. Moreover, we can better specify other aspects of the business scenario, such as the internal decision of the customer which chooses if it has to accept or not depending on the price (e.g. by introducing a conversation `if` $\mathit{price} < 100$ `then` $\mathit{confirm} :=_{\rho_C} \mathit{yes}$ `else` $\mathit{confirm} :=_{\rho_C} \mathit{no}$), etc. Here, for the sake of brevity, we do not introduce other designing steps that, however, can be achieved by following the same approaches we have used within the designing steps presented in the two examples of this chapter.

Chapter 10

New languages for programming SOC applications

In the previous chapters, we have analyzed orchestration and choreography as synergic approaches for distributed system design by following a formal approach. As it emerges, the orchestration represents, w.r.t. choreography, a refinement step towards the implementation of service oriented applications. Informally, if on the one hand choreography does not produce executable systems, on the other hand the orchestration makes it possible to program each service involved in the application. Here, we focus on orchestration, presenting JOLIE (Java Orchestration Language Interpreter Engine) which we have developed in order to animate orchestration programs written in a language based on SOCK. JOLIE ia an open source project[1] [Ope] whose syntax is C/Java-like in order to provide a more programmer friendly development environment differently from WS-BPEL which has a less human readable XML-based syntax. The peculiar and original characteristic of JOLIE is that it combines the solid mathematical basis provided by SOCK with a programmer friendly development and execution environment. Such a fact contrasts with most of the actual Web Services orchestration languages for which the formal operational semantics has been investigated and (partially) defined after the syntax. In particular, at the present, JOLIE implements the service behaviour part of SOCK where some new constructs have been added in order to deal with implementa-

[1]The first implementation of the JOLIE Java code has been developed by Fabrizio Montesi for his master degree at the Corso di Laurea in Scienze dell'informazione in Cesena.

tive aspects. JOLIE, indeed, implements also a timing statement `sleep(msec)` for programming processes that wait for a certain amount of milliseconds and it has contructs `in` and `out` for communicationg with the user by console which is also a novelty in the domain of orchestration languages. It is worth noting that JOLIE has been developed in a modular way which allows us to be protocol and communication medium independent. Such a fact, will allow us to simply extend the engine in order to run orchestrators that exploit different and heterogeneous communication medium such SOAP, Internet sockets, shared files, etc.

10.1 JOLIE language overview

JOLIE is an implementation of the service behaviour part of the SOCK language and it provides a C-like syntax for designing orchestrator services. A C-like syntax makes the language intuitive and easy to learn for a programmer customed to it. In the following we introduce some basics of the JOLIE language, except expression and condition syntaxes which are similar to that of C language.

10.1.1 Identifiers

An *identifier* (often abbreviated to *id*) is an unambiguous name stored in the orchestrator shared memory which identifies a location, an operation, a variable or a link. An identifier must match the following regular expression: `[a-zA-Z]([0-9a-zA-Z])*`. Some JOLIE statements require that the programmer provides a *list of identifiers*, which is formed by identifiers separated by commas (as ”identifier1, identifier2, a, b, c”). In the following, we refer to the list of identifiers by using the name *id list*.

10.1.2 Program structure

A JOLIE program structure is represented by the following grammar:

program::=
`locations` { *Locations-definition** }
`operations` { *Operations-declaration** }
`variables` { *Variables-declaration* }
`links` { *Links-declaration* }
definition*
`main` { *Process* }
definition*
definition:= `define` *id* { *Process* }

where we represent non-terminal symbols in italic and the Kleene star represents a zero or more time repetition. For the sake of clarity, the non-terminals *Locations-definition*, *Operations-declaration*, *Variables-declaration*, *Links-declaration* and *Process* are separately explained in the following.

10.1.2.1 Locations

JOLIE communications are socket based: an orchestrator waits for messages on a network port (the default is 2555[2]). In order to communicate with another orchestrator it is fundamental to know its hostname (or ip address) and the port it is listening to: these information are stored in a *location*. A location definition joins an identifier to a hostname and a port. The non-terminal follows:

Locations-definition:= *id*=*"hostname*:*port"*

where we do not define the *hostname* and the *port* non-terminals which must be intended as a representation of any hostname and any port respectively. In the following we present program fragment which shows a possible location declaration:

[2]the default port can be overridden by command line

```
locations {
   localUri = "localhost:2555",
   googleUri = "www.google.com:80",
   ipUri = "192.168.0.1:2556"
}
```

10.1.2.2 Operations

The operations represent the way a JOLIE orchestrator exploits for interacting with other orchestrators. We distinguish two types of operations:

- Input operations.
- Output operations.

The former represent the access points an orchestrator offers to communicate with it, whereas the latter are used to invoke input operations of another orchestrator. We distinguish two groups of input operations: One-Way and Request-Response. A One-Way operation simply waits for a message, while a Request-Response operation waits for a message, executes a code block and then sends a response message to the invoker. As far as output operations are concerned they can be a Notification or a Solicit-Response operation. The former is used to invoke a One-Way operation of another orchestrator, sending a message to it, while the latter is used to invoke a Request-Response operation. It is worth noting that a Solicit-Response operation, after sending the request message, is blocked until it receives the response one from the invoked service. The non-terminal follows:

Operations-declaration:= `OneWay:`*id list*
| `RequestResponse:`*id list*
| `Notification:`*id-assign list*
| `SolicitResponse:`*id-assign list*

id-assign:= *id*=*id*

By definition, input operations expect a list of identifiers, while the output ones expect a list of pairs *id*=*id* (we have identified such a list by using the notation *id-assign list*). As far as the output operations are concerned we distinguish between the operation name used within the orchestrator and the bound operation name of the invoked one. In a pair *idA*=*idB*, *idA* represents the internal operation name whereas *idB* the bound name of the external one to be invoked. Such a language characteristic allows us to decouple the orchestrator code from the external operation name binding. In the following a program fragment shows an example of operation declaration.

```
operations
{
OneWay:
   ow1
RequestResponse:
   rr1, rr2
Notification:
   n1 = serverOneWay1, n2 = serverOneWay2, n3 = serverOneWay3
SolicitResponse:
   sr1 = serverRequestResponse1
}
```

10.1.2.3 Variables

JOLIE variables are typeless. Implicit supported types are integers and strings. The variables declaration non-terminal requires only a list of identifiers which represent the shared memory variables. The definition follows:

Variables-declaration:= *id list*

In the following example three variables, *a*, *b* and *c* are declared:

```
variables {
     a, b, c
}
```

10.1.2.4 Links

Links model the SOCK signals and are used for internal parallel processes synchronization. As for variables the links declaration non-terminal requires only a list of identifiers where the *id*s will represent internal links used for synchronization purposes.

Links-declaration:= *id list*

In the following example two links, *link1* and *link2*, are declared:

```
links {
    link1, link2
}
```

10.1.2.5 Definitions

Definitions allows to define a procedure which will be callable by another one by exploiting the `call` statement. Each definition joins an identifier to a *Process*. Syntactically, a *Process* is a piece of code composed by JOLIE statements. Informally, the process defined within a definition can be viewed as the body of a C function. In the following we report an example where the procedure *calc* is defined. Its body is composed by two assignments on variables *a* and *b*:

```
define calc
{
   a = 5*2-9;;
   b = a * (2-1)
}
```

10.1.2.6 Main

The main block allows to define the process which will be run at the start of the program execution. Informally, it is comparable to the main function of a C program. In the following we report an example where, within the main procedure, the string *Hello, world!* is printed out on the user concole and there is a call at the procedure *calc* defined above:

```
main
{
    out( "Hello, world!" );;
    call(calc)
}
```

10.1.3 Statements

This paragraph shows a brief survey of JOLIE statements.

10.1.3.1 Program control flow statements

- `call(` *id* `)` : calls and executes the procedure which has been defined with the given identifier.
- `if (` *condition* `) {...} else if (` *condition* `) {...} else {...}`: condition statement
- `while(` *condition* `) {...}`: loop statement

10.1.3.2 Operation statements

- *id<id list>* : waits for a message for the OneWay operation declared in the operations block as *id*, and stores its values in the *id list* variables.
- *id<id list> <id list>* (*Process*) : waits for a message for the RequestResponse operation *id*, stores its values in the first *id list* variables, executes the code block *Process* and sends a response message containing the values of the second *id list* variables.
- *id@id<id list>* : uses the Notification operation represented by the first *id* to send a message which contains the values of the *id list* variables, to the orchestrator located at the second *id*. The second *id* can be a location declared in the locations block, or a variable containing a string that can be evaluated as a location. It is worth noting that such a feature allows to implement the location mobility. It is possible,

indeed, to receive a location which can be exploited for performing a Notification or a Solicit-Response.

- *id@id<id list> <id list>* : uses the SolicitResponse operation represented by the first *id* to send a message which contains the values of the first *id list* variables, to the orchestrator located by the second *id* (which can be, as for the Notification, a location or a variable). Once the message is sent, it waits for a response message from the invoked Request-Response and stores its values in the second *id list* variables.

10.1.3.3 Synchronizing statements

- `linkIn(` *id* `)` : linkIn and linkOut are used for parallel processes synchronization and must be always considered together. In particular the linkIn waits for a linkOut trigger on the same internal link identified by *id*. In case there are already one or more linkOut processes triggering for the same internal link, it synchronizes itself with one of them by following a non-deterministic policy.

- `linkOut(` *id* `)` : triggers for a linkIn synchronization on the same internal link identified by *id*. In case there are already one or more linkIn processes waiting for the same internal link, it synchronizes itself with one of them by following a non-deterministic policy.

10.1.3.4 Console input/output statements

- `in(` *variable id* `)` : waits for a console user input and stores it in the given variable.

- `out(` *expression* `)` : writes the evaluation of the given expression on the console (note that a variable can be considered as an expression).

10.1.3.5 Others

- `sleep(` *n* `)` : makes the current process sleeping for *n* milliseconds where *n* is a natural.

- `nullProcess` : no-op statement.[3]

10.1.4 Statement composers

As the SOCK calculus, JOLIE provides three ways to compose statements: sequence, parallelism and non-deterministic choice.

10.1.4.1 Sequence

Sequences are composed by exploiting the ;; operator. Let $x_1, x_2, \ldots, x_{n-1}, x_n$ be statements. Then, the sequential composition

$$x_1;;x_2;;\ldots;;x_{n-1};;x_n$$

executes x_1 and waits for it to terminate, then executes x_2 and waits for it to terminate and continues with this behaviour until it reaches the end of the sequence.

10.1.4.2 Parallel

Parallel processes are composed by exploiting the || operator. The || operator combines sequences (note a single statement is a sequence of one element). Let $s_1, s_2, \ldots, s_{n-1}, s_n$ be sequences. Then, the parallel composition

$$s_1||s_2||\ldots||s_{n-1}||s_n$$

executes every sequence in parallel. A parallel composition is terminated when all the sequences are terminated.

[3]the `nullProcess` statement is usually exploited within the RequestResponse when there is no need to execute anything before sending the response.

10.1.4.3 Non-deterministic choice

A non-deterministic choice can be expressed among different guarded branches by using the ++ operator. A branch guard can only be an input operation or a linkIn statement, whereas the branch can be any process. Let

$$(g_1, p_1), (g_2, p_2), \ldots, (g_{n-1}, p_{n-1}), (g_n, p_n)$$

be branches where g is the branch guard and p the guarded process. The syntax of the non-deterministic choice follows:

$$\texttt{[}g_1\texttt{]}p_1\texttt{++[}g_2\texttt{]}p_2\texttt{++...++[}g_{n-1}\texttt{]}p_{n-1}\texttt{++[}g_n\texttt{]}p_n$$

The guards are defined within square brackets. When a non-deterministic choice is programmed it makes the interpreter waiting for an input on one of its guards. Once an input has come, the related p process is executed and the other branches are deactivated.

10.1.4.4 Priority of the composers

The statement composers interpretation priority is: ;; || ++. In the following example, where A, B, C and D are statements, we show how priority works.

```
[req1<a>] A || B ;; C ++ [req2<b>] D ;; C ;; B || D
```

In this code fragment there is a non-deterministic choice between two branches guarded by two One-Way operations (`req1<a>` and `req2<b>`). By considering the operator priority the same code would be explicited as follows.

```
[input1](A || (B ;; C)) ++ [input2] ((D ;; C ;; B) || D)
```

10.1.5 Example

As a practical example, consider a scenario in which we have an orchestrator which acts as a service provider. The orchestrator declares a Request-Response operation, named *factorialRR*, which has the purpose to receive a number and, as a response, to send its factorial. Moreover, the orchestrator has to interact with a logging server in order to communicate its activity for constructing a statistic of its usage. The following code snippet shows the part of a possible implementation. For the sake of brevity, only the *main* procedure is shown.

```
main
{
   while( 1 ) {
       [ factorialRR< n >< result >( call( calcFactorial ) ) ]
           servedClients = servedClients + 1
       ++
       [ linkIn( logLink ) ]
           notifyActivity@logServerUri< servedClients >;;
           servedClients = 0
   }
   ||
   while( 1 ) {
       sleep( 60000 );; /* 60 seconds */
       linkOut( logLink )
   }
}
```

The main is composed by two processes in parallel. The former defines a non-deterministic choice between the One-Way on which the service can be accessed for returning the factorial calculation and the linkIn process defined on the internal link `logLink`. The linkOut process which triggers the internal link `logLink` is defined in the second parallel process which, every 60 seconds, interrupts the service for sending the number of the served clients to the logging service located at `logServerUri`.

10.2 JOLIE interpreter architecture

This section is devoted to briefly describe the architecture of JOLIE.

10.2.1 Structure overview

Figure 10.1 describes the JOLIE interpretation algorithm and the parts composing the interpreter. In order to explain how JOLIE works we proceed by describing the main

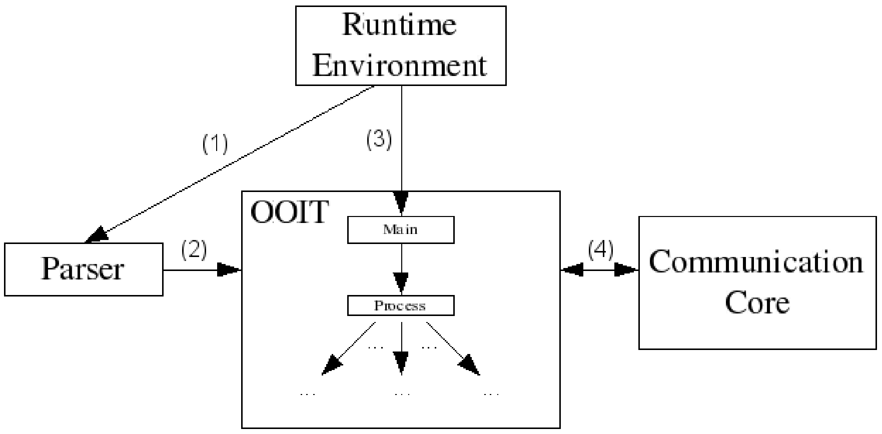

Figure 10.1: JOLIE architecture

steps of the run-time environment and then its main components: the Parser, the Object Oriented interpretation tree and then the Communication core.

JOLIE interpreter behavior

- Step 1: initialize the communication core.

- Step 2: create an instance of the parser.

- Step 3: create the Object Oriented Interpretation Tree (OOIT).

- Step 4: invoke the **run()** method of the OOIT's root node (that corresponds to the main).

We will now examine the different parts composing the interpreter.

10.2.2 Parser and Object Oriented Interpretation Tree

JOLIE is based on an object oriented infrastructure created during the parsing of the orchestration to be executed, which is realized by a recursive descendant parser. The principle we follow is to create objects as small as possible, which will know –abstracting away from the context– how to execute the simple task they represent. This goal is obtained by exploiting the encapsulation and composition mechanisms. In order to understand how this is realized we first introduce the main components present in the Object Oriented Interpretation Tree: the *Process* class and the *Basic Process* and *Composite Process* concepts. The former is an object class present in the implementation, while the latter are concepts which we will use to distinguish the general behaviour of *Process* objects.

10.2.2.1 The Process class

Process is a class representing a generic piece of JOLIE code. *Process* has a **run()** method which performs the activities that the object represents.

10.2.2.2 Basic Process

A *Basic Process* is a *Process* composed by a single statement of the JOLIE language, like an assignment operation, an output or an input one. The **run()** method in this case performs such a statement.

10.2.2.3 Composite Process

A *Composite Process* is a *Process* composing other *Process* objects (by running them in parallel, in a sequence or in a non-deterministic choice). The **run()** method executes such composition and, to this end, exploits the **run()** method of the enclosed *Process* objects.

Example 10.1 In order to illustrate how these concepts are used we use the following example:

```
a = 1 ;; out( a )
```

The parser will create three *Process* objects (see Figure 10.2):

- A *SequentialProcess* (which is as a *Composite Process*) object that encloses the following two processes:
  - An *AssignProcess* (which is a *BasicProcess*) object that assigns the value `1` to `a`.

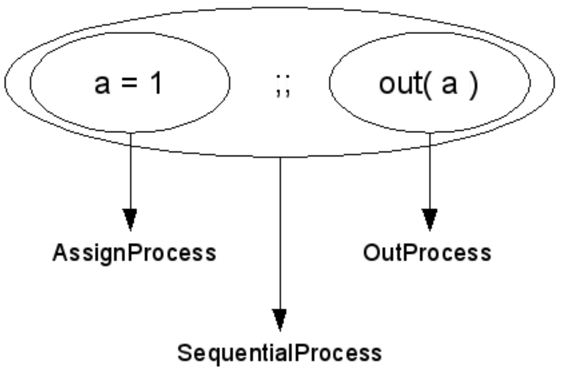

Figure 10.2: Objects tree representing `a = 1 ;; out( a )`

– An *OutProcess* (which is a *BasicProcess*) object that prints on the console the value of variable *a*.

When the runtime environment will interpret this code block, it will call the **run()** method of the *SequentialProcess* object which will sequentially call the **run()** method of the *Assign-Process* and the *OutProcess* objects it contains. It is worth noting that the *SequentialProcess* object knows only that its children are *Process* instances; it simply invokes their **run()** method without knowing anything about their behavior (e.g., they could be themselves *Composite Process* objects).

Since this process encapsulation principle is followed in the entire OOIT, to start the execution requires just the call of the **run()** method of the root node (which is the object that contains the `main` process).

10.2.3 The communication core

The communication core provides an interface for supporting the communication between services that allows us to abstract away from the following aspects:

- The communication medium.
- The communication data protocol.

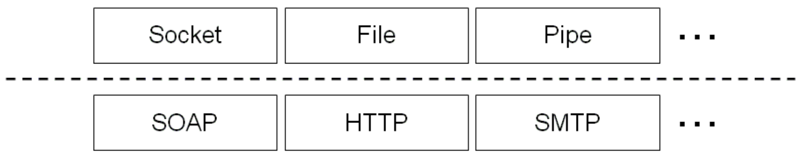

Figure 10.3: Communication medium and data protocols

Figure 10.3 reports some examples of communication medium and of communication data protocol. For instance the communication medium –which supports the communication– can be a socket, a file or a pipe, while the communication data protocol, which defines how the data should be formatted as well as the interaction modalities that should be used to implement a message exchange, can be (we list the most significant ones in the Internet context) HTTP, SMTP or SOAP.
The communication core supports such abstractions by means of the communication channel *CommChannel* object. The runtime environment exploits the communication channels to send and receive data. Once instantiated, a *CommChannel* object is able to send and receive *CommMessage* (communication message) objects that are composed by:

- The operation name.
- An array of values.

The idea is that each communication channel must be associated to a communication medium and a communication protocol. For instance consider a channel, say c, associated to the SOAP data protocol and to the FILE "host1@/home/services/op1.ss" communication medium. In order to send a message M on that channel a process must write on the file "host1@/home/services/op1.ss" the SOAP message containing M and, in order to perform an input on that channel, the process must read (and consume the piece of stream it reads) the "host1@/home/services/op1.ss" file by using the SOAP data protocol on the input stream. Although such a interface is designed to support such kind of flexibility on communication medium and data protocol, the current available version of the JOLIE interpreter supports only the socket communication medium and an internal default data protocol.

10.3 Modelling a WS-BPEL service with JOLIE

In this section we model the *Purchase order service* extracted from the WS-BPEL specifications and presented in section 2.1.3.3, by exploiting JOLIE. The aim of this example is to show that JOLIE programs reveals more human readable and manageable than WS-BPEL programs written in XML. The example models a service for handling a purchase order. The service starts its activity after the reception of a message on the Request-Response operation *sendPurchaseOrder*. Before sending the response message the service executes concurrently three processes defined within the *body* subroutine. One process selects a shipper by invoking the shipping service operation *requestShipping*, another process starts the price calculation by invoking the invoice service operation *InitiatePrice-Calculation* and the the third process starts the production scheduling by invoking the operation *requestProductionScheduling* of the production scheduling service. It is worth noting that we abstract away from service locations that are represented by the names *shippingServiceUri*, *InvoiceServiceUri* and *productionSchedulingService*. Furthermore, we remark the use of *linkOut* and *linkIn* statements for synchronizing concurrent processes.

```
locations {
   shUri = shippingServiceUri, inUri  = InvoiceServiceUri,
   schUri = productionSchedulingService
}

operations {
 OneWay: sendSchedule, sendInvoice
 Notification:
   InPr = InitiatePriceCalculation,
   SnShPr = sendShippingPrice,
   rqPrSch = requestProductionScheduling,
   snShSch = sendShippingSchedule
 RequestResponse: sendPurchaseOrder
 SolicitResponse: reqShp = requestShipping
}
variables { customerInfo, purchaseOrder, IVC, shippingInfo, scheduleInfo }
links { ship-to-invoice, ship-to-scheduling }
```

```
define body {
   reqShp@shUri< customerInfo >< shippingInfo > ;;
   linkOut( ship-to-invoice ) ;; sendSchedule< scheduleInfo > ;;
   linkOut( ship-to-scheduling )
   ||
   InPr@inUri< customerInfo, purchaseOrder > ;;
   linkIn( ship-to-invoice ) ;; SnShPr@inUri< shippingInfo > ;;
   sendInvoice< IVC >
   ||
   rqPrSch@schUri< customerInfo, purchaseOrder > ;;
   linkIn( ship-to-scheduling ) ;; snShSch@schUri< shippingInfo >
}
main {
   sendPurchaseOrder< customerInfo, purchaseOrder >< IVC >( call( body ) )
}
```

10.4 The market example

In this section we present a complete orchestrated system designed with JOLIE. The business scenario resembles that presented in Section 9.2 where a market example is presented. Here, we built a system with six orchestrators where we suppose that all the orchestrators are hosted by the localhost but on different ports. The presented system can be executed by downloading the JOLIE libraries at [Ope]:

- the client, port 2555
- the market, port 2556
- the register, port 2557
- the apple supplier, port 2558
- the banana supplier, port 2559

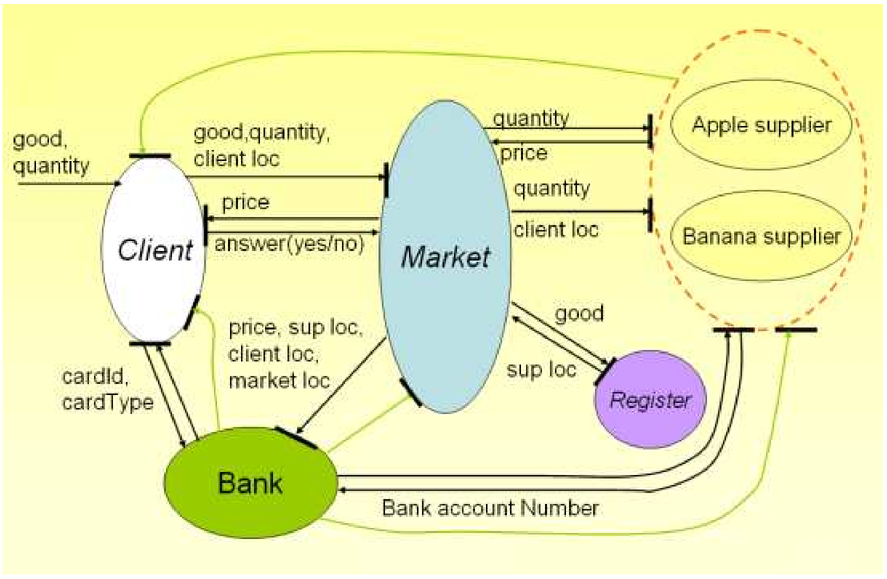

Figure 10.4: The JOLIE Market Example

- the bank supplier, port 2560

In Fig. 10.4 we reppresent the communication links among the orchestrator where, for the sake of this example, we do not explicit the operation names but we specify the exchanged data and we exploit the green arrows for representing the commit messages. The system behaviour can be summarized as if follows:

- the client asks to the user for inserting a good and a quantity and invokes the market by forwarding these information
- the market invokes the register, by sending the good, in order to retrieve the location of the supplier which is able to perform the order for that kind of good
- when the market receives the response from the register, it invokes the supplier by sending the quantity in order to receive the price order
- the supplier replies to the market by sending the order price
- the market forwards the price to the client
- the client asks to the user if he accepts or not the order

- if the client accepts the order, the market sends the order request to the supplier by sending the quantity and the location of the client that will be used, by the supplier, for sending the commit message. Concurrently, the market initiates a financial transaction by invoking the bank
- the bank invokes both the client and the supplier for retrieving their bank data
- the client asks for the bank data to the user (credit card type and credit card number) whereas the supplier is modelled in a way that it just contains the bank account number
- when the bank has obtained all the data, it sends a commit message to the client, to the market and to the supplier
- Finally, the supplier sends a commit message to the client.

In the following we report the code of each orchestrator with some comments.

10.4.1 The client

The code of the client follows:

```
locations {
        marketLoc = "localhost:2556"
}

operations
{
OneWay:
        BankCommitOW,
        SupplierCommitOW
Notification:
        PurchReqN = PurchReqOW
RequestResponse:
        DoYouAcceptRR,
        GetClientDataRR
SolicitResponse:
}

links {}

variables {
        good, quantity, price, answer, cardId, cardType, myLoc
}

main
{
        myLoc = "localhost:2555" ;;
        out( "Good: " ) ;; in( good ) ;;
        out( "Quantity: " ) ;; in( quantity ) ;;
        PurchReqN@marketLoc< good, quantity, myLoc >;;
        DoYouAccetptRR<price><answer>(
                out("The price is " + price + ". Do you accept?[yes/no]")
                ;; in(answer)
        );;
        if (answer=="yes"){
                GetClientDataRR<><cardId, cardType>(
                        out("Insert your card Id: ");; in(cardId);;
                        out("Insert your card type ");; in(cardType)
                )
                ;;
                (
                        (BankCommitOW<>;;out("\n Received Bank Commit"))
                ||
                        (SupplierCommitOW<>
                         ;;out("\n Received Supplier Commit"))
                )
        }
}
```

The client exploit the primitive `in` for requesting the good and the quantity to the user, then it invokes the market by exploiting the Notification `PurchReqN`. After that, it waits for the price on the Request-Response `DoYouAccept` and, before replying, it asks to the user if he wants to accept or not. If the user accepts the order, the client waits for a request from the bank (on the Request-Response operation `GetClientDataRR`) in order to provide its bank data and then waits for commit messages both from the bank (on operation `BankCommitOW`) and the supplier (on operation `SupplierCommitOW`).

10.4.2 The market

The code of the market follows:

```
locations {
        RegLoc = "localhost:2557",
        bankLoc = "localhost:2560"
}

operations
{
OneWay:
        PurchReqOW,
        BankCommitOW,

RequestResponse:

SolicitResponse:
        SuppLocSR = SuppLocRR,
        GetPriceSR = GetPriceRR,
        DoYouAccetptSR = DoYouAccetptRR
Notification:
        PurchaseOrderN = PurchaseOrderOW,
        TransactionN = TransactionOW
}

links {}

variables {
        good, quantity, SupLoc, clientLoc, price, answer, myLoc
}
```

```
main
{
        myLoc="localhost:2556";;
        PurchReqOW<good,quantity, clientLoc>;;
        SuppLocSR@RegLoc<good><SupLoc>;;
        GetPriceSR@SupLoc<quantity><price>;;
        DoYouAcceptptSR@clientLoc<price><answer>;;
        if (answer=="yes"){
                (
                        (PurchaseOrderN@SupLoc<quantity,clientLoc>;;
                         out("Sending Purchase order to "+good+" supplier"))
                ||
                        (TransactionN@bankLoc<price,SupLoc,clientLoc,myLoc>;;
                        out("\n Waiting for Bank Commit...");;
                        BankCommitOW<>)
                )
        }
}
```

The market starts its activities by receiving a request from a client on the operation `PurchReqOW`. After that, it invokes the register in order to obtain the location of the supplier joined to the order good (on the Solicit-Response operation `SuppLocSR`). Once received the location, it asks to the supplier for the price (operation `GetPriceSR`) and then, asks to the client for accepting or not the order (operation `DoYouAcceptSR`). If the client accepts the order, the market invokes both the supplier (operation `PurchaseOrderN`) and the bank (operation `TransactionN`) for starting the order and the financial transaction, respectively. Finally, it waits for a commit from the bank on the operation `BankCommitOW`.

10.4.3 The register

The code of the register follows:

```
locations {}
```

```
operations
{
RequestResponse:
        SuppLocRR
}

links { stop    }

variables {
        good, SupLoc, exit, cmd
}

main
{
        exit="true";;
        (
                in(cmd);;
                if (cmd=="exit") {linkOut(stop)}
        ||

                while(exit=="true"){
                        ([SuppLocRR<good><SupLoc>(
                                if (good=="apple"){
                                        SupLoc="localhost:2558"
                                };;
                                if (good=="banana"){
                                        SupLoc="localhost:2559"
                                }
                          )]nullProcess
                        ++
                        [linkIn(stop)]exit="false"
                        )
                }
        )
}
```

This orchestrator represents a service register where, by sending a query, it is possible to retrieve the location of a service which is able to perform a task realted to the sent query. Here, we simply models such a kind of service with an orchestrator that receives a good and then, by means of a simple `if then else` construct, replies with the supplier location joined with that good. In particular, if the selected good is `apple` then the location of the supplier is `localhost:2558`, whereas if teh selected good is `banana` the supplier location is `localhost:2559`. It is worth noting that the main code is structured, by exploiting a `while` construct, in a way that the service is always available except when it is stopped by means of a console command (`exit`). The body of the while operator indeed, contains a non-deterministic choice between the operation that supplies the retrieving location service (`SupLocRR`) and an internal synchronization signal on the link `stop`. The latter link is raised when the user insert the console command `exit`. This particular characteristic of JOLIE cannot be implemented by using WS-BPEL because on the one hand, WS-BPEL does not support primitives which deal with user interactions and, on the other hand, it does not allow for a not-deterministic choice between an external message and an internal link.

10.4.4 The suppliers

Here we present the code of the apple supplier. It is worth noting that all the suppliers for different kind of good have always the same code.

```
locations {}

operations
{
OneWay:
        PurchaseOrderOW,
        BankCommitOW
RequestResponse:
        GetSupplierDataRR,
        GetPriceRR
Notification:
        SupplierCommitN = SupplierCommitOW
}

links {
        stop
}

variables {
        exit, quantity, price, cmd, clientLoc, accountNumber
}

main
{
        accountNumber="123456";;
        exit="true";;
        (
                in(cmd);;
                if (cmd=="exit") {linkOut(stop)}
        ||
                while (exit=="true"){

                        ([GetPriceRR<quantity><price>(price=100*quantity)]
                         out("\n submitted price:" + price)
                        ++
                        [PurchaseOrderOW<quantity,clientLoc>](
                        out("\n Received purchase order!");;
                        GetSupplierDataRR<><accountNumber>(nullProcess);;
                        out("\n Bank account number sent.
                          Waiting for Bank Commit...");;
                        BankCommitOW<>;;out("...received Bank Commit");;
                        SupplierCommitN@clientLoc<>;;
                        out("\n Client commit sent.")
                        )
                        ++
                        [linkIn(stop)]exit="false"
                  )
                }
        )
}
```

Similarly to the register, the supplier is structured, by exploiting a while and a not-determistic constructs, in a way that it can be stopped by means of the console command `exit`. In particular, within the non-deterministic choice the following branches can be executed:

- it is possible, by means of the operation `GetPriceRR`, to request for an order price
- it is possible to purchase an order by means of the operation `PurchaseOrderOW`. In particular, when the order is started, the supplier also waits for a request from the bank in order to provide its bank account data (operation `GetSupplierDataRR`) and, finally, it waits for the bank commit on operation `BankCommitOW` and then sends teh commit message to the client (operation `SupplierCommitN`)
- it is possible to stop the execution of the orchestrator by inserting the console command `exit`

10.4.5 The bank

The code of the bank follows:

```
locations {}

operations
{
OneWay:
        TransactionOW
Notification:
        BankCommitN = BankCommitOW
SolicitResponse:
        GetClientDataSR = GetClientDataRR,
        GetSupplierDataSR = GetSupplierDataRR
}

links {}

variables {
        amount, SupLoc, clientLoc, marketLoc, cardId, cardType, accountNumber
}

main
{
        TransactionOW<amount, SupLoc, clientLoc, marketLoc>;;
        (
                (GetClientDataSR@clientLoc<><cardId, cardType>;;
                out("\n Received client data. CardId:" + cardId
                  + ", card Type:" + cardType))
        ||
                (GetSupplierDataSR@SupLoc<><accountNumber>;;
                out("\n Received account number of supplier:"
                  + accountNumber))
        );;
        out("\n Received both client and supplier data.
          Processing transaction of " + amount + "euros...");;
        /* internal check, not modelled */
        (
        (BankCommitN@clientLoc<>;;out("\n Client commit sent"))
        ||
        (BankCommitN@SupLoc<>;;out("\n Supplier commit sent"))
        ||
        (BankCommitN@marketLoc<>;;out("\n Market commit sent"))
        );;
        out("\n Transaction Completed Succesfully!")
}
```

The bank starts its activities when it is invoked for a financial transaction on the operation `TransactionOW`. When the transaction is initiated, the bank requests for both the cliente and the supplier data (operations `GetClientDataSR` and `GetSupplierDataSR`). Finally, it sends commit messages to the client, the supplier and the market by means of the same operation `BankCommitN`.

Chapter 11

Conclusions and future works

We have started this work from the perspective that Service Oriented Computing is a new programming paradigm which deals with distributed applications over the Internet, and we have considered Web Services as the most credited technology in this setting. In particular, we have focused on service design and service composition issues where the former deals with the design of the behaviour of a service, whereas the latter deals with the composition of a number of services into a system. On the one hand, we have investigated the basic characteristic of the SOC paradigm in a formal way, whereas, on the other hand, we have proposed a new services system design approach, called *bipolar approach* which is based upon a formal framework. In the following we remind the most important contributes of this thesis:

- We have defined a core language, equipped with a formal semantics, which deals with both service design and composition. It is called SOCK and it is structured on three layers: the service behaviour, the service engine and the services system. The service behaviour models the internal of a service, the service engine models the execution of a service behaviour by means of an engine and the services system models a system composed by more than one service.

- Starting from SOCK, we have discussed the mobility mechanisms in Service Oriented Computing and we have shown that there are four kind of mobility mechanisms: the internal state mobility, the location mobility, the interface mobility and the service behaviour mobility. Furthermore, we have shown as Web Services technology supplies only internal state mobility and location mobility.

- From the experience of SOCK, we have extracted a general model for Service Oriented Computing where the service design can be represented as a finite state automaton, the service engine as a formal machinery able to execute infinitely often some service behaviour sessions and the services system as a concurrent system modelled with a process algebra approach.

- We have considered the services system design issue by following the so-called bipolar approach where two kind of languages are exploited for designing and composing services: the choreography language and the orchestration one.

- We have presented a formal language, equipped with a formal semantics, which deals with choreography. It is called C_L and it is inspired to WS-CDL.

- We have exploited a subpart of SOCK for dealing with the orchestration language.

- We have developed a notion of conformance, which resembles a bisimulation relation, that allows us to state if an orchestrated system is coherent with a given choreography.

- Finally, we have started to develop an interpreter implementation for SOCK, called JOLIE, which aims at being a concrete language for programming orchestrators. JOLIE supplies a C/Java-like syntax in order to provide an easy way for dealing with service design differently from XML-based languages like WS-BPEL. JOLIE is an open source project.

In the future we intend to continue our investigation by following two directions: on the one hand, we intend to continue the formalization of Service Oriented Computing by considering all the other aspects which deal with security transactions, long running transactions, etc, whereas on the other hand, we intend to develop a user-friendly tool for dealing with the bipolar approach. In particular:

- We intend to enhance SOCK with primitives for dealing with faults, long running transactions and security aspects.
- We intend to enhance C_L and the conformance notion for dealing with session management and all the new future features of SOCK.
- We intend to develop a concrete language for choreography as we have done for the orchestration one by means of JOLIE.
- As far as the bipolar approach is concerned, we intend to develop the folowing algorithms:
  - an algorithm for extracting an orchestrated system skeleton from a choreography
  - an algorithm for extracting a choreography from an orchestrated system
  - an algorithm for testing the conformance between an orchestrated system and a choreography
- We intend to enhance JOLIE in order to deal with all the aspects of the SOCK calculus

References

[act] ActiveBPEL Open Source Engine. [http://www.active-endpoints.com/active-bpel-engine-overview.htm].

[AKR^{+}05] M. Acharya, A. Kulkarni, R.Kuppili, R. Mani, N. More, S. Narayanan, P. Patel, K. W. Schuelke, and S. N. Subramanian. Soa in the real world - experiences. In *Proc. of 3rd International Conference on Service Oriented Computing (ICSOC'05)*, volume 3826 of *LNCS*, pages 437–449, 2005.

[Apaa] Apache ODE. [http://incubator.apache.org/ode/index.html].

[Apab] Apache. *Axis (Java2WSDL)*. [http://ws.apache.org/axis/index.html].

[Apac] Apache. *Axis (WSDL2Java)*. [http://www.w3.org/TR/ws-arch/].

[BB05] A. Barros and E. Borger. A compositional framework for service interaction patterns and interaction flows. In *Proc. of International conference on formal engineering methods (ICFM 2005)*, LNCS, pages 5–35. Springer Verlag, 2005.

[BBC^{+}06] M. Boreale, R. Bruni, L. Caires, R. De Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V. Vasconcelos, and G. Zavattaro. SCC: A Service Centered Calculus. In *Proc. of Web Services and Formal Methods Workshop (WS-FM'06)*, volume 4184 of *LNCS*, pages 38–56. Springer-Verlag, 2006.

[BBM^{+}05a] M. Baldoni, C. Badoglio, A. Martelli, V. Patti, and C. Schifanella. Verifying the conformance of web services to global interaction protocols: a first step.

In *Proc. of Web Services and Formal Methods Workshop (WS-FM'05)*, volume 3670 of *LNCS*, pages 257–271. Springer-Verlag, 2005.

[BBM$^+$05b] Matteo Baldoni, Cristina Baroglio, Alberto Martelli, Viviana Patti, and Claudio Schifanella. Verifying the conformance of web services to global interaction protocols: A first step. In *EPEW/WS-FM*, volume 3670 of *LNCS*, pages 257–271. Springer, 2005.

[BCNR06] Mario Bravetti, Adalberto Casalboni, Manuel Nunez, and Ismael Rodriguez. From theoretical e-barter models to an implementation based on Web Services. In *In Proc. of thr first IPM international workshop on Foundations of software Engineering (FSEN'05)*, volume 159 of *ENTCS*, pages 241–264. Elsevier-Science, 2006.

[BCPV04] A. Brogi, C. Canal, E. Pimentel, and A. Vallecillo. Formalizing web services choreographies. In M. Bravetti and G. Zavattaro, editors, *Proc. of 1st International Workshop on Web Services and Formal Methods (WS-FM 2004)*, volume 105 of *ENTCS*. Elsevier, 2004.

[BDtH] A. Barros, M. Dumas, and A. H.M. ter Hofstede. Service interaction patterns: Towards a reference framework for service-based business process interconnection. *Tech. Report FIT-TR-2005-02,Faculty of information Technology, Queensland University of technology, Brisbane, Australia, March 2005.*

[BGG$^+$05a] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Towards a formal framework for Choreography. In *Proc. of 3rd International Workshop on Distributed and Mobile Collaboration (DMC 2005)*, pages 107–112. IEEE Computer Society Press, 2005.

[BGG$^+$05b] Nadia Busi, Roberto Gorrieri, Claudio Guidi, Roberto Lucchi, and Gianluigi Zavattaro. Choreography and orchestration: A synergic approach for system design. In *ICSOC'05*, volume 3826 of *LNCS*, pages 228–240, 2005.

[BGG$^+$06] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and orchestration conformance for system design. In *Proc. of 8th International Conference on Coordination Models and Languages (COORDINATION'06)*, volume 4038 of *LNCS*, pages 63–81, 2006.

[BKZ05] Mario Bravetti, Leïla Kloul, and Gianluigi Zavattaro, editors. *Formal Techniques for Computer Systems and Business Processes, European Performance Engineering Workshop, EPEW 2005 and International Workshop on Web Services and Formal Methods, WS-FM 2005, Versailles, France, September 1-3, 2005, Proceedings*, volume 3670 of *Lecture Notes in Computer Science*. Springer, 2005.

[BP06] Antonio Brogi and Razvan Popescu. Automated generation of bpel adapters. In *Service-Oriented Computing - ICSOC 2006, 4th International Conference, Chicago, IL, USA, December 4-7, 2006, Proceedings*, volume 4294 of *LNCS*, pages 27–39, 2006.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge University Press, 1990.

[CFNS05] Francisco Curbera, Donald Ferguson, Martin Nally, and Marcia L. Stockton. Toward a programming model for service-oriented computing. In *Proc. of 3rd International Conference on Service Oriented Computing (ICSOC'05)*, volume 3826 of *LNCS*, pages 33–47, 2005.

[CGK$^+$] F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. Business Process Execution Language for Web Services (BPEL4WS 1.1). [http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/], 2002.

[CHT] K. Channabasavaiah, K. Holley, and E. Tuggle. Migrating to a Service Oriented Architecture, Part I. [http://www-128.ibm.com/developerworks/library/ws-migratesoa], 16 December 2003.

[CHY07] M. Carbone, K. Honda, and N. Yoshida. Structured Communication-Centred Programming for Web Services. In *ESOP*, 2007. To appear.

[CNM06] Massimiliano Colombo, Elisabetta Di Nitto, and Marco Mauri. Scene: A service composition execution environment supporting dynamic changed disciplined through rules. In *Service-Oriented Computing - ICSOC 2006, 4th International Conference, Chicago, IL, USA, December 4-7, 2006, Proceedings*, volume 4294 of *LNCS*, pages 191–202, 2006.

[Coh] F. Cohen. Understanding Web Service interoperability. [http://www-128.ibm.com/developerworks/webservices/library/ws-inter.html], 1 February 2002.

[Con] World Wide Web Consortium. Web service choreography interface (wsci) 1.0. [http://www.w3.org/TR/wsci], 2002.

[DD04] Remco Dijkman and Marlon Dumas. Service-oriented Design: a Multi-viewpoint Approach. *Int. J. Cooperative Inf. Syst.*, 13(4):337–368, 2004.

[DFS06] Schahram Dustdar, Jos Luiz Fiadeiro, and Amit P. Sheth, editors. *Business Process Management, 4th international conference, BPM 2006, Vienna, Austria, September 5-7, 2006, Proceedings*, volume 4102 of *Lecture Notes in Computer Science*. Springer, 2006.

[DL06] Asit Dan and Winfried Lamersdorf, editors. *Service-Oriented Computing - ICSOC 2006, 4th International Conference, Chicago, IL, USA, December 4-7, 2006, Proceedings*, volume 4294 of *Lecture Notes in Computer Science*. Springer, 2006.

[DMK$^+$] S. Durvasula, M.Guttmann, A. Kumar, J. Lamb, T. Mitchell, B. Oral, Y. Pai, T. Sedlack, H. Sharma, and S.R. Sundaresan. SOA Practitioners' guide, Part 1, Why service oriented architecture? [http://dev2dev.bea.com/pub/a/2006/09/soa-practitioners-guide.html], Dev2Dev, 2006.

[DZD06] Gero Decker, Johannes Maria Zaha, and Marlon Dumas. Execution semantics for service choreographies. In Mario Bravetti, Manuel Núñez, and Gianluigi Zavattaro, editors, *WS-FM*, volume 4184 of *Lecture Notes in Computer Science*, pages 163–177. Springer, 2006.

[FLB06] Jos Luiz Fiadeiro, Antonia Lopes, and Laura Bocchi. A formal approach to service component architecture. In *Proc. of Web Services and Formal Methods Workshop (WS-FM'06), Vienna, Austria, September 2006*, volume 4184 of *LNCS*, pages 193–213. Springer-Verlag, 2006.

[Fos06] Howard Foster. *A Rigorous Approach To Engineering Web Service Compositions*. PhD. thesis, Imperial College London, University of London, Department of Computing, 2006.

[GGL05] R. Gorrieri, C. Guidi, and R. Lucchi. Reasoning on the interaction patterns in choreography. In *Proc. of Web Services and Formal Methods Workshop (WS-FM'05)*, volume 3670 of *LNCS*, pages 333–348. Springer-Verlag, 2005.

[GL06] C. Guidi and R. Lucchi. Mobility mechanisms in service oriented computing. In *Proc. of 8th International Conference on on Formal Methods for Open Object-Based Distributed Systems (FMOODS'06)*, volume 4037 of *LNCS*, pages 233–250, 2006.

[GLG$^+$06] Claudio Guidi, Roberto Lucchi, Roberto Gorrieri, Nadia Busi, and Gianluigi Zavattaro. Sock: A calculus for service oriented computing. In *Service-Oriented Computing - ICSOC 2006, 4th International Conference, Chicago, IL, USA, December 4-7, 2006, Proceedings*, volume 4294 of *LNCS*, pages 327–338, 2006.

[HM05] R. Heckel and L. Mariani. Automatic Conformance Testing of Web Services. In *Proc. of Fundmental Approaches to Software Engineering (FASE'05)*, volume 3442 of *LNCS*, pages 34–48. Springer-Verlag, 2005.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[IBMa] IBM. *Web Secure Conversation Language (WS-SecureConversation).* ftp://www6.software.ibm.com/software/developer/library/ws-secureconversation.pdf.

[IBMb] IBM. *Web Services Trust Language (WS-Trust).* ftp://www6.software.ibm.com/software/developer/library/ws-trust.pdf.

[Kel76] Robert M. Keller. Formal verification of parallel programs. *Commun. ACM*, 19(7):371–384, 1976.

[Koc] C. Koch. A new blueprint for the enterprise. [http://www.cio.com/archive/030105/blueprint.html], 1 March 2005.

[KP06] Raman Kazhamiakin and Marco Pistore. Choreography conformance analysis: asynchronous communication and information alignment. In *Proc. of Web Services and Formal Methods Workshop (WS-FM'06)*, volume 4184 of *LNCS*, pages 227–241. Springer-Verlag, 2006.

[Ley] F. Leymann. Web Services Flow Language (WSFL 1.0). [http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf], Member IBM Academy of Technology, IBM Software Group, 2001.

[LM07] R. Lucchi and M. Mazzara. A pi-calculus based semantics for WS-BPEL. *Journal of Logic and Algebraic Programming*, volume 70 Issue 1, Web Services and Formal Methods:pages 96–118, January 2007.

[LPT06] A. Lapadula, R. Pugliese, and F. Tiezzi. A wsdl-based type system for ws-bpel. In Springer and Verlag, editors, *Proc. of 8th International Conference on Coordination Models and Languages (COORDINATION'06)*, volume 4038 of *LNCS*, pages 145–163, 2006.

[MC06] J. Misra and W. Cook. Computation orchestration, a basis for wide-area computing. *Journal of Software and Systems modeling*, 2006. To appear.

[MCY] K. Honda M. Carbone and N. Yoshida. Programming interaction with types. [http://www.w3.org/2002/ws/chor/5/06/F2FJune14.pdf], W3C WS-CDL WG London F2F, June 14 2002.

[MGLZ] F. Montesi, C. Guidi, R. Lucchi, and G. Zavattaro. JOLIE: a Java Orchestration Language Interpreter Engine. In *CoOrg06*, volume to appear of *ENTCS*.

[Mica] Microsoft, BEA, IBM. *Web Services Coordination*. [http://www-106.ibm.com/developerworks/library/ws-coor/].

[Micb] Microsoft, BEA, IBM. *Web Services Transactions*. [http://www-106.ibm.com/developerworks/webservices/library/ws-transpec/].

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts 1 and 2. *Inf. Comput.*, 100(1):1–77, 1992.

[MS06] Carlo Montangero and Laura Semini. A logical view of choreography. In Paolo Ciancarini and Herbert Wiklicky, editors, *COORDINATION*, volume 4038 of *Lecture Notes in Computer Science*, pages 179–193. Springer, 2006.

[OASa] OASIS. Reference Model for Service Oriented Architecture v. 1.0. [http://www.oasis-open.org/committees/download.php/19679/soa-rm-cs.pdf], 2 August 2006.

[Oasb] Oasis. *UDDI - Universal Description, Discovery and Integration of Web Services*. [http://www.uddi.org/specification.html].

[OASc] OASIS. *Web Services Business Process Execution Language Version 2.0, Working Draft*. [http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.pdf].

[OASd] OASIS. *Web Services Relibility*. [http://docs.oasis-open.org/wsrm/ws-reliability/v1.1/wsrm-ws_reliability-1.1-spec-os.pdf].

[OASe] OASIS. *Web Services Security (WS-Security).* [http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf].

[OMG] OMG, Object Management Group. *Corba 3.0.* http://www.omg.org/technology/documents/formal/corba_2.htm.

[Ope] Open source project. *JOLIE: Java Orchestration Language Intepreter Engine.* [http://sourceforge.net/projects/jolie].

[ora] Oracle BPEL process manager. [http://www.oracle.com/technology/products/ias/bpel/index.html].

[Plo81] G. Plotkin. A structural approach to operational semantics. *Tech. Rep. DAIMI FN-19*, Aarhus University (Denamrk), 1981.

[San93] Davide Sangiorgi. From pi-calculus to higher-order pi-calculus - and back. In *TAPSOFT'93: Theory and Practice of Software Development, International Joint Conference CAAP/FASE, Orsay, France, April 13-17, 1993, Proceedings*, volume 668 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 1993.

[Suna] Sun Microsystems. *Java Remote Method Invocation.* http://java.sun.com/products/jdk/rmi/.

[Sunb] Sun microsystems. *Java Web Services Developer Pack.* [http://java.sun.com/webservices/downloads/webservicespack.html].

[Tha] S. Thatte. XLANG: Web Services for Business Process Design. [http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm], Microsoft Corporation, 2001.

[Vir04] M. Viroli. Towards a Formal Foundation to Orchestration Languages. In M. Bravetti and G. Zavattaro, editors, *Proc. of 1st International Workshop on Web Services and Formal Methods (WS-FM 2004)*, volume 105 of *ENTCS*. Elsevier, 2004.

[W3Ca] W3C. *Extensible Markup Language (XML)*. [http://www.w3.org/XML/].

[W3Cb] W3C. *Web Services Activity*. http://www.w3.org/2002/ws/.

[W3Cc] W3C. Web Services Architecture. [http://www.w3.org/TR/2004/NOTE-ws-arch-20040211], 11 February 2004.

[W3Cd] W3C member submission 10 august, 2004. *Web Services Addressing*. [http://www.w3.org/submission/ws-addressing/].

[WCG+06] Martin Wirsing, Allan Clark, Stephen Gilmore, Matthias Holzl, Alexander Knapp, Nora Koch, and Andreas Shoroeder. Semantic-based development of service-oriented systems. In *Proc. of Formal Techniques for Networked and Distributed Systems (FORTE 2006), Paris, France, September 26-39, 2006*, volume 4229 of *LNCS*, pages 24–45, 2006.

[web] IBM Websphere. [http://www-306.ibm.com/software/websphere/].

[Wora] World Wide Web Consortium. *SOAP Version 1.2 Part 1: Messaging Framework*. [http://www.w3.org/TR/soap12-part1/].

[Worb] World Wide Web Consortium. *Web Services Choreography Description Language: Primer. W3C Working draft 19 June 2006*. [http://www.w3.org/TR/2006/WD-ws-cdl-10-primer-20060619/].

[Worc] World Wide Web Consortium. *Web Services Choreography Description Language Version 1.0. Working draft 17 December 2004*. [http://www.w3.org/TR/2005/CR-ws-cdl-10-20051109/].

[Word] World Wide Web Consortium. *Web Services Description Language (WSDL) 1.1*. [http://www.w3.org/TR/wsdl].