IMT - Institutions, Markets, Technologies Computer Science and Engineering
Ph.D. Thesis

Designing and Experimenting Coordination Primitives for Service Oriented Computing

Daniele Strollo

Ph.D. Thesis
March 13, 2009

Supervisor
Prof. Gianluigi Ferrari

Co-Supervisor
Emilio Tuosto

Via San Micheletto 3, 55100 Lucca, Italy.
Tel: +39 0583 4339561, Fax: +39 0583 4339564
Email: daniele.strollo@imtlucca.it. URL: http://www.di.unipi.it/~strollo.

... al mio cuscino che spero si ricordi ancora di me ...
(... to my pillow, with the hope that it still remembers me ...)

Abstract

Service Oriented Architecture (SOA) and Web Services (WS) are becoming a widely accepted device for designing and implementing distributed systems.

SOAs have given an important contribution to software engineering providing a model where applications are defined by assembling together certain functionalities, called services, possibly provided by remote suppliers.

The characterizing issue of SOAs consists of defining common principles to make services accessible and usable regardless their execution context. Nevertheless, the architectural specification is far from giving a complete reference application model on which systems should rely on. The specification just includes principles for achieving interoperability and reusability of services; other aspects are left to the implementing platforms.

As a consequence, it is understood how services are specified in isolation and how their functionalities are made available to the requesters, but the definition of languages for describing service composition are far from being widely accepted and reveals to be an impelling challenge.

In the last years, several solutions have been proposed for describing aggregated services. However, they often lack a formally defined semantics. Moreover, these solutions are often specific for a platform (e.g. WSs) and are difficult to adapt to other platforms since they rely on low level assumptions that are out of the SOA specifications.

This thesis aims at providing new methodologies for implementing the coordination of services. Our framework proposes to be flexible enough to support high level languages and to provide reliable tools for testing correctness of implementation.

Our approach relies on a formal model that takes the form of a process calculus specifically designed to deal with services and their coordination.

The process calculus has been the main tool driving the specification issues as well the implementation issues.

Indeed, it acts as a bridge between the high level specification language and the run-time environment.

A distinguished feature of our proposal is that our formal model, i.e. the process calculus, describes distributed processes relying on an event notification mechanism as machinery for interactions. Services are represented by certain components that embody local computations and react to changes of the overall environment in which they are involved. The adoption of event notification results particularly fashionable for tackling service coordination. The principles studied at specification level are from one side understood within a theoretical framework that provides instruments for checking correctness of interaction policies and from the other side offers the core model for implementing and experimenting a programming middleware.

Table of Contents

List of Figures v

1 Introduction 1
- 1.1 Service Oriented Computing 2
- 1.2 Service Oriented Architectures 3
- 1.3 Beyond Message Based Coordination 4
- 1.4 Contributions of the Thesis 5
- 1.5 Structure of the thesis 7
- 1.6 Origin of Chapters 8
- 1.7 Acknowledgments 8

2 Preliminaries 9
- 2.1 Service Oriented Architectures 9
- 2.2 Web Services as instance of SOAs 13
- 2.3 Event Notification 16
  - 2.3.1 A walk through events 17
  - 2.3.2 Some Considerations 20
- 2.4 π-Calculus 24
- 2.5 Long Running Transactions 28
  - 2.5.1 Business Process Modeling Notation 30
  - 2.5.2 Naïve Sagas 32

3 Signal Calculus 37
- 3.1 Introduction 38
- 3.2 Alarm system: a running example 38
  - 3.2.1 Modeling the alarm controller in π-calculus 40
    - 3.2.1.1 Alarm system revisited 42

3.2.1.2 A few remarks: π-calculus and Event Notification . . . 44
3.3 Signal Calculus . . . 45
3.4 A few remarks: basic SC and CCS . . . 52
3.5 Managing sessions in Signal Calculus . . . 53
3.5.1 Some useful patterns in SC . . . 57
3.5.1.1 Joining events . . . 58
3.5.1.2 Rendez-vous in SC . . . 59
3.5.1.3 Flow removal and reaction hiding . . . 60
3.5.2 Modeling the alarm controller in SC . . . 61
3.6 Concluding remarks . . . 64

4 Java Signal Core Layer **67**
4.1 Introduction . . . 68
4.2 Architecture . . . 69
4.2.1 Inter Object Communication Layer . . . 70
4.2.2 Signal Based Layer . . . 75
4.2.2.1 Message Delivering Protocol . . . 77
4.3 A running example: the alarm system . . . 78
4.4 Illustrating the network flexibility . . . 80
4.4.1 Gateways . . . 81
4.4.2 Implementation Overview . . . 82
4.4.3 X-Mixed-Replace SOAP Binding . . . 83
4.4.4 JSCL implementation outline . . . 84
4.5 Additional programming facilities . . . 86
4.5.1 Logical Ports . . . 86
4.5.1.1 A sketch of logical ports . . . 87
4.5.1.2 Logical ports: API . . . 87
4.5.2 Guarded Flows . . . 91
4.5.3 The dark side of serializers . . . 92
4.6 Concluding remarks on JSCL . . . 93

5 Programming Environment **95**
5.1 Event based Service Coordination . . . 95
5.1.1 JSCL Graphical Notation . . . 96
5.1.2 Signal Core Language . . . 97
5.1.3 Basic Facilities . . . 99
5.1.4 Synchronizing behaviors . . . 102
5.1.5 Logical Ports in SCL . . . 104
5.2 A case study . . . 105

5.2.1 The car repair scenario 105
5.2.2 Designing the Car Repair Scenario 106

6 Experimenting Long Running Transactions 109
6.1 From BPMN to SC (informally) 109
6.1.1 Compensable activity 110
6.1.2 Sequence 111
6.1.3 Parallel composition 112
6.1.4 Transactional enclosure 113
6.2 BPMN to SCL Model transformation 114
6.2.1 BPMN atomic process 115
6.2.2 Parallel composition 116
6.2.3 BPMN sub-transaction 117
6.3 Scl Model refactoring 119
6.4 Concluding remarks 120
Bibliography 122

List of Figures

2.1 SOA: actors and interactions. 11
2.2 Event notification: core elements 20
2.3 Event notification overview 21
2.4 Event notification: brokered 22
2.5 $BPMN_{tr}$: composition of compensable activities 31
2.6 $BPMN_{tr}$: transactional boundaries 31

3.1 Alarm system automaton 39
3.2 Abstract graphical notation of join 58
3.3 Alarm system in SC: overall system 62
3.4 Alarm system in SC: the door sub network 63

4.1 JSCL Architecture 70
4.2 IOCL metamodel: «iocl_core» profile 72
4.3 IOCL metamodel: «iocl_comm» profile 74
4.4 Signal Based Layer metamodel: «`sbl`_data» profile 75
4.5 Signal Based Layer metamodel: «`sbl`_core» profile 76
4.6 Signal delivery protocol 78
4.7 Registration and signal emission protocol 84
4.8 Joining two components 86
4.9 Profile «sbl_ext»: logical ports 88
4.10 Search and buy books overview 89
4.11 Profile «sbl_ext»: guarded flows 91

5.1 JSCL diagram of the example 97
5.2 ESC design: the join pattern 100
5.3 Synchronizing reactions installation 104
5.4 Car repair scenario: the BPMN model 106

5.5 Car repair scenario the graphical JSCL representation 107

6.1 Internal view of SC compensable activities 110
6.2 SC sequential composition 112
6.3 SC parallel composition 113
6.4 SC transactional enclosure 113
6.5 The generated SCL network 114
6.6 SCL refactoring: merging parallel collector 119

List of Codes

3.1 Event subscription in π-calculus 40
3.2 Alarm system in π-calculus 41
3.3 Alarm system revisited 43
3.4 Alarm System Interfaces: Reactions 63
3.5 Alarm System Interfaces: Flows 64
3.6 Alarm System: Network 64

4.1 Alarm system in JSCL: local names 79
4.2 Alarm system in JSCL: adding flows 79
4.3 JSCL alarm coding: adding a reaction 80
4.4 Logical Ports: AND behavior 87
4.5 Search & Buy Books: OR port 90

5.1 Synchronizing behaviors in SCL: queue 102
5.2 Synchronizing behaviors in SCL: sync 103
5.3 Synchronizing behaviors in SCL: consumer 103
5.4 Binary AND port in SCL 104

6.1 SC coding of BPMN$_{tr}$: compensable activity 111
6.2 BPMN to SCL transformation: compensable activity 115
6.3 BPMN to SCL transformation: parallel dispatcher 116
6.4 BPMN to SCL transformation: parallel collector 117
6.5 BPMN to SCL transformation: saga dispatcher 118
6.6 BPMN to SCL transformation: saga not 118
6.7 BPMN to SCL transformation: saga collector 119

Chapter 1

Introduction

The enormous expansion of the Internet has recently brought to a new vision of the development of distributed software. In fact, modern digital networks yield great possibilities for individuals and institutions to cooperate through sophisticated communication mechanisms. Arguably, the evolution of communication networks and the middleware for their programming let modern distributed applications to be envisaged as the composition of units of computations. Such units are often called services as they are supposed to be available, invokable, and possibly substitutable without affecting invokers. To give a metaphor, software services can be very much thought of as ’real’ services; namely, they can be discovered (by looking on the yellow pages), selected (according to some criteria) and used as long as they suit the user needs. In particular, on different occasions different services may be invoked without this requiring to change the application.

In a sense, services are computational units no longer relegated on the specific machines, or in restricted domains, but replicated and distributed so to be accessible to invokers and provide an open and dynamic computational environment. Services consist of interactive building blocks which, once published, become reusable by other components making applications more adaptable and flexible as they can more easily adapt to local and global changes. As [ibma, All98] puts it, the component based architecture is extended with the notion of service. More precisely, a component can be characterized by the following features:

> *“ it is an executable unit of code that provides physical black-box encapsulation of related services; its services can only be accessed through a consistent, published interface that includes an interaction*

standard; it must be capable of being connected to other components (through a communications interface) to form a larger group. ”

Albeit attractive, the vision of distributed applications as consisting of interactive and loosely coupled services, yields several complexities at many different levels. Remarkably, it is necessary to radically change the way applications are designed so to focus, not only on the development of the so-called *business logic* (namely, the domain-specific functionalities), but also consider the *distributed coordination* issues. This is in fact a rather important research topic that has recently been tackled by both academia and industry with the introduction of the so-called *service oriented computing* paradigm.

1.1 Service Oriented Computing

The basic idea of *service oriented computing* (SOC) is to exploit the possibility of building applications by "gluing" together element of computation called *services*. The ultimate principles of this paradigm are that:

- services can published with an interface that is amenable to be searched, bound, and invoked by other services;
- applications can dynamically replace the services they use with other services.

Typically, it is also assumed that services are executed on heterogeneous host machines and no assumptions can be taken on their running platforms.

The main challenge of SOC consists in the ability to permit services to be accessed in a uniform and platform independent manner, adopting communication mechanisms suitable for coordination of distributed services.

One of the key features of SOC is therefore to make it possible the coexistence of services offered by different providers, possibly deployed on heterogeneous platforms.

Despite the potential offered by SOC, it is still in its infancy. Many tools related to SOC have been released in the last decade, but their level of maturity is still far from the maturity reached by their counterparts in conventional middleware for distributed systems. In addition, existing tools lack of a full-fledged foundational basis.

Also, rigorous mechanisms for automatically designing and controlling their work-flows are indeed required. As a matter of fact, it is still a challenging issue consists to define mechanisms for describing how these components can be aggregated. Finally, SOC lacks standards for defining behavioral properties (e.g. transactional requirements) related to services.

1.2 Service Oriented Architectures

In order to make the SOC paradigm concretely realizable, *Service Oriented Architectures* (SOAs) have been proposed [CKM+03, AACP04, Pap03]. Basically, SOAs yield the middleware that allow services to comply with a common strategy so that they can interact in a reliable way abstracting from low level details.

The initial attempts of designing rudimentary SOAs can be traced back to DCOM [Mica], Corba [Tea] and Java Remote Method Invocation (RMI) [SUNb], to cite a few. Unfortunately all these approaches fall short to meet all the requirements of SOAs for SOC. These solutions rely on run-time environments strictly dependant on the operating system (it is the case of DCOM) or on the network infrastructure adopted (e.g. Corba) or on the host language (as for Java RMI). Comparative evaluations among these approaches can be found in [TW97, PS98].

These first attempts have given impulse to the raising of a new strategy for implementing SOAs, the Web Services (WS) [W3Cb].

The Web Service core specifications provide mechanisms for describing, publishing, retrieving and accessing services. These specifications characterize the wire level *of the SOA architectural model.*

WSs better adhere to the SOC paradigm since they do not impose constraints on the run-time of services. Moreover, the communications of WSs are implemented by exchanging messages declared in an open format like XML (eXtensible Markup Language) [BPS97]. The adoption of XML has opened a new perspective for developers and service providers enabling language and platform independence (a.k.a. *interoperability*).

While the wire level has reached an appreciable level of maturity, methodologies for describing service work-flows and service aggregations are still at an early stage. Moreover, the extensional features for coordinating service activities are strictly tailored to WS platform and introduce additional constructs to the basic structure of messages to give control on their flows so they cannot be considered a fully general solution for SOC.

1.3 Beyond Message Based Coordination

The majority of the proposals for SOAs, including WSs, adopt interaction mechanisms based on message exchange. Namely, the coordination of services is modeled by describing the flow of messages that should exchanged among parties. The intuition behind this choice is that the network infrastructure is directly reflected in the architectural model of languages for service aggregation so that messages represent the core constructs for implementing coordination of services.

Alternatively, coordination of services can be regulated by specifying how they *react* to the evolution of their execution environment. The system undergoes modifications that are represented by suitable events that are promptly notified to the interested partners. Components passively observe changes that are applied to the environment and trigger handling routines at their occurrences. Moreover, they actively concur to modify the system state. Basically, distributed pieces of functionalities access a common virtual global state that acts as a bridge for the involved parties hiding to them the underlying network structure. This paradigm is commonly referred to as *event notification* (EN).

The adoption of event notification yields to model services in terms of reactive entities that, autonomously, declare the set of events they are interested in and the behavior that they perform upon their occurrence. The coordination of services is obtained by regulating the flows of events issued during the computation.

The EN paradigm allows programmers/designers to focus on how each computational entity behaves upon occurrences of environmental stimuli instead of considering complex interactions among several agents.

In the first instance, components are designed in “isolation”, focusing on how:

- components act upon occurrence of an event of interest and
- notifications of events issued by components take place.

Once deployed, such components are “injected” in a particular network and suitably “linked” together by exploiting the subscription mechanism. This two-phase design relaxes the inter-dependencies among components achieving an high degree of loosely coupling and, consequently, reveals to be particularly suitable for SOAs.

1.4 Contributions of the Thesis

Although the EN paradigm is receiving the attention of the scientific community investigating distributed systems, its potential to model SOC is still not fully exploited. Also, the existing middleware based on EN mechanisms are typically built without a precise semantics.

This thesis promotes a framework based on EN mechanisms as a model for SOC and as the basis for a middleware suitable for SOAs. The main contributions of this dissertation can be summarized as follows:

- formalization of an EN programming model in terms of a process calculus called *Signal Calculus* (SC);
- development of the JSCL middleware (after *Java Signal Core Layer*) based on SC primitives;
- advanced features of JSCL not present in SC;
- definition of a programming framework for JSCL to design and program coordination of distributed services;
- application of the SC/JSCL framework to design, program, and reasoning about *Long Running Transactions*.

Our approach relies on a process calculus called *Signal Calculus* (SC) featuring on event notification as basic interaction mechanism. The *Signal Calculus* (SC) is a process calculus enriched with a concept of component locality and suitable primitives (such as multi-cast mechanisms) for dealing with event notification. A key feature of SC consists of multi-cast asynchronous communications and yielding an high degree of loosely coupled components.

We shall illustrate the effectiveness of event notification approach embedded in SC to tackle service coordination issues. This thesis shows how the SC/JSCL programming abstractions can be exploited to facilitate the design and the development of SOA applications. Moreover, we shall compare the programming model of SC with the one of other formalisms outlining how, some of the native primitives of EN require more efforts (e.g. the reconfiguration of the processes) and cumbersome coding when not featured directly.

One of the key aspects of JSCL is that it has been designed and implemented by a two-level architecture in order to achieve interoperability and modularity. The lower architectural layer is the *Inter Object Communication Layer* (`iocl`) consisting of several instances called *network adapters* that abstract the actual networking level. The `iocl` hides the network complexity to the higher layer called *Signal Based Layer* (`sbl`). The basic idea is that each `iocl` network adapter acts as a bridge among several network infrastructures supporting distribution of services and the notification/delivery of messages to the distributed components. The `sbl` layer provides all the facilities to declare services that interact in an event notification style. Additionally, `sbl` declares a set of high level mechanisms tailored to define complex coordination patterns with the aim to reduce the development efforts.

We shall discuss how the network model of SC is reflected into the JSCL so to provide the run-time support for the coordination of distributed SC components. As said, an important aspect of our approach is that the design and the implementation of the JSCL middleware are grounded on solid formal basis. Indeed, JSCL design and the related development methodology are characterized by a close interplay between formal semantics modeling, implementation pragmatics and application.

As a matter of fact, all the notions of SC are paralleled in the JSCL API. Hence, SC and JSCL can be regarded as a foundational framework and its programming counterpart for specifying, verifying and programming coordination policies of distributed services.

It is important to remark, however, that JSCL has been extended with features that are not present in SC. In particular, JSCL features, among others, more sophisticated mechanisms for subscribing and notifying events than SC. This makes JSCL rather expressive and flexible as illustrated by the encoding in JSCL of transactional behaviors. In fact, the JSCL middleware architecture results flexible enough to adequately implement *Long Running Transactions* on top of JSCL. For instance, the JSCL programming environment introduced in this dissertation allows designers to easily map BPMN designs into a graphical notation for JSCL programs (that has been implemented as an Eclipse plug-in). Besides being a handy tool for practitioners, this enables to effectively apply some results on refactoring of long running transactions that have been defined on top of SC. Remarkably, this approach hides the underlying theory from designers and programmers.

We envisage the impact of our approach on the service oriented computing technologies as follows. Conceptually, the SC/JSCL framework adds a further layer to the basic SOC, like web services protocol stack (SOAP, UDDI, WSDL). The SC/JSCL layer provides the formal and programming mechanisms to design, verify and program web service coordination policies (e.g. a BPEL orchestrator or a WS-CDL choreography) on top of the basic service protocols.

This thesis provides a complete description of JSCL from its formal modelling (the SC process calculus) to the middleware implementation and applications. The experimental results of the thesis reinforced our idea that a framework for service coordination based on formal ground, such as JSCL, can be an effective software engineering tool for the SOC paradigm.

We shall limit the technical presentation of SC to the minimum required to illustrate the JSCL middleware. For a more comprehensive presentation of the SC framework the reader is referred to [Gua09].

1.5 STRUCTURE OF THE THESIS

The thesis is organized as follows.

Chapter 2 provides a reviews the main concepts and notations that will be used in the thesis. We start by introducing the idea of SOA and its technological background. A survey of event based systems is also given (mainly by suitable examples) in order to gain confidence with aspects that are peculiar in SOA and to give a flavor of many existing variants of this paradigm. We outline the effectiveness of EN to tackle with SOA relevant aspects. We conclude by supplying an overview of the π-calculus and of long running transactions as specified by Naïve Sagas.

In Chapter 3 we introduce the Signal Calculus. We argue the adequacy of the event notification to model a wide range of computational scenarios where systems involve dynamic rearrangements of the surrounding environment. We also motivate why the Signal Calculus provides suitable foundational machineries for SOAs. A comparative evaluation of message style interactions (e.g. the π-calculus-like approaches) with respect to the event notification interactions (as adopted in SC) is given.

Chapter 4 describes the design and implementation of the JSCL middleware acting as run-time support of `sbl` the actual programming counterpart of SC. Moreover, we argue the flexibility of the middleware to adapt to different network overlays and to support actual SOA platforms. Through some examples, we point out the strict interplay between the actual programming primitives and the theoretical constructs. This Chapter ends by reviewing a set of high level facilities introduced to make more flexible the development of JSCL networks via suitable lightweight synchronization facilities.

Chapter 5 presents the Event based Service Coordination (ESC) programming environment that provides the programming supports for designing and implementing JSCL services from a higher level of abstraction. The framework is constituted by different Eclipse plug-ins that offer different levels of design facilities. From the one hand, a graphical toolkit is used to design the components and their interconnections. On the other hand, a textual representation can be used to specify how components are internally implemented by declaring their behavioral properties. We also illustrate the usage of ESC through a case study borrowed by the SENSORIA Project.

Chapter 6 exploits Sagas to give a formal semantics to the BPMN transactional constructs via a semantic-based mapping to SC. We end by outlining how the results presented in [Gua09] on the *refactoring* of SC coordination patterns effectively simplify the development of BPMN transactions. Indeed, the refactoring rules of [Gua09] are proved sound by a correctness result stating that they preserve conformance of the specifications (via weak bisimilarity).

1.6 Origin of Chapters

Many chapters of this thesis are based on already published papers.

- The formal specification of the SC calculus described in Chapter 3 appears in [FGS06b, FGST07].

- The design and the implementation of the JSCL middleware described in Chapter 4 has been originally presented [FGS06b, BFM+05, FGS06a].

- A preliminary version of the ESC environment presented in Chapter 5 appears in [FGST08]. The semantic-based debugger for JSCL has been presented [FGSTa].

- The case study presented in Chapter 6 extends results given in [CFGS08, FGSTb, CFSG08].

Notice that the above list points out the papers where the results of this thesis have been introduced and described for the first time. However, there are some results which are contained in this thesis and have never written before. Most of these are contained in Chapter 3 and in Chapter 4 which provides significant extensions of the published papers. Finally, a presentation of some of the results of this thesis targeted for IT professionals has been published in [BCF+08].

1.7 Acknowledgments

This research is supported by the EU FET-GC2 IST-2004-16004 Integrated Project Sensoria.

Chapter 2

Preliminaries

2.1 Service Oriented Architectures

The use of Internet access as a way of computing opened new prospectives. In the past, applications were usually intended as software solutions residing and executed on the end-user machine. Tangible scenarios of such applications include graphical manipulation, word processing and so on. In the last decade, we assisted to a gradual migration from stand-alone applications, to distributed solutions capable of taking advantage of functionalities exposed on the network. Such applications are able to exploit the network to retrieve the functional capabilities they are interested in and to access data remotely stored in a promiscuous modality involving interactions with both local and remotely available software components.

Initially, the Internet was intended as the vehicle for exposing document centric contents (e.g. HMTL pages, images, etc.) through the Web; web pages and applications were playing distinct roles and the interactions occurring among them were obtained through platform dependent protocols or requiring the explicit interaction with the user. A new challenge consists in making these worlds to coexist. Consider a site offering the possibility to look up contacts from Yellow Pages. Clients can access, through a Web page, some useful information, (e.g. the e-mail contact) and further reuse them in other applications (e.g. a mail client). Instead of demanding the user to perform these tasks, the application (mail client) can be equipped with the facility for automatically retrieving from the site the desired data (mail contact) and performing the remaining task (sending the message). The client application performs a sequence of operations that involve remote functionalities. All the steps involved in issuing the proper service will be hidden to the user, promoting a service obtained by composition of remotely exported functionality. Such a service exhibits adaptability to respond to configuration changes (e.g. the remote service is moved to another address) or to changes of plans (e.g. the support of additional functionalities).

The idea of providing a standard solution for defining software modules and their compositionality is not new. If we take into account the well known Object Oriented Programming (OOP) paradigm, applications are built up on modular functionalities that can be provided by third parties and assembled together to implement more complex functionalities. At the basis of OOP there is the idea of *code reuse*. Modules are declared in a standard manner in order to be easily integrated. Yet, the integration was allowed under certain constraints as the compiled code was strictly tied to the executing machine. Modern object oriented languages, such as Java and C#, have evolved to be machine independent adopting an intermediate *virtual machine* that interprets instructions defined in a meta-language and acts in behalf of the real machine. The machine abstraction achieves a first level of interoperability. Nevertheless, these languages remain enclosed into local boundaries so that communications with remote components have to be explicitly programmed. The *linking* among modules happens statically and all the pieces of code referred have to be provided on the running machine at compile time.

In order to design applications by composition of software modules made available on the network a new model has been defined.

The Service Oriented Architecture is an architectural style whose goal is to achieve loose coupling among interacting software agents. A service is a unit of work exposed by a service provider to achieve desired functionalities for a service consumer.

Services are functional units exported to the world through a standard and well known “contract”. Once invoked, services *serve* a request coming from the customer and return the results, so that the computation is remotely performed.

We now recall the main characteristics of services.

Service *interfaces* and the exchanged data are defined in a *neutral* manner independent from platforms and programming languages and communication happen over standard Internet protocols. This constraint allows services to interact with each other in an uniform manner and is often referred as *interoperability*.

Services can be dynamically searched and retrieved. For this purpose, so called *registries* are defined. Registries act as Yellow Pages and allow service providers to declare their published services. Subsequently requesters are capable to find and retrieve the services from registries according to the interface description (or to other constraints). In this way the client can access a published service at run-time. This solution offers a strong separation among providers and requesters, since no one needs to be conscious of the counterpart. Besides, services can be directly accessed by customers avoiding the interaction with registries. The usage of registries increases the dynamicity and the adaptability of services. In fact, the requester just requires to the registry to find a service able to provide a given functionality. It can happen that several services match the requirements and they can be interchangeably replaced with no acknowledgment of the requester (*adaptability*). For instance, the registry can balance the machine loading by properly scheduling the access to different services matching the same request (e.g. to achieve *scalability*). Remarkably, in all these cases, the linking to services happens in a *dynamic* fashion.

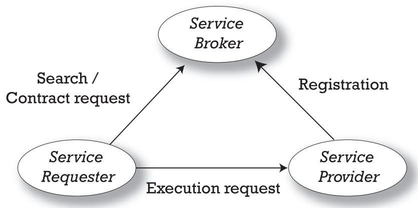


Figure 2.1: SOA: actors and interactions.

A typical interaction pattern for publishing and accessing services is shown in Figure 2.1. The main actors involved in such interaction are:

i) *Service Provider*: the entity which defines a service which will be published in order to be used by a *service requester*. Usually this role is played by an application or by another service.

ii) *Service Broker*: is responsible for maintaining the registrations of service providers and for making these services available to the requesters.

iii) *Service Contract*: defines the structure of the messages which can be exchanged among the *actors*, both requests and responses.

iv) *Service Requester*: corresponds to the client entity which wants to access some functionalities exported by the services. This actor is, however, a software module, meaning that in the SOAs the interactions are modeled always among software agents, not among human and software agents.

Services must be *self-contained* meaning that they have to be independent from the environmental state. Their computations simply depend on the parameters provided by the request and are independent from the internal state. A more restrictive clause suggests to use *stateless* services. Depending on the coordination pattern adopted, this constraints may be relaxed, since often the state is needed e.g. for keeping track of the network topology in which a computation is taking place (it happens for the choreography discussed later).

Services must be *loosely coupled.* The coupling is referred to inter-modules dependencies. Components loosely coupled have a minimal number (possible nothing) of dependencies among them. This constraint is essential, since each service must be isolated from the execution context to be *reusable*.

Depending on the purpose for which a service has been designed, it can expose a single logical functionality or a group of them; the terms *course-grained* and *fine-grained* are used, respectively, to represent these two different situations. Coarse-grained services offer a set of related functions rather than a single function. For example, a coarse-grained service might handle the processing of a complete purchase order. By comparison, a fine-grained service might handle only one operation in the purchase order process. A fine-grained approach is preferred when the same functionality have to be reused in several contexts while course-grained one permits to reduce the communications between the service and the requester.

Asynchronous interactions among services are preferred since a long period of times can elapse between the request of the customer and the availability of a response from the service. As a consequence, the interactions with services are usually described in a message passing fashion.

The *transport protocol* adopted between requester and provider is out the specifications of the architecture, and so strictly related to the choices made by the particular implementation adopted. Usually, in order to give more flexibility to the framework, overlay networks are introduced allowing multi-layered/multi-protocol support.

Another important aspect, not directly covered by SOA specifications, is the possibility to *compose* services. Even if externally a service is seen as a single entity, it can be the result of aggregation of several previously defined services and, on its turn can be reused for a new service composition. Since the main focus when developing services must be put on reuse and composition, the proper design of their interface is crucial in order to reach these objectives. The loosely coupling of services is determinant to enable compositionality.

2.2 Web Services as instance of SOAs

Web Services (WSs) [W3Cb] are often confused with SOAs, hence a clarification is needed. Web services constitute an implementing platform for SOAs that uses specific standards and language protocols. Nevertheless, due to the choices made by many vendors, WSs are often equated with SOAs and the term "business processes" is used on behalf of "services" (or "composition of services") and, accordingly, the services implementation referred to as "business logics" (the terminology is detailed in [ibma]).

Differently from other solutions, like CORBA [Tea], Java RMI [SUNb] and Message Oriented Middlewares, such as the Java Message Service (JMS) [SUNa], WSs meet all the SOAs requirements since they define interfaces which can be published in the web and subsequently retrieved by client applications, using as description a W3C standard language, the Web Services Definition Language (WSDL) [CCMW01], based upon the well-known XML (eXtensible Markup Language) [BPS97].

Another advantage of WSs relies on the adoption of an intermediate protocol, the Simple Object Access Protocol (SOAP) [BEK+00], to envelop the messages exchanged among services. SOAP makes the services independent from the transport protocol adopted (e.g. HTTP, SMTP, TCP) and provides an overlay network which hides the low level details (e.g. the serialization and the deserialization of objects exchanged through messages). Interestingly, SOAP features RPC (Remote Procedure Call) and message driven communications. The main advantages of the RPC strategy is the simplicity of use, since it reflects the usual OOP interaction pattern. The alternative solution seems instead to be preferred since it features asynchronous communications, that encourage the loosely coupling interactions. Hence, many companies deprecate RPC communications against message driven ones, being the last one more accordant to SOA specifications. Anyway, the synchronous behavior of procedure call can be avoided by making use of callbacks.

The Web Service platform reflects the key ingredients of SOAs in this manner: the *web service*, the *client*, the *Universal Description Discovery and Integration* (UDDI) [W3C00] registries, and the *WSDL* correspond, respectively, to the concepts of *service provider*, *service requester*, *service broker* and *service contract* hinted in Section 2.1.

If we compare Web Services to other existing technologies for distributed components, we can observe that WSs enjoy many advantages. They are *platform independent*, meaning that components developed on heterogeneous platforms can be involved in the same computation. Another important feature regards the *reuse of existing infrastructures* that permits to take advantage from standard transport protocols such as HTTP or SMTP providing more flexibility and cost reduction. Finally, Web Services by design permit to define *loosely coupled* components.

The characteristics of WSs recalled up till now, define the *wire level* of the platform, in the means of the lower architectural layer needed to develop and interact with services.

Moreover, the WS platform comes equipped with a large set of specifications, generally referred to as WS-* that are devoted to support transactional capabilities (WS-Transaction [IBM05]), and to deal with security (WS-Security [OAS06]) and reliability (WS-Reliability [OAS04]) of exchanged messages.

Regarding the specifications for implementing coordination of services, hereafter, we will review the main branches of coordination policies (*orchestration* and *choreography*) and the two main industry standard solutions proposed for approaching them, BPEL and WS-CDL.

Service Coordination

Regarding the *coordination* of services, two different approaches can be adopted: *orchestration* and *choreography*. In the orchestration, services are thought as isolated and the main focus relies on their internal behavior. The participants are not aware of the surrounding network. An intermediate component, the *orchestrator* is responsible to arrange service activities according to a planned work-flow. This strategy provides a *local view* of the participants. A *choreography* instead specifies how services should be connected and how they should interact so that each service accomplishes its task within the given choreography. Roughly, choreographies yield an abstract global view of SOA systems to be “projected” on the distributed components.

Several work-flow languages have been proposed in order to achieve services *compositionality* (a few of them are listed in [IBMb]). Among them, the Business Process Execution Language for Web Services (WS-BPEL or BPEL for short) [AGK+03, Spe] and the Web Service Choreography Description Language (WS-CDL) [W3Cc].

BPEL4WS

BPEL has been obtained by combining the Web Services Flow Language

(WSFL) [IBMc] and the XLANG [Micb] specifications. Quoting the BPEL specifications [Spe]:

> “ *WS-BPEL is an XML based language enabling users to describe business process activities as Web services and define how they can be connected to accomplish specific tasks. WS-BPEL is designed to specify business processes that are both composed of, and exposed as, Web Services. WS-BPEL is an orchestration language.* ”

In BPEL, orchestration relies on the WSDL description of involved services to build a *business process*, in the means of a service based application, that involves access to the functionalities exposed by such services. The set of operations (called *ports*) and the set of data types needed to interact with them are retrieved from the WSDL specification. On top of this information, the BPEL designs are built by specifying how the data have to be exchanged among partners and the handling functions to activate in case of their failures. The **invoke** and **reply** actions declare the role that two services are playing in a single activity (resp. the service requester and the service provider). Additionally, activities can be composed in sequence or in parallel (resp. **sequence** and **flow**), or can **throw** and exception. The **while** construct permits to repeatedly execute an activity and the **switch** construct permits to apply a conditional logics on flows. Variables are explicitly managed in BPEL. The **assign** primitive allows to update the value of a variable, that are accessed by services through the **receive** primitive.

However, the BPEL description would not specify how a given service should process a given task internally. A BPEL program describes an interaction protocol that expresses what pieces of information an activity consists of, and what exceptions may have to be handled. Only the WSs composition is described at this layer. It is assumed that BPEL will be combined with other languages which are used to implement internal functionalities (also referred to as *business logics*). The execution of BPEL orchestrations is performed by BPEL *engines* that interpret and execute the BPEL processes. The engine is responsible, at each step of the orchestration, to make the proper decision and to engage the required tasks on the services.

In order to give a reference guide for comparing the effectiveness and the expressiveness of work-flow languages, a set of relevant work-flow patterns have been isolated and proposed in [vdAtHKB00]. These patterns have been used in [WvdADtH03b] for analyzing the BPEL constructs and in [WvdADtH03a], for making a comparison of BPEL with existing alternatives, such as WSCI [W3Ca], BPML [OMG02], BPSS [UO01] and XPDL [WfM02].

WS-CDL

WS-CDL is a choreography language that describes the messages exchanging among services that participate in collaborative environment. As defined in [W3Cc]:

> “ *The WS-CDL is an XML-based language that describes peer-to-peer collaborations of parties by defining, from a global viewpoint, their common and complementary observable behavior; where ordered message exchanges result in accomplishing a common business goal.* ”

The WS-CDL does not supply an executable process of the collaborative definition, since there is no center of control as the participants are peers. While BPEL defines the coordination from the point of view of a participant, here the focus is on a global public view. In few words, the choreography designs are specified as sequences of interaction that involve all (or a part of) the coordinated services. At this level it is possible to express coordination patterns in this manner: “after service A and B communicate on channel ch_x, the services C and D communicate on channel ch_y”. Additionally, the data exchanged among participants can be bound to variables that are used during conversations.

The WS-CDL language is considered more abstract than BPEL and not directly executable. Nevertheless, a given choreography can be *projected* to orchestration views in the means of single services participating to the overall coordinated activity. This way, to each service it is provided the local view of the business process that can be executed through BPEL engines as discussed in [MH05].

2.3 Event Notification

Service oriented applications can be conveniently thought of as being *event driven*, namely, they perform their work in response to events issued externally by means of some notification mechanism. Accordingly, services are programmed as *reactive* components.

The *event notification* paradigm embodies a possible implementation for event based systems and is often adopted in presence of systems that integrate heterogeneous components and thus require *loosely coupled* interactions. The event notification pattern turns out to be suitable for coordinating distributed activities; the typical situation is when a single event occurrence should be notified to many agents.

The event based solutions mainly share common functionalities and, in the example presented in Section 2.3.1, we give an intuition of how these systems work and the basic concepts they rely on. Further practical considerations on the possible variants of event based strategies are reported in Section 2.3.2.

2.3.1 A walk through events

In the every day life, we are surrounded by systems that react to events remotely issued and also human behavior is often influenced by the occurrences of events. We are familiar with event based systems that, more than having a common clear meaning, expose an high flexibility to adapt to several contexts. In particular we discuss how the events can be useful to implement the coordination of services.

In this section we recall the main characteristics of the event based paradigms through the following running example.

Example 2.3.1 *We model a traffic light as an event-notification system. Specifically, when the light becomes red, drivers stop their cars and wait for the green light. Two kinds of events are possible. For the sake of simplicity we omit the "yellow light" situation since not relevant.*

From the example we isolate the main ingredients and the main features of event notification.

The *ingredients* are events, notifications, publishers, subscribers, subscription and topics. We now detail each of them, and after, in Chapter 3, we use them as pillars of our programming model.

i) abstractly an ***event*** represents the viariation of a state. An event occurrence can regard a change of the *internal state* of an agent or could be *environmental*. Depending on the abstraction level, a state variation may be the assignment to a variable, a clock tick, or a mouse click, etc.

   The relevant events of Example 2.3.1 are those that involve changes to the traffic lights, for example "green" or "red" lights. The events are generated by an internal timer that associates a slot of time to each color. When the assigned time expires a new event occurs.

ii) a wide range of events can occur, so it must be possible to classify them offering the possibility to declare the class of events an agent is interested in. In order to characterize domains of events, classes of homogeneous events are grouped into ***topics***.

   The drivers of Example 2.3.1 declare to be interested to watch the traffic lights and will ignore other events that eventually will occur (e.g. "it starts to rain"). As consequence, the relevant topics for the drivers will be "red light" and "green light".

iii) ***notifications*** are the machinery for interactions, namely they consist of messages carrying the description of the event to be delivered. The notification process relies on topics to appropriately deliver events. In fact, topics differ from usual data types as they drive notification of events.

In Example 2.3.1, once the semaphore timer expires, the related event is notified (to the drivers) by changing the lights.

iv) *two typologies of actors* are possible. On one hand, we have the ***publishers*** that "actively" participate to the interactions by raising and notifying events. On the other hand, there are the ***subscribers*** that declare their interest to receive notifications for events and "passively" wait their happening. Upon a notification reception will correspond an action on the subscriber for triggering its handling. In other terms, such agents *react* at the happening of an event.

In the Example 2.3.1 the traffic lights act as publishers and, as counterpart, the drivers act as subscribers. When the red light is notified, the driver reacts by stopping the car.

Notice that a subscriber can act as publisher (and vice versa) with the respect to other participants or to other events, so that there is no clear separation of the two roles.

An intelligent semaphore may serve car once approaching a crossroad and react accordingly. Here the roles are clearly inverted; the semaphore acts as subscriber.

v) the act of declaring the interest to receive notifications for a class of events is called ***subscription***. Two kinds of subscription *policies* are possible. According to the system we are considering, subscriptions can happen *anonymously*, in the means of it is not discriminating the notifier but just the topic (e.g. "if it starts to rain you open the umbrella") or *non anonymously*, if the notifier is relevant.

In Example 2.3.1, the driver is responsible to "subscribe" to the events related to the traffic lights, or rather he observes the light changes.
Moreover, the driver decides to observe just the traffic lights for its direction, so that the events coming from other traffic lights are not captured; the driver applies a non anonymous subscription to a precise semaphore.

The event based systems are characterized by the following *features*:

i) the notification delivering usually happens in a ***multi-cast*** fashion, the notification coming from a publisher is delivered to all the subscribed agents.

In Example 2.3.1, when the lights become green the drivers are informed altogether.

This is the distinguishing feature of event based systems with the respect to message passing or method invocation paradigms. By exploiting the subscription mechanism and the multi-cast notification delivering, the publishers are independent from the subscribed agents and no explicit knowledge about them is required.

ii) publishers are ***isolated*** from the subscribers.

Referring to the Example 2.3.1, the traffic lights behave always in the same manner, regardless the drivers that are involved. The traffic lights have been thought as isolated agents and their behavior defined independently from any particular running situation. The drivers will be responsible to catch the events notified by the traffic lights and to act in correspondence to their happening.

Apparently, while for publishers it is possible to guarantee the isolation, subscribers seem to have acknowledge of the existing publishers to which they subscribe. Other solutions are possible. The dynamic subscription mechanism permits to relax the dependences of subscribers from the publishers. Additionally, some event based implementations give the possibility to decouple the responsibility of implementing subscriptions from the one of reacting to events as clarified in Section 2.3.2.

iii) entities interested in receiving informations can be dynamically registered with services able to produce the information. The ***dynamic*** nature of ***registrations*** implies that both the publishers and the subscribers need no knowledge about the presence of the counterpart.

Considering the Example 2.3.1, the driver decides to make a subscription to a semaphore just when approaches it. This kind of subscription has not been pre-planned and so takes place dynamically.

iv) the notification of an event and its further handling activation usually happen ***asynchronously***.

Suppose the driver, in Example 2.3.1, busy in a call conversation. The light becomes green, he will hang down the conversation before leaving. The driver agent is performing some other task and the event has been scheduled.

The asynchrony is a peculiar requirement of SOA, so the possibility to enable this feature is particularly important.

v) the event notification encourages ***one way*** communications. In analogy to the method invocation, the notifications correspond to *single call* strategy, namely the requester receives no response from the invoked method.

The traffic lights of Example 2.3.1 notify the events to the drivers (if any) and does not care if they will eventually receive the corresponding notifications.

2.3.2 Some Considerations

The event based systems rely on interactions happening in a “cause-effect” way and introduce few primitives for implementing them: subscription and notification.

The event based style of interaction mainly differs from usual method invocation and message driven formalisms, for the adoption of the subscription mechanism that features an high degree of *loose coupling*, resulting so, particularly adequate for SOA. This is a direct consequence of the isolation and the dynamic subscription features discussed in Section 2.3.1. Coherently to how suggester in SOA, agents can be discovered and, accordingly, connected at run-time.

Figure 2.2 illustrates the three agents A, B and C acting as *subscribers* and P_1 acting as *publisher*. The publisher P_1 will emit events of topics evt_1 and evt_2; agents A and C will react to events of topics evt_1, while B to events of topic evt_2. The subscriptions are represented by solid lines and, conversely, the notification delivering is represented by dashed lines. The architecture underlying Figure 2.2 is the one adopted in the *observer pattern* (see [GHJV95]), where subscribers are *observers* and, conversely, the (set of events raised by the) publisher is the *observable*.

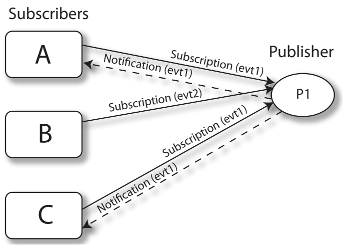


Figure 2.2: Event notification: core elements

Notice that the scenario of Figure 2.2 encompasses the situation described in Example 2.3.1 where subscribers correspond to drivers and each semaphore to a topic. However there are other possible event based solutions that, besides keeping the same ingredients and features given in Section 2.3.1, introduce additional roles or adopt different delivery strategies as explained below. An exhaustive survey of the "many faces" that event based models assume can be found in [EFGK03]. Hereafter, we recall the main characteristics on most important variants of such models.

Decoupling roles Some event based solutions make use of additional components that decouple the roles of publishers and subscribers in relation to the handling of the message delivering and its handling.

A more general specification considers two phases of event subscription (resp. publication). In this optic, a subscriber (resp. publisher) is just responsible to implement the subscription to a publisher (resp. to implement the delivering of notifications to the subscribers). Two further component types are defined: the ***notification producer*** and the ***notification consumer***. Events are raised by producers and subsequently notified through the suitable publishers, and as counterpart they are received by subscribers and their handling demanded to the proper consumers. Alternative solutions that mix the concepts here exposed are possible. In Figure 2.3 we report a configuration in which just the subscribers and consumers are decoupled. On the other side instead, the producer plays also the subscriber role.

In particular, in Figure 2.3, there is a *publisher* that initially declares the topics which can be observed during its execution and two *consumers*, *B* and *C*, that are responsible to handle notifications. The subscriptions are made by a third service, *A*, which acts as *subscriber* for the consumers.

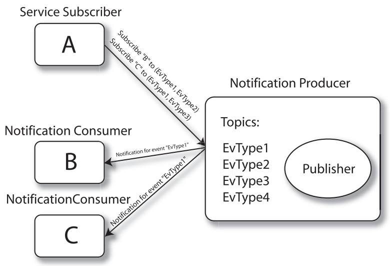

Service Subscriber
A
Subscribe "B" to (EvType1, EvType2)
Subscribe "C" to (EvType1, EvType3)
Notification Producer
Topics:
EvType1
EvType2
EvType3
EvType4
Publisher
Notification Consumer
B
Notification for event "EvType1"
NotificationConsumer
Notification for event "EvType1"
C

Figure 2.3: Event notification overview

Delivery strategies Besides the *non-brokered* solutions described in Figure 2.2 and in Figure 2.3, other delivery strategies are possible. Remarkably, *brokered* solutions can be adopted, as shown in Figure 2.4.

The brokered interaction introduces intermediary component, called ***notification broker***, that decouples consumers from publishers. As depicted in Figure 2.4, the notification broker receives all the notifications from the producers (P_1 and P_2). Any notification is stored by the broker and, depending on the dispatching strategy, dispatched to the consumers (B and C). Observe that, as in Figure 2.3, A acts as subscriber on behalf of the consumers B and C.

In some of event based paradigms, such as the *publish/subscribe*, the broker is mandatory while in the event notification specifications it is considered optional. Since a notification broker is an intermediary agent, it provides *additional capabilities* to the basic producer behavior:

i) it manages directly the subscriptions relieving so a publisher from having to implement message exchanges associated with producer;

ii) it reduces the number of instances and inter-component connections;

iii) it can act as discovery service, so that publishers and subscribers can be retrieved by simply accessing a shared broker;

iv) it permits the use of *anonymous notifications*, so decoupling publishers and consumers from producers;

v) it permits also to manage *persistent notifications* by keeping track of all the events happened into the system; the related notifications will be dispatched asynchronously.

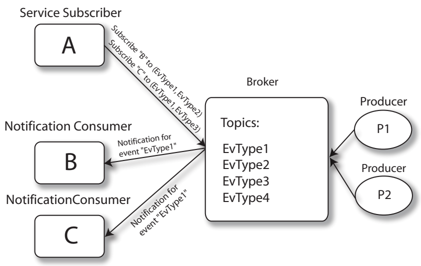

Service Subscriber
A
Subscribe "B" to (EvType1, EvType2)
Subscribe "C" to (EvType1, EvType3)
Broker
Topics:
EvType1
EvType2
EvType3
EvType4
Producer
P1
Producer
P2
Notification Consumer
B
Notification for event "EvType1"
NotificationConsumer
C
Notification for event "EvType1"

Figure 2.4: Event notification: brokered

The brokered solutions seems to be attractive when persistence of notifications is required and features an high degree of loosely coupling among the consumers and the producers of events, with the respect to the non brokered one, since the subscriptions happen anonymously. Yet, it brings to a centralization point that in some situations should result disadvantageous.

Remarkably, delivery strategies constrain communication policies, because of anonymity of the notifications. Specifically, brokered strategies imply the adoption of *broad-cast* communication (notifications are sent to any consumers who are unaware of producers), while non-brokered strategies make use of *multi-cast* communication since notifications are received by the only consumers registered with a given producer.

Typical models based on brokered interactions are “Linda spaces” [CGL86, Gel85]. Several implementations of Linda spaces or, more in general, of tuple spaces are available, among them a Java implementation of Linda [WCC04] and SUN Java Spaces [Mic05], to cite a few. The benefits that Linda spaces offer in presence of reactive systems are discussed in [GZ97].

A further characteristic discriminates the notification dispatching. Two main approaches are possible: *topic-based* and *content-based* mechanisms [CW02, LP03]. The dispatching mechanism in topic-based (also known as *subject-based*) EN systems is simpler than in content-based systems. In topic-based systems, events are categorized into topics which subscribers register to. When an event belonging to a topic τ is emitted, all the components subscribed for τ will eventually react to the event. Notice that publishers and subscribers have to know the topics at hand. In content-based systems, component decoupling is enforced by allowing subscribers to register for events satisfying a given *property*. When an event is emitted the middleware has to dispatch it to *all* the subscribers whose property holds on that event (an example of content-based is SIENA [CRW00]). Notoriously, content-based dispatching mechanisms must be efficient because notification sets, i.e. the set of subscribers that must be notified for the event, can be order of magnitude larger than in topic-based EN [CW03, TAJ03]. A main advantage of content-based EN is that publishers and subscribers do not have to share any *a priori* knowledge about the topics. Subscribers use, instead, a language for expressing properties on events that publishers must simply accomplish with when emitting their events.A more abstract content-based model is the so called *type-based* EN [EG04] where topics are replaced by types (in a suitable type language). Typed events are also used in commercial middleware (see [EG04] and the references therein).

Emerging event based models introduce complex mechanisms of pattern matching on the topics so that the discrimination of the subscription (resp. notification delivering) allows to shift part of the logic at the coordination level.

Although in its basic form, our formalization focuses on topic-based delivering mechanism, some recent results, available in [FGST07], have provided the basis for extending the structure of our model to natively support type-based delivering strategies.

In order to take advantage of the main features of both brokered and non-brokered strategies, we have designed an hybrid solution. Indeed, in Chapter 3 we introduce a specification language for event based systems that adopts a non-brokered delivery strategy (in accordance to the *observer design pattern*) that, on the other side, guarantees persistence of events.

2.4 π-Calculus

The π-calculus primitives are here introduced incrementally recalling, at first, the constructs for modeling interactive concurrent processes. The section ends by introducing some additional aspects that are relevant for comparing our language and the π-calculus. In particular we recall the asynchronous communication and the operators needed for dealing with recursive and non deterministic behaviors.

The π-calculus consists of a suitable language for describing interactive concurrent processes and has been originally proposed in [MPW92]. Further insights can be found in [Mil99, SW02]. The π-calculus models the communication among autonomous and concurrent processes. The emphasis of the language is put on the concept of "named channels", consisting of the set of active media used to implement the process interactions. Besides, names are used to model data conveyed on channels enabling the modeling of value passing.

The basic form of π-calculus is called *monadic*, since it allows only single names to be communicated on channels. A *polyadic* variant enables tuples of names to be communicated and can be encoded into the monadic form as discussed in [Mil92, QW98, Yos96]. An important aspect we will take into account is the possibility to deal with asynchronous communications as presented in [Bou92, HT91]. Asynchrony is crucial for SOA applications since it provides a mechanism for dealing with non blocking message delivering freeing the sender to wait until a receiver declares its availability to consume the message as opposite to the *rendez-vous* behavior.

Names are the essence of the π-calculus and are both used to denote variables and channels. A process performing an output of value (*name*) z through the channel x assumes the form $\overline{x}z$. Its counterpart, the process receiving a datum on the channel x, will assume the form $x(w).P$.

Syntax The syntax and semantics of the π-calculus assume an infinite (countable) set $\mathcal{N}$ of *names*, ranged over by x, y, etc and $\mathcal{P}$ the infinite set of process names ranged by P, Q, R. The *process terms* are built by the grammar in Table 2.1.

P, Q	$::=$	0	empty process
		$P \mid Q$	parallel composition
		$\overline{x}z.P$	output
		$x(w).P$	input
		$(\nu x)P$	name restriction

Table 2.1: π-calculus syntax

Processes, can be the inert process 0, composed in parallel, prefixed by input or output action and declare local (restricted) names.

Example 2.4.1 *Consider $\overline{x}u.P \mid x(v).Q \mid R$ whose first (resp. second) parallel component sends (resp. receives) on x while R runs in parallel with the first two; after the synchronization of the first two components the process becomes $P \mid Q\{u/v\} \mid R$ as will become clear from the semantics described in the following.*

Semantics The *reduction semantics* of the π-calculus is the smallest binary relation $\longrightarrow$ closed under the rules in Table 2.4 and relying on the *structural congruence* relation defined in Table 2.2.

$$P \mid 0 \equiv P \qquad P \mid Q \equiv Q \mid P \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R$$
$$(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P \qquad P \mid (\nu x)Q \equiv (\nu x)(P \mid Q) \quad x \notin fn(P)$$

Table 2.2: Structural congruence relation

The first three rules, in Table 2.2, state that processes form a commutative monoid with respect to the parallel composition. Namely, 0 acts as the neutral element and that the commutative and associative properties hold on the terms. Moreover, the name restriction is commutative. The last rule is the *scope extrusion*: the scope of the name x, restricted in Q, can be extruded to P, providing that x does not occur free in P.

Scope extrusion is central since it describes how a bound name x may be extruded, by an output action, causing the scope of x to be extended.

$fn(0) = \emptyset$	$bn(0) = \emptyset$
$fn(P \mid Q) = fn(P) \cup fn(Q)$	$bn(P \mid Q) = bn(P) \cup bn(Q)$
$fn(\overline{x}z.P) = \{x, z\} \cup fn(P)$	$bn(\overline{x}z.P) = bn(P)$
$fn(x(z).P) = \{x\} \cup (fn(P) \setminus \{z\})$	$bn(x(z).P) = \{z\} \cup bn(P)$
$fn((\nu x)P) = fn(P) \setminus \{x\}$	$bn((\nu x)P) = bn(P) \cup x$

Table 2.3: Free and bound names

In Table 2.3 there are the rules for free and bound names of a process (respectively denoted as fn and bn) defined on the terms of the calculus. Also, the *names* of a process P are defined as $n(P) \overset{\text{def}}{=} fn(P) \cup bn(P)$. The input and the name restriction operators apply binders, in particular, the *input* prefix $x(z)$ binds z and $(\nu x)P$ binds x.

As usual, structural congruence include alpha-equivalence, meaning that two processes are identified if they only differ by a change of bound names.

Example 2.4.2 *A term $(\nu x)P \mid Q$ can be replaced by the term $(\nu w)\{w/x\}P \mid Q$ on the assumption that $w \notin fn(P)$. The application of alpha renaming to the first process implies the replacement of all the occurrences of x in P with the new name w.*

$$\frac{}{\overline{x}z.P \mid x(w).Q \longrightarrow P \mid \{z/w\}Q}\,(COM) \qquad \frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q}\,(PAR)$$

$$\frac{P \longrightarrow P'}{(\nu x)P \longrightarrow (\nu x)P'}\,(RES) \qquad \frac{P \equiv P' \longrightarrow Q \equiv Q'}{P \longrightarrow Q'}\,(STRUCT)$$

Table 2.4: Reduction rules of synchronous π-calculus

With reference to Table 2.4, we just recall the COM rule, since the others have the obvious meaning. In the case of $x(w).Q$, w represents a variable acting as *placeholder* for the value z received on the channel x, and Q the process to execute once the input has been received. Notice that, after the synchronization, the receiver continues as Q where w is replaced with z.

A crucial aspect of the calculus relies on the possibility to generate a new channel name and to communicate it to other processes. In a such way a process can be informed about the presence of a new channel and, later on, use it. An application of scope extrusion obtained by applying the rules in Table 2.4 is reported in Example 2.4.3.

Example 2.4.3 *The term $(\nu y)\overline{x}y.Q \mid x(z).\overline{z}w.P$ represents a process that creates a new name y and sends it to the process concurring with it. The other process receives on the channel x the freshly generated name that is later on used to communicate on it the name w. Having $y \notin fn(x(z).\overline{z}w.P)$, it is possible to extrude the scope of y, so that:*

$$(\nu y)\overline{x}y.Q \mid x(z).\overline{z}w.P \equiv (\nu y)(\overline{x}y.Q \mid x(z).\overline{z}w.P) \longrightarrow (\nu y)(Q \mid \{y/z\}\overline{z}w.P) \equiv (\nu y)(Q \mid \overline{y}w.P)$$

Asynchronous π-calculus As previously stated in Section 2.1, the possibility to enable the treatment of asynchronous interactions is crucial for SOAs. For this reason, our formal approach features asynchrony. Here we recall the basic principles of asynchronous π-calculus (π_α for short) just discussing the syntactic modifications needed to the calculus presented in Table 2.1 to support asynchrony.

In the synchronous form, both the output and the input operations have a *prefixed* form. Although, even if the input operation $x(z).P$, by its nature, is blocking (since it needs to receive, on the channel x, the value for the bound variable z before activating the continuation P), the sender process can avoid blocking by adopting the *bare output*. The output becomes so $\overline{x}z$ with no continuation. The operational semantics remains the same under the assumption that the output is not involved into a non-deterministic choice (described below).

The π_α adopts a relaxed from of output where the sender is unaware of the existence of the counterpart. Roughly speaking, the output operation can be thought of as an insertion of the message in a queue associated to the channel and respectively the input as the extraction of a message pending in the queue. As consequence no warranties about the reception of the message are given. The main advantage that comes from this solution is the decoupling of senders and receivers.

Sum operator The calculus is often enriched with the *choice operator* $P + Q$ through which the language exhibits the possibility to deal with several possible behaviors that can be non-deterministically executed. In addition the *silent prefix*, $\tau.P$ is added to represent a process that, after an internal computation τ, becomes P. To deal with the new operators, the structural congruence of Table 2.2 must be extended with the monoidal rules for the choice operator that are similar to those for the parallel (and therefore omitted).

With the respect to the reduction rules of Table 2.4, some modifications are needed. First, the rule COM is replaced by COM' defined in Table 2.5. In addition, rule TAU is introduced to deal with silent prefix.

$$(x(y).P + ...) \mid (\overline{x}z.Q + ...) \longrightarrow \{z/y\}P \mid Q \quad \text{(COM')}$$

$$\tau.P + Q \longrightarrow P \quad \text{(TAU)}$$

Table 2.5: Reduction rules of asynchronous π-calculus

The term $0 + P$ always evolves in P. Additionally $\tau.P + \tau.Q$ performs an *internal choice* since both the prefixed can be activated. Finally, $xy.P + zw.Q$ is said to perform an *external choice*, since the evolution of the two processes is determined by the context in which they will act. Hence, an output on channel x will involve the activation of the term reading from channel x.

Replication Many π-calculus variants include the *replication* operator $!P$, that allows to write processes that are indefinitely repeated.

Adding replication only requires to extend the structural congruence rules with $!P \equiv P|!P$.

2.5 Long Running Transactions

An important aspect that emerges in current SOAs is to ensure transactional properties related to the execution flow. Note that the problem is not just to coordinate the updates of a distributed repository (e.g., a database), since components are independent and each of them is responsible for maintaining the consistency on local data.

Usually, transactions are thought of as a sequence of actions to be executed *atomically*. Such kind of transactions are also referred as *ACID* (Atomicity, Consistency, Isolation and Durability) transactions, whose virtues and limitations are discussed in [Gra81]. ACID transactions cannot suitably model transactional aspects related to SOAs (for a deeper discussion see e.g. [Lit03]), since often their execution spawns over a relatively long time. Moreover, it is often impossible to guarantee the whole restore of initial state and often these concurrent activities do not permit the isolation of data, meaning that it is not possible to apply locking mechanisms since the resources involved are in general distributed on a large scale in distinct network domains.

Example 2.5.1 *To better clarify the new problematics that arise from the application of transactions in the SOA context consider a site offering the a service for reserving air flights. Under the assumption that credit card payment is adopted, and after a week the customer decides to revoke the previously reserved flight, the usual ACID mechanisms cannot be employed. At, first the locking mechanisms cannot be adopted to ensure isolation of the overall transaction. Usually a compensation is adopted in behalf of the rollback activity. For example the air agency can decide to send an e-mail to the customer with a coupon for using in a further travel.*

An alternative solution consists in using so called Long Running Transactions (LRTs) instead of usual ACID transactions.

In LRTs a relaxed form of atomicity is adopted. Processes are associated with compensations*, that can recover partial executions, in correspondence to failures that happen in the middle of a transactional stack.*

The compensation strategy has been widely accepted by the emerging models for WS coordination. In fact, most of the work-flow languages proposed in recent years, like BPML, WSFL, XLANG and BPEL4WS, include primitives for handling LRTs. Noteworthy, all those proposals lack a formal semantics. The informal specification introduces ambiguities that can lead to different implementations of the same language (as an example, see [WvdADtH03b] and the large list of open issues for BPEL4WS [BPE]). Moreover, those proposals mix together many different concepts and programming constructs. Hence, it is difficult to establish a clear semantics for them because of the mutual interactions of such different constructs.

Alternative solutions for describing LRTs have been proposed in the last years giving rise to several theoretical models, such as StAC [CGV+02, BF04] and Sagas [GMS87]. Comparisons among different LRTs approaches can be found in [BMM05, BBF+05].

Another mainstream in transactional process calculi takes as starting point well-known name passing calculi, like π, and adds to them transactional features like compensable nested contexts [BLZ03], timed transactions [LZ05, MG05], interacting compensable transactions [BMM04] and event scopes [ML04].

Differently from other proposals, like the Business Transaction Protocol (BTP) [OAS02] and Web Services Transaction (WS-TX) [IBM05], that envisage the management of the distributed state by adopting the “two phase commit” discipline [SS83] or similar solutions, the adoption of Sagas better fits our intention, since it allows to adopt a non centralized solution in a choreography fashion requiring no intermediate coordinator.

2.5.1 Business Process Modeling Notation

The Business Process Modeling Notation (BPMN) [Gro02] provides a graphical notation for specifying the work-flow of business processes. The model keeps abstraction from any implementation platform and provides a common set of primitives for describing the activities performed in a coordination pattern, the data exchanged, the events raised during the computations and the connections among the components.

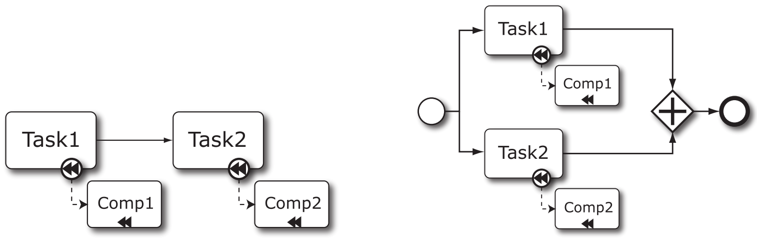
As stated in [Ste04]:

“ *BPMN is also supported with appropriate graphical object properties that will enable the generation of executable BPEL. Thus, BPMN creates a standardized bridge for the gap between the business process design and process implementation.* ”

The BPMN notation provides the same expressiveness of usual work-flow languages. In particular, in [Ste05] the author presents an informal guideline for mapping BPMN designs on top of BPEL processes. BPMN permits to describe the work-flow of a distributed system by a global point of view. The software architect can abstract from the distribution of the processes, the communication mechanisms and the technologies that will implement each process. No explicit notion of location or of software agent is provided at this level. The core elements of computations are the activities that are artifacts of operations. They are not directly bound to an active partner (or to a service). The concept of *swimlanes* (*pool* and *lane*) is introduced to enclose activities inside groups of work units, thus they can be used to identify single participants or to separate the responsibilities of groups of activities. Still, this is just a possible interpretation since no precise constraint (semantics) is given to specify these constructs.

On top of the core BPMN specification (whose quick introduction is given in [Ste04]), a further layer extends the meta-model to treat aspects regarding transactional activities and introduces constructs that result fashionable for treating LRTs.

Here, we focus on the transactional set of BPMN that in the following we indicate as BPMN$_{tr}$. The basic elements of BPMN$_{tr}$ are *compensable activities*, namely pairs of main activities and compensations that can be composed sequentially or in parallel. Figure 2.5 depicts the standard BPMN$_{tr}$ designs respectively for sequential (a) and parallel (b) composition of compensable activities.



(a) Sequence (b) Parallel

Figure 2.5: *BPMN$_{tr}$*: composition of compensable activities

In BPMN$_{tr}$ main activities and their related compensations are represented as boxes linked by dashed arrows. Referring to Figure 2.5, the main activity `Task1` has a "compensation" entry point to which its compensation `Comp1` is attached. The sequential composition is performed by linking together the main activities (cf. Figure 2.5(a)), while the parallel composition makes use of “fork” and “join” operators. In Figure 2.5(b) it is reported a parallel composition of two transactional activities. The two circles represent the start event and the end event of the whole process, while the diamond with the plus operation represents the join of the two parallel activities. The fork operation is implicit in the multiple connections on the start event.

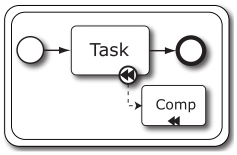


Figure 2.6: *BPMN$_{tr}$*: transactional boundaries

Finally, compensable activities, and their compositions, can be enclosed inside transactional boundaries as shown in Figure 2.6.

All the elements presented at this layer are inherited from the BPMN core meta-model and have the usual meaning of flowchart designs. Just the core activities have been reinterpreted to support transactional capabilities. In particular, in BPMN$_{tr}$, activities are modified to support the binding with the related compensation activity. So that, at this level, activities are thought of as couples of basic BPMN activities. Still, the way these activities internally implement their workings and the interactions needed to coordinate composed activities are not formally specified.

Indeed, as a matter of fact, BPMN$_{tr}$ designs (*i*) neglect distribution aspects of the transactional activities, (*ii*) does not specify if activities are atomic or consisting of hidden sub-activities, (*iii*) do not pay attention on how compensations are distributed (e.g. by replicating the same functionality to achieve load balancing).

2.5.2 Naïve Sagas

Saga calculi have been proposed to formalize Long Running Transactions. Among the several existing variants, we will consider the Naïve Sagas of [BMM05] that exhibit the basic concepts needed to model and formalize BPMN$_{tr}$. The "naïve" adjective refers to the execution of the parallel activities. In particular, Naïve Sagas assumes a simplified execution of parallel branches so that, once an activity fails it must wait until the other concurrent branches terminates before continuing. Optimized executions of the parallel blocks have been proposed and implemented in [BMM05] but are out of our scope and so omitted.

Activities in a saga are described at the high level of abstraction, where the elementary actions are not interpreted. Transactional flows are processes built by composing with the standard parallel and sequential composition plus the *compensation pair* construct. Given two actions A and B, the compensation pair $A \div B$ corresponds to a process that uses B as compensation for A. Intuitively, $A \div B$ yields two flows of execution: the *forward flow* and the *backward flow*. During the forward flow, $A \div B$ starts its execution by running the main activity A; when A finishes, B is "installed" as compensation for A, and the control is forwardly propagated to the other stages of the transactions. In case of a failure in the rest of the transaction, the backward flow starts so that the effects of executing A have to be compensated. This is achieved by activating the installed compensation B and, afterward, by propagating the request to compensate to the activities that were executed before A. Note that B is not installed if A is not executed. Having that, with LRTs, the atomicity constraint is relaxed, in case of failure, the compensation activities will bring to a final state that should be different from the initial one, but, anyway, it will be consistent. When clear from context, the compensations will be also referred to as rollback activities.

Syntax In reference to Table 2.6, the (Step) can also be expressed as the 0 process and the single main activity A having no compensation. This last term is a shorthand for $A \div 0$. We assume $\mathcal{A}$ a set of names for atomic activities ranged over by $A, B, \ldots$. Moreover, 0 is considered a special empty activity $0 \notin \mathcal{A}$ that always completes and has no effect.

$$
\begin{array}{lcll}
X & ::= & 0 \mid A \mid A \div B & (\textsc{Step}) \\
P & ::= & X \mid P; P \mid P \parallel P & (\textsc{Process}) \\
S & ::= & \{\!\![P]\!\!\} & (\textsc{Saga})
\end{array}
$$

Table 2.6: Naïve Sagas syntax

Processes P are expressed as basic compensable activities and can be composed either in sequence $P; Q$ or in parallel $P \parallel Q$. A Saga consists of a process P enclosed into a transactional block ($\{\!\![P]\!\!\}$). Sagas achieve atomicity if all the enclosed activities (sub-transactions) reach a consistent state. Either all the components successfully complete all their main activities, or all their installed compensations are executed.

We consider a fragment of original Naïve Sagas where nested sagas (e.g. $\{\!\![P; \{\!\![P']\!\!\}]\!\!\}$) and the programmable compensations (e.g. $P \div Q$) are omitted since their practical need is marginal for our purpose. Without loss of generality, we assume that any activity appears at most once in any saga (resp. process), i.e. that different instances of the same action are named differently.

Example 2.5.2 *The Sagas representation of* BPMN$_{tr}$ *processes depicted in Figure 2.5 is:*

$$
\begin{array}{ll}
(Task_1 \div Comp_1); (Task_2 \div Comp_2) & (\textsc{Sequence}) \\
(Task_1 \div Comp_1) \parallel (Task_2 \div Comp_2) & (\textsc{Parallel})
\end{array}
$$

while for transactional enclosure construct, depicted in Figure 2.6, the intended meaning in Sagas is:

$$
\{\!\![Task \div Comp]\!\!\} \qquad (\textsc{Saga})
$$

Big Step Semantics We now recall the big step semantics of Naïve Sagas introduced in [BMM05]. The execution of Sagas can either evolve in a commitment of the overall transaction, or no effect is externally observed when the execution fails. Sagas are supposed to always terminate in a consistent state, however compensations can internally fail, bringing to an *abnormal termination*. With respect to the semantics defined in [BMM05], we omit the treatment of failure in compensation activities.

The semantic of Saga is given up to a structural congruence over terms, which is defined by the axioms in Table 2.7. Sequential and parallel compositions are both associative having 0 as neutral element. Moreover, the parallel operation is commutative.

$$
\begin{array}{lll}
A \div 0 \equiv A & 0; P \equiv P; 0 \equiv P & (P; Q); R \equiv P; (Q; R) \\
P \parallel 0 \equiv P & P \parallel Q \equiv Q \parallel P & P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R
\end{array}
$$

Table 2.7: Naïve Sagas structural congruence

For the sake of simplicity, in the following the semantics has been presented with an incremental approach by presenting, the reduction rules for the basic activities (Steps) and subsequently the semantics for composing processes in sequence and in parallel.

In the following, $\boxdot$ and $\boxtimes$ represent the possible outcomes of the execution of atomic activities (resp. commit and abort). Let $\mathcal{R} = \{\boxdot, \boxtimes\}$ (ranged over by $\square$) be the set of the possible results of the execution of a saga and $\Gamma \vdash \mathcal{A} \rightarrow \mathcal{R}$ be a function that assigns a result to atomic activities.

The semantics of Sagas is given by two relations $\Gamma \vdash S \xrightarrow{\alpha} \mathcal{R}$ and $\Gamma \vdash \langle P, \beta \rangle \xrightarrow{\alpha} \langle \square, \beta' \rangle$. The former evaluates processes to results and the latter gives the semantics of processes. Both those relations use labels α that amounts to be traces of executions of atomic actions. In the relation $\Gamma \vdash \langle P, \beta \rangle \xrightarrow{\alpha} \langle \square, \beta' \rangle$ β represents the "stack" of installed compensations.

In Table 2.8 we report the reduction rules for Steps. The empty activity 0 is assumed to successfully terminate installing no compensation (rule (Zero)). The rules (S-Act) and (S-Cmp) describe the execution of a compensable activity $A \div B$ in a configuration having a compensation stack β. The former regulates the correct execution of the main activity A; in fact, it evolves into a new configuration having B on the top of the compensation stack and observes the main activity itself. The latter, instead, considers the happening of an internal failure during the execution of A; the activity is now considered not executed, the configuration reaches a state $\boxtimes$, the stack of compensations is executed (and subsequently emptied) by observing α. Notice that it is assumed that compensation execution does not fail, so that the execution of β reaches a $\boxdot$ state.

The evaluation of processes composed in sequence is described by rules

$$\frac{}{\Gamma \vdash \langle 0,\beta\rangle \xrightarrow{0} \langle \boxdot,\beta\rangle}\ (\textsc{Zero})$$

$$\frac{\Gamma(A) = \boxdot}{\Gamma \vdash \langle A \div B,\beta\rangle \xrightarrow{A} \langle \boxdot, B;\beta\rangle}\ (\text{S-}\textsc{Act})$$

$$\frac{\Gamma(A) = \boxtimes \quad \Gamma \vdash \langle \beta, 0\rangle \xrightarrow{\alpha} \langle \boxdot, 0\rangle}{\Gamma \vdash \langle A \div B,\beta\rangle \xrightarrow{\alpha} \langle \boxtimes, 0\rangle}\ (\text{S-}\textsc{Cmp})$$

Table 2.8: Naïve Sagas semantics: step

$$\frac{\Gamma \vdash \langle P,\beta\rangle \xrightarrow{\alpha} \langle \boxdot,\beta''\rangle \qquad \Gamma \vdash \langle Q,\beta''\rangle \xrightarrow{\alpha'} \langle \Box,\beta'\rangle}{\Gamma \vdash \langle P;Q,\beta\rangle \xrightarrow{\alpha;\alpha'} \langle \Box,\beta'\rangle}\ (\text{S-}\textsc{Step})$$

$$\frac{\Gamma \vdash \langle P,\beta\rangle \xrightarrow{\alpha} \langle \boxtimes, 0\rangle}{\Gamma \vdash \langle P;Q,\beta\rangle \xrightarrow{\alpha} \langle \boxtimes, 0\rangle}\ (\text{A-}\textsc{Step})$$

Table 2.9: Naïve Sagas semantics: sequence

(S-Step) and (A-Step) in Table 2.9. If P correctly ends, the resulting configuration of the execution $P;Q$ will be the one reached by Q executed in a context having, as initial compensation stack, the one produced by the execution of P. On the contrary, if P fails, Q will never be activated and a $\boxtimes$ state will be reached.

Table 2.10 reports the rules for parallel composition, for which three cases have to be considered. If both parallel processes are successfully executed (S-Par), their main activities α and α' are observed and their compensations are installed. Notice that both the observed activities and the installed compensations are composed in parallel. If any process fails, say P, its concurrent process Q has to compensate, by executing β' and, subsequently the stack of pre-installed compensations β is executed and emptied. This behavior is described by rule (S-Par-1). Finally, the rule (S-Par-2), states that if both activities internally fail, no compensations will be installed for them and just the pre-installed compensations β have to be executed.

The rule for saga is reported in Table 2.11. A saga is executed as an isolated

$$\frac{\Gamma \vdash \langle P,0\rangle \xrightarrow{\alpha} \langle \boxdot,\beta'\rangle \quad \Gamma \vdash \langle P',0\rangle \xrightarrow{\alpha'} \langle \boxdot,\beta''\rangle}{\Gamma \vdash \langle P \parallel Q,\beta\rangle \xrightarrow{\alpha \parallel \alpha'} \langle \boxdot,\beta' \parallel \beta'';\beta\rangle}\ (\text{S-P{\small AR}})$$

$$\frac{\Gamma \vdash \langle P,0\rangle \xrightarrow{\alpha} \langle \boxtimes,0\rangle \quad \Gamma \vdash \langle Q,0\rangle \xrightarrow{\alpha'} \langle \boxdot,\beta'\rangle \quad \Gamma \vdash \langle \beta';\beta,0\rangle \xrightarrow{\alpha''} \langle \boxdot,0\rangle}{\Gamma \vdash \langle P \parallel Q,\beta\rangle \xrightarrow{(\alpha \parallel \alpha');\alpha''} \langle \boxtimes,0\rangle}\ (\text{S-P{\small AR}-1})$$

$$\frac{\Gamma \vdash \langle P,0\rangle \xrightarrow{\alpha} \langle \boxtimes,0\rangle \quad \Gamma \vdash \langle Q,0\rangle \xrightarrow{\alpha'} \langle \boxtimes,0\rangle \quad \Gamma \vdash \langle \beta,0\rangle \xrightarrow{\alpha''} \langle \boxdot,0\rangle}{\Gamma \vdash \langle P \parallel Q,\beta\rangle \xrightarrow{(\alpha \parallel \alpha');\alpha''} \langle \boxtimes,0\rangle}\ (\text{S-P{\small AR}-2})$$

Table 2.10: Naïve Sagas semantics: parallel

$$\frac{\Gamma \vdash \langle P,0\rangle \xrightarrow{\alpha} \langle \Box,\beta\rangle}{\Gamma \vdash \{\![P]\!\} \xrightarrow{\alpha} \Box}\ (\text{S-S{\small AGA}})$$

Table 2.11: Naïve Sagas semantics: saga

transaction, hence its enclosed process P is evaluated in a context having the compensation stack empty.

At this level the compensation stack is not considered since if P internally fails, the compensations are executed and the stack emptied otherwise the whole transaction is considered consistently ended and the stack explicitly removed.

Chapter 3

Signal Calculus

An important aspect strictly related to Service Oriented Architectures is the possibility to aggregate functionalities exposed by several services to build more complex ones. The compositionality of services can be achieved by coordinating their interactions through suitable work-flow languages.

In the last decade, several languages tailored to coordinate services have been proposed. Most of such languages adopt message oriented mechanisms, so that the interactions are modeled by describing the flow of messages that are exchanged among parties. The idea is that the network infrastructure is directly reflected in the architectural model of such languages so that the messages represent the core constructs for implementing communications among services.

Nevertheless, the coordination of services can be regulated by describing their work-flow in terms of "how they react" to the evolution of their execution environment. The system undergoes modifications that are represented by suitable events that are promptly notified to the interested partners. Components passively observe changes that are applied to the environment and trigger handling routines at their occurrences, while, actively they concur to modify the system state. Essentially, distributed pieces of functionalities access a common global state that acts as a bridge for the involved parties hiding to them the underlying network structure.

The adoption of the event notification paradigm yields several benefits since it features high level coordination mechanisms that allow programmers/designers to decouple components and rely entirely on event handling. Specifically for SOAs, this strategy enforces the loosely coupling among services and enables multi-cast asynchronous communications.

This chapter describes the Signal Calculus, a variant of the π*-calculus with* *explicit primitives to deal with event notification and component distribution.*

Through a running example, we provide a comparison of the our programming model and the π-calculus formalism.

3.1 Introduction

The cornerstone of our approach is the adoption of an event notification (EN) paradigm as a modeling and programming abstraction for SOAs. In fact, we propose an event based language to model and orchestrate services in a SOA scenario. The EN paradigm allows programmers/designers to focus on how each computational entity behaves upon occurrences of environmental stimuli instead of considering complex interactions among several agents.

In the first instance, components are designed in "isolation", focusing on the way they act once an event of a certain kind occurs and on how they raise notifications for events internally occurred during their computations. Once deployed, such components are injected in a particular network and "linked" together by exploiting the subscription mechanism. This two-phase design relaxes the interdependencies among components achieving an high degree of loosely coupling and, consequently, reveals to be particularly suitable for SOAs.

The event notification paradigm, whose main characters and features have been widely discussed in Section 2.3, is here recalled through an example that is used to analyze the effectiveness of the π-calculus to deal in presence of situations typically event based. Through this analysis will emerge some problematics that evidence how some primitives, native in EN, comport some efforts or cumbersome code to be expressed with π-calculus formalism.

The chapter concludes with the definition of a process calculus, the Signal Calculus, that represents the starting point of our work and has been adopted as specification language for the service coordination. The principles treated at this level bring to the definition of a "network model" which entirely relies on event notification and has been adapted to better fit the programmers needs in a SOA environment.

3.2 Alarm system: a running example

To gain confidence with events, we start from the description of a typical scenario we are used to, the alarm controller system, whose behavior can be simplified as follows:

"when a door is forced, the alarm is activated and, in the meanwhile*, both the apartment owner and the police office telephones are contacted to inform that a theft is possibly taking place"*.

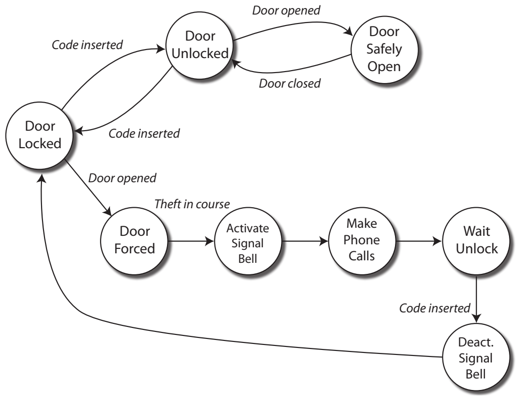


Figure 3.1: Alarm system automaton

In Figure 3.1 it is depicted a sketch of the automaton for the alarm system controller. As usual, in the automata notation, nodes represent the states, or the configurations, that are reachable by the system, while the arcs are labeled with the "conditions" (corresponding to *events* in our formalism) under which transitions among states are possible. Namely, a *transition* from a state to another is only possible if there is an arc connecting the two states and it happens once the event on that arc has been discovered. The arcs having no label represent the ϵ transitions in the automaton definition. Namely the transitions on those arcs occur regardless the events happened in the system. For the sake of simplicity, we have drawn a flat form of the real configuration, since several components are involved in the real situation (e.g. the bell ring, the phone caller and the door controller agents). The whole schema will be presented and formally described in the following sections.

We assume to start from a situation where the door is closed and the alarm engaged. Two possible events can occur. Either the code is inserted, and as consequence the alarm deactivated, or the door is opened and then a probable theft is discovered. In the former case, the configuration is considered "safe", namely the door can be further closed and the alarm reactivated. In the latter case, instead, the alarm agent must notify the signal bell and the phone caller of the happening of the "event" *theft in course*. In the meanwhile, the controller enters a state “wait unlock” allowing the apartment’s owner to insert the code to deactivate the alarm. Once the code is inserted the alarm system reaches again the initial state (this is a simplified assumption) and the signal bell is informed that the alarm has reentered.

3.2.1 Modeling the alarm controller in π-calculus

Now we discuss how the π-calculus primitives may be used to model, in an event notification style, the example introduced in Section 3.2. The protocol can be viewed as composition of two distinct phases: *i)* an initial phase where agents perform their subscriptions and *ii)* a “running” phase that describes the life-cycle of the overall system in terms of flow of events.

As notation, in this section we will use the terms *event notifications* for values passed through channels and *publishers* and *subscribers* for the π processes that access channels for sending (resp. receiving) notifications through such channels.

Phase 1: subscriptions The name passing capability exposed by π-calculus allows to implement in a natural way the subscription of agents to topics of interest. The channels adopted for receiving notifications are private to the subscribers. The subscription is implemented by communicating the private channels to the publishers so that they can suitably deliver notifications on them. We just present a short example for implementing the subscription of the *Phone* agent to topics *alarm*. In the code reported in Code 3.1, the *Phone* agent accesses the $subscribe_{alarm}$ channel of *Alarm* to require the subscription on its occurrences of events *alarm*. The channel ch_{alarm}, through which the *Phone* will receive notifications, is so passed to the *Alarm* agent.

$$\begin{array}{lcl} Alarm & = & subscribe_{alarm}(ch_{alarm}) \\ Phone & = & \overline{subscribe_{alarm}}\ ch_{alarm} \end{array}$$

Code 3.1: Event subscription in π-calculus

We assume that, before activating the second phase of the protocol, all the subscriptions have been suitably performed, similarly to Code 3.1. Moreover, in the following, we start from a configuration in which agents *Phone* and *Bell* share the channel ch_{alarm} to model they subscribe on the same event.

Phase 2: notifying events In Code 3.2, we report a preliminary sketch of the π-calculus coding of the alarm system presented before.

At this phase of the protocol, π channels are intended as the vehicle for delivering, to other participants, notifications of raised events. As consequence, channel names are assigned according to the events they notify. For example, an *output* on the channel *code* stands for a delivering of a notification for the event representing the situation “the user has inserted the code”. Analogously the triggering of an event is represented by the respective *input* action on the respective channel.

We provide a compact representation of this part of the protocol, in terms of the Calculus of Communicating Systems (CCS) [Mil80] by omitting name communication that is indeed useful for implementing the event subscription capability. The reading and writing operations on channels are to be intended as notifications for events so that writing $\overline{alarm} \mid alarm$ will be intended as a shorthand for $\overline{ch_{alarm}}evt \mid ch_{alarm}(x)$ where evt is the instance of the event notified on the channel ch_{alarm}. That is because the instance of the event itself is not relevant for this example and so is omitted for better readability.

$$
\begin{array}{llcl}
Alarm & = & & code.\overline{unlocked}.code.\overline{locked} \\
 & & + & forced.\overline{alarm}.\overline{alarm}.code.\overline{disableBell}.\overline{locked} \\
Door & = & & opened.closed \\
 & & + & locked.(unlocked + opened.\overline{forced}) \\
Bell & = & & alarm.Ring.disableBell \\
Phone & = & & alarm.MakeCalls \\
System & = & & \overline{locked} \mid !Alarm \mid !Door \mid !Bell \mid !Phone
\end{array}
$$

Code 3.2: Alarm system in π-calculus

The π agents are defined in Code 3.2. The *Alarm* consists of a process that, non deterministically, is activated by a code insertion and continues by notifying that the door has been unlocked. Next, the process waits for the reinserting of the code and notifies that the door has been locked again. The other possible behavior expected by the *Alarm* consists in receiving, from the *Door* controller, a notification of a door forced detection. In this case, a notification is sent twice on channel *alarm*, one will be triggered by the *Bell* and the other one by the *Phone* devices. The activation order of the two *alarm* handlers in not relevant. Once notified the *alarm*, the controller becomes idle waiting for the code insertion in order to deactivate the alarm. The deactivation is finally notified to the *Bell* system. The system now reaches its initial state by notifying that the door is locked again.

The *Door* agent interacts with both the “environment” (the *User*), to catch if the door has been opened or closed, and with the *Alarm* system to be informed about the state of the alarm engine. We assume that the alarm can be activated only once the door is closed, so that, once the user "safely" opens the door, he is assumed to close the door before inserting the code again. Three traces are possible. The door is unlocked and the user is able to *open* and *close* the door several times. But, once the user inserts the code, the alarm control notifies that the door is in a *locked* state. Now, two possible behaviors are caught. Either the door is unlocked or the door is opened and a *forced* event is thrown.

The *Bell* and *Phone* processes are simple to understand. The internal logic used to implement the activation of the ringing bell and of the phone calling are not described since out of our scope. We refer to these two behaviors as generic processes *Ring* and *MakeCalls*, respectively. With the respect to the π-calculus, these instructions correspond to the silent prefix τ.

The interactions with the human are represented by processes built as combination of actions in the set:

$$UserActs = \{\overline{opened}, \overline{closed}, \overline{code}\}$$

under the assumption that user cannot perform actions of the form $\overline{closed}.\overline{closed}$ for obvious reasons. The *System* is obtained by composition of the agents defined in Code 3.2. In its initial configuration the alarm is active.

3.2.1.1 Alarm system revisited

The sketch of the alarm system, reported in Code 3.2, hides an ambiguous behavior due to the choice construct. Referring to the *Door* agent, two input prefixes *opened* are involved in different branches of execution. Since it holds that $!P \equiv P|!P$, it is not possible to distinguish when the sending of a message on channel *opened* should trigger which one between the two possible read operations. In a situation in which the door has been locked the *Door* agent reaches a state of the form:

$$unlocked + opened.\overline{forced} \mid Door$$

Once an opened situation occurs, the process can continue by signaling a forced event or the "reinstalled" process can consume it by reactivating the behavior previously described. The disambiguation is obtained by introducing an ad hoc internal communication *start* (a guard on the recursion) that states when the process has terminated the whole behavior and can be executed again. In Code 3.3, the *Door* has been fixed to solve the ambiguous behavior.

The *start* name is restricted to the scope of *Door* agent and internally used to ensure the proper application of replicated process. The replicated process has

$$
\begin{array}{lcll}
Alarm & = & & code.\overline{unlocked}.code.\overline{locked}.\overline{start} \\
 & & + & forced.\overline{alarm}.code.\overline{disableBell}.\overline{locked}.\overline{start} \\
Door & = & & opened.closed.\overline{start} \\
 & & + & locked.(unlocked.\overline{start} + opened.\overline{forced}.\overline{start}) \\
Bell & = & & alarm.Ring.disableBell.\overline{alarmPhone} \\
Phone & = & & alarmPhone.MakeCalls \\
System & = & & (\nu start)(\overline{start} \mid !start.Alarm) \mid (\nu start)(\overline{start} \mid !start.Door) \\
 & & & \mid !Bell \mid !Phone
\end{array}
$$

Code 3.3: Alarm system revisited

to wait that the previous work-flow instance has been completely finished before starting a new handling of communications coming from the system.

It is easy to see that, after some steps, the initial configuration of the process *Door* reaches a state:

$$(unlocked.\overline{start} + opened.\overline{forced}.\overline{start}) \mid start.Door$$

so that the *Door* agent is able to respond only to two situations: the code is reinserted and the alarm deactivated or the door is forced.

Analogous considerations hold for the *Alarm* agent since the *code* input prefix is present at beginning of a branch and inside the other one. Notice that the two *start* names have to be considered distinct since restricted.

Other conflicts can arise from Code 3.2 because of the usage of the channel *alarm* for implementing the multiple delivering (multi-cast) of events to both *Bell* and *Phone* agents. In fact, there are no warranties that the two instances of the event *alarm* are distinctly caught by these processes. In particular, due to the recursive behavior of processes, it can happen that a single process (e.g. *Bell*) consumes the same notification twice. In Code 3.3 is shown a possible solution that involves sequential delivering of notification firstly to *Bell* and secondly to *Phone*. In this case the channels are distinct and so the conflicts on access to the notification channel avoided. Alternatively the multi-cast can be ensured by applying guards to the recursive behaviors of conflicting agents in analogy to the solution previously discussed for the agent *Alarm*.

3.2.1.2 A few remarks: π-calculus and Event Notification

In reference to the concepts exposed in Section 2.3, we now characterize how the EN structure is reflected in π-calculus.

The usage of channels as core mechanism for both implementing the subscriptions and for conveying notifications to subscribed agents seems to be adequate for decoupling publishers from subscribers. The connections among processes are created and acceded according to subscription policies. As consequence, the system becomes more *adaptable* to the changes of the environment lying around. To support the subscription in the π-calculus, the subscriber (the notification consumer) has to communicate the channel used for receiving notifications to the process acting as publisher (the notification producer). Consequently, publishers and subscribers access a shared channel for delivering (resp. consuming) notifications for raised events. This way, processes can be opportunely replaced by keeping the access to the same channel invisibly to their counterparts. For example, in reference to the alarm system, the *Phone* process can be substituted by a different process that respond to a change of the initial configuration (e.g. the user phone number changes) or to code modifications (e.g. the process involves additional tasks). The *Alarm* agent will not be affected by these modifications.

Nevertheless, the illusion of loosely coupling is just offered to subscribers. In fact, the absence of multi-cast primitives requires the notifier itself to be reconfigured to support new subscriptions. In reference to the example described above, the further addition of a process interested in receiving notifications for the *alarm* events requires the modification of the *Alarm* controller that must be conscious of the reconfiguration of the system (the notification must be sent three times).

As a matter of fact, the π-calculus results more suitable when multi-to-one message delivering policy is needed, while more complex becomes to encode multi-cast (which turns out to be central in event based systems).

The adoption of event notification paradigm will drive the specification of the Signal Calculus, that directly relies on events as machinery of interactions enabling multi-cast asynchronous communications.

Section 3.3 reports the original contribution of Signal Calculus that relies on topic based event notifications, while Section 3.5 recalls a subset of the typed version of SC presented in [FGST07].

3.3 Signal Calculus

The Signal Calculus (SC) is a process calculus in the style of the asynchronous π-calculus [Mil93] specifically designed to describe coordination policies of services distributed over a network.

The original contribution, presented in [FGS06b], has given the basis for designing a programming framework for SOA, the Java Signal Core Layer. Furthermore, the structure of SC has been reflected in a Domain Specific Language (DSL) for which a graphical representation has been provided.

Similarly to the π-calculus and other analogous process algebra, SC describes interactions happening among distributed concurrent components. Differently from the π-calculus, where interactions happen in a *point-to-point* manner, in SC a broadcasting approach is preferred to adhere to the event notification ideas. Additionally, SC lays on locations as key rule for modeling where actual computations are taking place. The adoption of locations is close to the approach of the Ambient calculus, devised by Cardelli and Gordon in [CG98], that, introduces the notion of locality of components in terms of membranes in which computations take place. Albeit the Ambient calculus envisages the modeling of concurrent systems in presence of code mobility, our main goal is to model coordination of distributed computations, so SC does not feature code mobility.

Moreover, the adoption of the event notification tailors broadcast service activations and features a more natural description of the work-flow. The attention is shifted on how the activities are involved in response to events raised in the system. This level of abstraction increases the loosely coupled interactions among components since the caller has no longer to know which component is able to service a type of request. The kind of functionalities exported by components is retrieved from its declared reactions that can be added or modified at run-time.

Standard EN paradigms rely on brokered communication (c.f. Section 2.3); SC, instead, adopts a non-brokered notification mechanism where subscription and emission are *explicitly* tagged with naming information, e.g. the name of the target components. This avoids any centralization point by distributing the connection managing to each involved participant. Brokered EN paradigms are more appropriate when coordination is handled by an orchestrator, while non-brokered approaches fit much better when choreography is adopted. For a detailed comparison among brokered and non-brokered EN see [HG06]. Additionally, SC implements a subscription strategy that differs from usual event notification approach. No explicit request of the subscriber is required, the publisher itself can decide during its life-cycle which agents have to be involved into the coordination. This approach is closer to the method invocation style since the caller (corresponding to the publisher) is capable of deciding, for each object (corresponding to the subscriber), which methods (the reactions) have to be invoked. Informally, in relation to the OOP paradigm, the SC model allows to deal with distributed objects with a dynamic type (reactions can be dynamically installed) and enables multi-cast asynchronous method invocations. This interpretation considers flows corresponding to references to methods.

Hereinafter, we recall the main ingredients of the calculus by presenting its original proposal, with few examples, concluding with some variants and the motivations standing behind. The chapter ends with the coding of the alarm controller with the SC calculus.

Syntax

The Signal Calculus model is centered around the notion of *component*. A component $a[B]^R_F$ is a service identified by a unique name a, the public address of the service. The active computations, called *behaviors* (B), are confined inside components. The expressions R and F, after *reactions* and *flows*, respectively, have to be thought of as the service interface. Notifications of events are encapsulated inside *signals* consisting of messages containing information regarding the managed resources and the events raised during internal computations. Signals are classified by *topics*; specifically, each component specifies (i) the *reaction* to be activated upon reception of signals of a certain topic and (ii) the set of event *flows*, namely the collection of component names the emitted signals will be delivered to. Hence, while reactions define the interacting behavior of the component, flows define the component view of the coordination policies. The SC primitives allow one to *dynamically* modify the topology of the coordination policies by adding new flows and reactions to components. New signals can be sent either by autonomous components or as reaction to other signals. SC focuses only on the primitives needed to design coordination protocols. Hence, the data conveyed inside signals is not explicitly modeled. At implementation level, instead, this aspect has been treated providing to the developer the constructs to attach data to signals and to manage it.

We now introduce the main syntactic categories of our calculus together with some notational machineries. We assume an infinite set $\mathcal{T}$ of *topic names* ranged by $\tau_1, ..., \tau_k$, a infinite set $\mathcal{A}$ of *component names* ranged by $a, b, c....$ We use $\vec{a}$ to denote a set of names $a_1, ..., a_n$. Finally, with names $N, M, ... \in \mathcal{N}$ we denote *networks*.

The syntax of reactions (R) is reported in Table 3.1. A reaction is a multi-set (possibly empty) of unit reactions. A unit reaction $\tau \gg B$ triggers the execution of the behavior B upon reception of a signal tagged by the topic τ. Informally, we say that τ corresponds to the "signature" of reactions, while B declares its "body".

(REACTIONS)	$R ::=$	$\emptyset_R$	*(Nil)*
		$\tau \gtrdot B$	*(Unit reaction)*
		$R \otimes R$	*(Composition)*

Table 3.1: SC reactions

Flows (F) are described by the grammar in Table 3.2. A flow is a set (possibly empty) of unit flows. A unit flow $\tau \rightsquigarrow \vec{a}$ implements the subscription of a set of component names $\vec{a}$ for the topic τ. Since flows are defined on the component interface, their configuration is locally maintained by each component. SC clearly inverts the subscription policy since the subscription is assumed to be required by notification producers in behalf of consumers. For convenience, in the following we denote with $\tau \rightsquigarrow a$ the flows having just an end point connected, instead of using the set notation $\tau \rightsquigarrow \{a\}$. Moreover, in the following we refer to R_1 and R_2 as *subreactions* of the reaction composition $R_1 \otimes R_2$.

(FLOWS)	$F ::=$	$\emptyset_F$	*(Nil)*
		$\tau \rightsquigarrow \vec{a}$	*(Unit flow)*
		$F \oplus F$	*(Composition)*

Table 3.2: SC flows

Reactions and flows are defined up-to a structural congruence ($\equiv$). Indeed, we assume that parallel composition of flows and reactions are associative, commutative and with $\emptyset_F$ (resp. $\emptyset_R$) behaving as identity. Furthermore, we let:

$$(\tau \rightsquigarrow \vec{a}) \oplus (\tau \rightsquigarrow \vec{b}) \equiv \tau \rightsquigarrow \vec{a} \cup \vec{b} \tag{3.3.1}$$

Now, we report the flow projection $(F)\downarrow_\tau$, consisting of an auxiliary function defined on flow terms that is used to retrieve, given a topic name τ, the set of subscribed components.

$$(\emptyset_F)\downarrow_\tau = \emptyset \qquad (\tau \rightsquigarrow \vec{a})\downarrow_{\tau'} = \begin{cases} \vec{a} & \text{if } \tau' = \tau \\ \emptyset & \text{otherwise} \end{cases} \qquad (F \oplus F')\downarrow_\tau = (F)\downarrow_\tau \cup (F')\downarrow_\tau$$

The syntax of component behaviors is reported in Table 3.3. The first two productions represent, respectively, the empty and the internal actions; $\epsilon.B$ corresponds the silent prefix of the π-calculus, reported in Section 2.4. The rule *reaction update* offers the possibility to extend the reaction part of the component interface by appending to it a new reaction. In presence of conflicts, for already defined reactions predicating on the same topic, at activation phase, one of them will be executed non deterministically. In a similar way, the *flow update* rule modifies the component flows. The flow composition operator, instead, acts as the union, so that, once a flow of the form $\tau \rightsquigarrow \vec{a}$ is added to the current interface, the conflicting names a_i already defined on that flow are removed. This constraints naturally comes from the definition of $\vec{a}$ as set of names and is guaranteed by Equation (3.3.1). An asynchronous *signal emission*, $\mathtt{out}(\tau)$, spawns into the network a set of envelopes containing the signal, one for each component name declared in the flow for topic τ. If we compare the treatment of asynchrony in SC with *pi*-calculus we observe that, in our formalism, the output operation (emission) is not presented as *bare output*. Still, as will be clarified by the semantic rules (Out) and (In) in Table 3.7, the continuation of output operation is activated without waiting the consumption of the message from the receiver. SC in fact, explicitly introduce a sort of queue notion. The output consists in spawning, immediately, a message on the network that will be responsible to deliver, in a second moment, the message to the proper consumer.

Behaviors can be composed in parallel. The bang replication $!B$ represents a behavior that can always activate a new copy of the behavior B. Notice that all the actions have been presented in a prefixed form $act.B$, with act representing an atomic action in the set $\{\epsilon, rupd(R), fupd(F), out(\tau)\}$. Once the atomic step act ends, its continuation B is activated. When it is clear from the context, we will omit the Nil behavior, writing act instead of $act.0$.

(Behaviors)	$B ::=$	0	*(Nil)*
		$\epsilon.B$	*(Internal step)*
		$\mathtt{rupd}(R).B$	*(Reaction update)*
		$\mathtt{fupd}(F).B$	*(Flow update)*
		$\mathtt{out}(\tau).B$	*(Asynchronous signal emission)*
		$B \vert B'$	*(Parallel composition)*
		$!B$	*(Bang)*

Table 3.3: SC behaviors

Components are structured to build a *network* of services. In addition, network provides the facility to transport *envelopes* containing the signals exchanged among components. This feature is the core of the asynchronous communication in SC. Envelopes $\langle\tau\rangle@a$ yield the signal emitted for notifying the event τ, associated with its addressee, the subscribed component a. The grammar for networks is reported in Table 3.4.

(NETWORKS)	$N ::=$	$\emptyset_N$	*(Empty net)*
		$a[B]^R_F$	*(Component)*
		$\langle\tau\rangle@a$	*(Signal envelope)*
		$N \| N$	*(Parallel composition)*

Table 3.4: SC networks

The SC components describe locations on which the current computations, the behaviors, are taking place. As consequence, networks are flat, namely there is no hierarchy of components and are considered well formed when component names are not replicated.

Non-deterministic and recursive behaviors

Even though SC does not explicitly provides constructs for describing the non-deterministic application of behaviors, it is possible to exploit the reactions to code this construct. In fact, while for the flow interface a constraint guarantees that the flow compositionality is idempotent (c.f. Equation (3.3.1)), for reaction interface, instead, it is given the possibility to compose reactions having the same signature. This way, conflicting reactions are non-deterministically activable. Yet, since reactions are bound to behaviors, the reaction activation can be used to model the non-deterministic activation of their internal behavior. As consequence, $B_1 + B_2$ can be coded according to the following schema:

$$a[B_1 + B_2]^R_F \triangleq a[\mathtt{out}(\tau_+)]^{R \otimes \tau_+ > B_1 \otimes \tau_+ > B_2}_{F \oplus \tau_+ \rightsquigarrow a}$$

where we assume the topic name τ_+ local to component a (the name restriction on names will be formalized in Section 3.5). The behaviors are declared inside reactions having τ_+ as activating topic. Such reactions are added to the reaction interface of a and, accordingly, the flows are connected to the component a so that it will receive the signal ($\mathtt{out}(\tau_+)$) that will trigger the non-deterministic activation of one of two behaviors B_1 or B_2.

Remarkably, analogous considerations hold for the replication of behaviors. In fact, though SC presents the *bang* operator ($!B$), it can be simulated by exploiting the persistence of reactions in the following manner:

$$a[!B]^R_F \triangleq a[\mathtt{out}(\tau_!)]^{R\otimes\tau_! > \mathtt{out}(\tau_!)|B}_{F\oplus\tau_!\rightsquigarrow a}$$

Operational Semantics

SC semantics is defined in a reduction style [BB92]. We first introduce a structural congruence over behaviors and networks. We assume $(N, \|, \emptyset_N)$ and $(B, |, 0)$ commutative monoids. The structural congruence for component behaviors ($\equiv_B$) is defined in Table 3.5. As usual the bang operator allows us to express recursive behaviors.

$$B_1|B_2 \equiv_B B_2|B_1 \qquad (B_1|B_2)|B_3 \equiv_B B_1|(B_2|B_3) \qquad 0|B \equiv_B B \qquad !B \equiv_B B|!B$$

Table 3.5: Structural congruence on behaviors

Structural congruence for networks $\equiv_N$ is the smallest relation satisfying axioms in Table 3.6. A component having *nil* behavior and empty reaction can be considered as the empty network since it has no internal active behavior and cannot activate any behavior upon reception of a signal. Two components, having the same name a, are considered structurally congruent if their internal behaviors, reactions and flows are structurally congruent. When it is clear from the context, we will use the symbol $\equiv$ for both $\equiv_B$ and $\equiv_N$.

$$N\|M \equiv_N M\|N \qquad (M\|N)\|O \equiv_N M\|(N\|O) \qquad \emptyset_N\|N \equiv_N N$$

$$a[0]^{\emptyset_R}_F \equiv_N \emptyset_N \qquad \frac{F \equiv F' \quad R \equiv R' \quad B \equiv_B B'}{a[B]^R_F \equiv_N a[B']^{R'}_{F'}}$$

Table 3.6: Structural congruence on networks

The reduction relation of networks ($\rightarrow$) is defined by the rules in Table 3.7. The rule (Skip) states that the silent prefix ϵ is executed regardless the context in which the component is acting. The rule (Rupd) appends to the component reactions a further reaction R' built on the grammar defined in Table 3.1. As stated before, the appending of a new reaction acting on a topic τ, for which there were already installed reactions, is allowed and brings to a non deterministic activation of "‘conflicting"’ reactions. The rule (Fupd) extends the component flows by appending to it F'. We remark that, flows are defined by a relation between a topic name and a set of component names, so that, for the same topic, no replication of target names is allowed. The duplicates will be removed. The notification of an event is delivered according to the rule (Out). The set $\vec{b}$ of components subscribed on a for notifications of topic τ is obtained by applying the flow projection. For each name b_i defined in the set, a message is *spawned* into the network that is demanded to asynchronously deliver the message to the target component. We often refer to this action with the term *signal emission*. Signals spawned into the network are enriched with information needed by the network to retrieve the target component and are called *signal envelopes* or, more simply, *envelopes*. Once a component a is ready to consume a signal envelope targeted to it, the rule (In) is applied. Notice that signal emission rule (Out) and signal receiving rule (In) do not consume, respectively, the flow and the reaction of the component. This feature provides SC with a further form of recursion. The rules (Struct) and (Par) are standard rules. In the following, we use $N \rightarrow^{+} N'$ to represent a network N that is reduced to N' after a finite number of steps.

To illustrate the expressiveness of SC, we consider an example of a producer P and a consumer C accessing a shared resource in mutual exclusion where C can consume the resource only after P has produced it.

Example 3.3.1 *The* SC *code for P and C is*

$$
\begin{aligned}
P &= p[\epsilon.\mathtt{out}(produced)]^{consumed > \epsilon.\mathtt{out}(produced)}_{produced \rightsquigarrow c} \\
C &= c[0]^{produced > \epsilon.\mathtt{out}(consumed)}_{consumed \rightsquigarrow p}
\end{aligned}
$$

where, to improve the readability, topics are named after the events they represent (produced and consumed) instead of using τ's.

Initially, P performs some internal steps to produce the proper message (ϵ) and afterward notifies that the resource is available ($\mathtt{out}(produced)$). Upon notification, C executes the behavior of the corresponding reaction installed so that it accesses the shared resource (ϵ) and sends a notification ($\mathtt{out}(consumed)$) to inform P that the resource has been consumed.

$$\frac{}{a[\epsilon.B|B']^R_F \longrightarrow a[B|B']^R_F} \text{ (SKIP)}$$

$$\frac{R'' = R \otimes R'}{a[\texttt{rupd}(R')|B]^R_F \longrightarrow a[B]^{R''}_F} \text{ (RUPD)}$$

$$\frac{F'' = F \oplus F'}{a[\texttt{fupd}(F')|B]^R_F \longrightarrow a[B]^R_{F''}} \text{ (FUPD)}$$

$$\frac{(F)\downarrow_\tau = \vec{b}}{a[\texttt{out}(\tau).B|B']^R_F \longrightarrow a[B|B']^R_F \| \prod_{b_i \in \vec{b}} \langle\tau\rangle@b_i} \text{ (OUT)}$$

$$\frac{R = R' \otimes \tau > B'}{\langle\tau\rangle@a\|a[B]^R_F \longrightarrow a[B|B']^R_F} \text{ (IN)}$$

$$\frac{N \equiv M \to M' \equiv N'}{N \to N'} \text{ (STRUCT)} \qquad \frac{N \to N'}{N\|M \to N'\|M} \text{ (PAR)}$$

Table 3.7: SC reduction rules

3.4 A FEW REMARKS: BASIC SC AND CCS

SC reveals the adequacy of event notification to model the coordination aspect in a service oriented architecture. The adoption of the EN paradigm, for managing coordination policies brings in two main advantages. On the one hand, it is a well known programming model and, on the other hand, it permits the distribution of coordination activities and of the underlying computational infrastructure. The dynamic flavor of the SC permits to model a wide range of coordination policies for service-oriented applications (e.g. in [FGS06a] the primitives have been used to deal with dynamic and heterogeneous networks). However, other primitives providing high-level programming abstractions are desirable. In particular, information associated to signals is not structured and topics cannot be created dynamically. Furthermore, the notion of session abstraction is missing: components cannot keep track of concurrent event notifications. Such features were not initially considered in SC as it was intended to give the minimal structure for dealing with events. As a matter of fact, this makes calculus very similar to the

CCS [Mil80] (where no communication of names is possible). Notice that the absence of constructs for restricting topics makes SC less expressive than CCS.

All the other constructs present in CCS can be found in SC. More precisely, SC presents constructs for implementing asynchronous communications, recursive and non-deterministic behaviors, and internal computations (ϵ corresponds to the silent action), channels correspond to flows, while input and output actions correspond to reactions and signal emission, respectively.

Nevertheless, in SC, inputs on channels are intended in terms of entry points of the components. Consequently, once activated, reactions are not removed from the component interface. This aspect better adheres to the constructs adopted in the SOAs, but on its turn involves some drawbacks. The persistence of reactions makes impossible to code the alarm system previously presented. For example, referring to the example of Section 3.2, it is not so easy to faithfully represent the agents since sometimes it is necessary to disable some reactions. (For instance, in relation to the *Door* agent, two possible readings on the channel *opened* are possible.)

If reactions may trigger many instances of a work-flow, it may be useful to be able to decide which of them should be activated.

In Section 3.5, we start from the extension of the SC calculus, presented in [FGST07], that permits managing of *sessions* and the handling *structured topics* via suitable types. This extension will provide mechanisms to deal with both persistent and "one time" reactions as direct implication of the introduction of session tracking mechanisms to the model.

3.5 Managing sessions in Signal Calculus

We now extend the structure of SC by introducing the notion of *signal type* for supporting the session labeling on topics of conveyed signals. The SC dialect supporting sessions is in the following referred to as SC_σ, still, when clear from the context, we will indicate this variant with the standard notation SC.

The definition of a signal type, has a twofold role. In the first instance, the signals come equipped with a session thus allowing to distinguish the work-flow instance in which the events of a certain topic has occurred. Secondly, the *sessions* determine a sort of "virtual communication link" among publishers and subscribers that can be established despite they do not need to know each other's names. Intuitively, a *session* identifies the scope within which an event is significant: partners that are not in this scope cannot react to events of the session. Additionally sessions make possible to implement name passing among components. In fact, depending on how a signal type is engaged, the session part can be used to convey topics. The introduction of sessions impacts the structure of both reactions and of networks. Two kinds of reactions are defined: the *lambda reaction*, triggered by signals independently from their session, and the *check reaction*, that reacts only to signals acting in a specific session.

For instance, a publisher can emit an event with topic τ and session τ' that should be received by subscribers that can react to events of type τ *and* τ'. Hence, typing allows subscribers to filter their events of interest (as usual in type-based EN). Conversely, publishers exploit type information to specify which (kind of) subscribers should react to events.

Furthermore, the session handling mechanisms provided by SC_σ can deal with multi-party sessions in a natural way. Sessions can be communicated to other components that can participate in a restricted work-flow instance.

In [FGST07], we argued that the introduction of structured topics complements these approaches by providing higher-level constructs on sessions that allow a closer formalization of more abstract protocols where multi-party sessions are relevant. To demonstrate the adequacy of our approach, we have applied SC_σ to specify the OpenID protocol [RF], a complex protocol for managing distributed identities whose behavior requires many parties to participate to the same session. In the following the sessioning mechanism is adopted to model the alarm system example described in Section 3.2.

Syntax

The introduction of sessions requires some changes to the structure of signals that becomes a pair of topics $\tau \diamond \tau'$, in the following referred as *signal type*, with τ the event topic and τ' the related session. Consequently, the structure on reactions, behaviors and networks is adapted to support the new changes. The flows, instead, remain unaltered.

Since sessions are used to enclose the scope of the work-flow in which events are significant, the reactions should be equipped with mechanisms for discovering this information. The rules for reactions are reported in Table 3.8. Two kinds of unit reactions have been defined. The *lambda reaction* that acts as previously defined reactions, without considering the session in which the caught event has been raised, and the *check reactions* that discriminate the notification triggering depending on their session. In the former case, the session τ' acts as a binder for the reaction so that, once received a signal, the parameter is instantiated with the formal parameter representing the session, and B is executed in the new scope of τ'. In the latter case, instead, τ' is considered a free name. Additionally, lambda reactions are considered persistent, while, check reactions, once activated, are removed from the component interface. Check reactions represent reactions that are "specialized" for a precise session.

(Reactions)	R	$::=$	$\emptyset_R \mid R \otimes R$	
		$\mid$	$\tau\lambda\tau' \gtrdot B$	*(Lambda reaction)*
		$\mid$	$\tau\diamond\tau' \gtrdot B$	*(Check reaction)*

Table 3.8: SC_σ reactions

The syntax of behaviors is given in Table 3.9. The *signal emission* $\mathtt{out}(\tau\diamond\tau').B'$ describes the emission of a signal having topic τ over the session identified by the topic τ'. Finally, topics can be freshly generated by using the *topic creation* primitive. The other rules are the same presented in Section 3.3.

(Behaviors)	B	$::=$	$0 \mid \epsilon.B \mid \mathtt{rupd}(R).B \mid \mathtt{fupd}(F).B \mid B \vert B' \mid !B$	
		$\mid$	$\mathtt{out}(\tau\diamond\tau').B'$	*(Signal emission)*
		$\mid$	$(\nu\tau)B'$	*(Topic creation)*

Table 3.9: SC_σ behaviors

Network syntax is defined in Table 3.10. A network can be empty $\emptyset_N$, a single component $a[B]^F_R$, or the parallel composition of networks $N \| N'$. Networks carry signals exchanged among components. The signal emission spawns into the network, for each target component, an "envelope" $\langle\tau\diamond\tau'\rangle@a$ containing the signal and the name a of the target component. Finally, the last production allows to extend, over networks, the scope of freshly generated names n ranged over by $\mathcal{T} \cup \mathcal{A}$. Networks can restrict both component and topic names, thus allowing to hide behavior of a part of the network. Remarkably, only topic names can be communicated among components. For multiple application of restriction operator we use the notation $(\nu a, b, ...)$ in behalf of $(\nu a)(\nu b)(\nu ...)$.

Free and bound names for networks, reactions, behaviors and flows are defined by structural induction in the usual way. We summarize the main rules in the following:

$$\begin{array}{ll}
fn(\tau\diamond\tau' \gtrdot B) = fn(B) \cup \{\tau, \tau'\} & bn(\tau\diamond\tau' \gtrdot B) = bn(B) \setminus \{\tau, \tau'\} \\
fn(\tau\lambda\tau' \gtrdot B) = fn(B) \cup \{\tau\} \setminus \{\tau'\} & bn(\tau\lambda\tau' \gtrdot B) = bn(B) \cup \{\tau'\} \setminus \{\tau\} \\
fn((\nu\tau)B) = fn(B) \setminus \{\tau\} & bn((\nu\tau)B) = bn(B) \cup \{\tau\} \\
fn((\nu\tau)N) = fn(B) \setminus \{\tau\} & bn((\nu\tau)N) = bn(B) \cup \{\tau\}
\end{array}$$

(Networks)	$N ::=$	$\emptyset_N \mid a[B]_R^F \mid N \| N$	
		$\mid \langle \tau \diamond \tau' \rangle @ a$	*(Signal envelope)*
		$\mid (\nu n) N$	*(New)*

Table 3.10: SC_σ networks

The structural congruence over reactions, flows and behaviors is the smallest congruence relation that satisfies the commutative monoidal laws for $(R, \otimes, \emptyset_R)$, $(F, \oplus, \emptyset_F)$ and $(B, |, 0)$. Also, for the structural congruence over behaviors, the following laws hold:

$$(\nu\tau)0 \equiv_B 0, \qquad (\nu\tau)B|B' \equiv_B (\nu\tau)(B|B'), \text{ if } \tau \notin fn(B')$$

and, whenever $B \equiv_B B'$:

$$\mathtt{rupd}(\tau\lambda\tau' > B).B'' \equiv_B \mathtt{rupd}(\tau\lambda\tau' > B').B''$$
$$\mathtt{rupd}\big(\tau\diamond\tau' > B\big).B'' \equiv_B \mathtt{rupd}\big(\tau\diamond\tau' > B'\big).B''$$

If $B \equiv_B B'$, the following rules hold for structural congruence over reactions:

$$\tau\lambda\tau' > B \equiv_R \tau\lambda\tau' > B'$$
$$\tau\diamond\tau' > B \equiv_R \tau\diamond\tau' > B'$$

Similarly, $\equiv_N$ is the smallest equivalence relation that respects the commutative monoidal laws for $(N, \|, \emptyset_N)$ and the following ones:

$$a[0]^F_{\emptyset_R} \equiv_N \emptyset_N, \qquad (\nu\tau)\emptyset_N \equiv_N \emptyset_N, \qquad (\nu\tau)N\|N' \equiv_N (\nu\tau)(N\|N'), \text{ if } \tau \notin fn(N')$$

$$\frac{F_1 \equiv_F F_2 \quad B_1 \equiv_B B_2 \quad R_1 \equiv_R R_2}{a[B_1]^{F_1}_{R_1} \equiv_N a[B_2]^{F_2}_{R_2}}, \qquad \frac{\tau \notin fn(R) \cup fn(F) \cup \{a\}}{a[(\nu\tau)B]^F_R \equiv_N (\nu\tau)a[B]^F_R}.$$

Reduction Rules

With the introduction of sessions in SC, the structure of signal envelopes and reactions has been modified to treat signals equipped with the session information and reactions that can be, as usual, either defined in persistent manner with no discrimination on the session conveyed (lambda), or specialized (check) for a well defined session and consumed after reacting to an event. Moreover, in [FGST07], the activation of reaction is disciplined by a type system on topics so that check reactions are prioritized with the respect of lambda reactions, if both of them are present on the same topic. In Table 3.11 we report a simplification of the original contribution with the explicit prioritized activation of check reactions. For further details on the structured topics see [FGST07].

$$\frac{(F)\!\downarrow_{\tau} = \vec{b}}{a[\mathtt{out}(\tau\diamond\tau').B|B']^{R}_{F} \longrightarrow a[B|B']^{R}_{F} \| \prod_{b_i \in \vec{b}} \langle \tau\diamond\tau' \rangle @ b_i} \ (\textsc{Out})$$

$$\frac{R = R' \otimes \tau\lambda\tau'' \gtrdot B' \wedge \nexists (R'', B'')\ R' = R'' \otimes \tau\diamond\tau' \gtrdot B''}{\langle \tau\diamond\tau' \rangle @ a \| a[B]^{R}_{F} \longrightarrow a[B|\{\tau'/\tau''\}B']^{R}_{F}} \ (\textsc{Lambda})$$

$$\frac{R = R' \otimes \tau\diamond\tau' \gtrdot B'}{\langle \tau\diamond\tau' \rangle @ a \| a[B]^{R}_{F} \longrightarrow a[B|B']^{R'}_{F}} \ (\textsc{Check})$$

$$\frac{N \longrightarrow N'}{(\nu n)N \longrightarrow (\nu n)N'} \ (\textsc{New})$$

Table 3.11: SC_{σ} reduction rules

The (Out) rule is adapted to deal with signals related to a session. The *flow projection* auxiliary function remains the same presented before. Given a topic τ and a flow set F it returns the set of component names subscribed on that topic. The (Lambda) and (Check) describe, respectively the activation of lambda and check reactions. In the former case, the reaction activated remains in the interface of the component and a substitution is applied in the scope of the behavior for the bound name τ' corresponding to the session. In the latter case, no substitution is needed, since both names are free, and the activated reaction is removed from the interface of the component. The meaning of the (New) rule is obvious.

3.5.1 Some useful patterns in SC

In order to provide an overview on the possibilities offered by the SC_{σ} language, here we discuss on some useful patterns and constructs can be formalized by extending the structure of the language or by simply presenting the ideas for explicitly coding them.

3.5.1.1 Joining events

Since SC components are autonomous entities communicating through asynchronous primitives, it could be useful to introduce a lightweight mechanism for synchronizing the execution of concurrent tasks (*join*). In this scenario we show how to encode a form of join synchronization among concurrent tasks.

Figure 3.2 shows an emitter E, two intermediate components C_1 and C_2, and the join service J. The emitter E starts the communications raising two events of different topics toward C_1 and C_2 that perform an internal computation and then notify their termination by issuing an event to the join service. The component J waits that both the intermediate services have completed their tasks and then executes its internal behavior B. The signals sent to C_1 and C_2 are both related to the same session τ that is later used by J to apply the synchronization on the same work-flow. Clearly, the two intermediate services C_1 and C_2 can concurrently perform their tasks, while the execution of the service J can be triggered only after the completion of their execution.

Briefly, J must perform its computation B_J only after both C_1 and C_2, that are running concurrently, have completed their tasks.

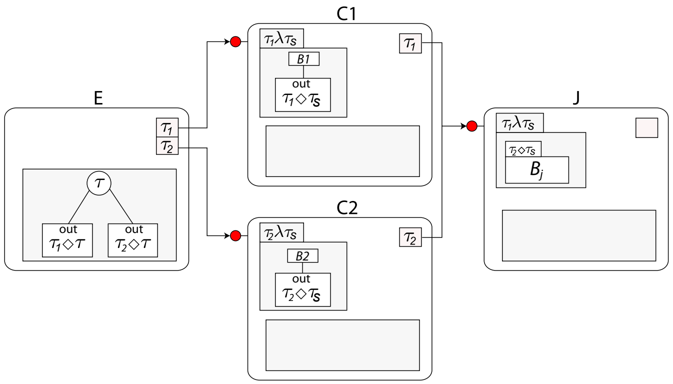


Figure 3.2: Abstract graphical notation of join

This example can be modeled by the SC network $E \| C_1 \| C_2 \| J$, where:

$$E = e[(\nu\tau)(\mathtt{out}(\tau_1 \diamond \tau) | \mathtt{out}(\tau_2 \diamond \tau))]^{\emptyset_R}_{\tau_1 \rightsquigarrow C_1 \oplus \tau_2 \rightsquigarrow C_2}$$

$$C_i = c_i[0]^{\tau_i \lambda \tau_s > \epsilon.\mathtt{out}(\tau_i \diamond \tau_s)}_{\tau_i \rightsquigarrow j} \qquad i = 1, 2$$

$$J = j[0]^{\tau_1 \lambda \tau_s > \mathtt{rupd}(\tau_2 \diamond \tau_s > B_j)}_{\emptyset_F}$$

and the internal behaviors of C_i are represented by the silent actions (ϵ).

To provide an intuition of the network configuration described here, in Figure 3.2, we adopt an abstract graphical notation that depicts the components defined into the network and details their interfaces, in terms of flows and reactions, and their actual computation, the behavior. In particular, reactions are boxed placed at the upper-left corner, flows boxes placed at the upper right and the behavior is a box at the bottom of each component figure. Reactions are labeled with their signature. Behaviors are given (reading from top to bottom) by circles (the restriction), nested reactions or links that regulate the sequence (the activation) of steps (e.g. from the restriction in E two outgoing arcs represent the parallel execution of the output operations on which the restriction has been applied). The nesting of reaction boxes in J states that once the lambda reaction $\tau_1 \lambda \tau_s$ is activated, the contained check reaction $\tau_2 \diamond \tau_s > B_j$ is installed. Finally, the arrows connecting components correspond to flows (e.g. E has two flows $\tau_1 \rightsquigarrow C_1$ and $\tau_2 \rightsquigarrow C_2$) .

The join component has only one active reaction installed for signals having topic τ_1. When the two intermediary services forward their signals, the envelope containing the τ_2 event cannot be consumed by the join, and remains pending over the network. The reception of the τ_1 envelope triggers the activation of the join generic reaction. The reaction *reads* the session of the signal τ_1 and creates a new specialized reaction for the signal topic τ_2. This reaction can be triggered only by signals that refer to the session received by the τ_1 signal. When such kind of signal is received, the proper behavior B is executed. Notice that the creation of the specialized reaction for the τ_2 implies that a possible pendent envelope is consumed.

3.5.1.2 Rendez-vous in SC

We now give an intuition of how the synchronous output primitive (Emit) may be programmed in SC. For this purpose we use the notation $\texttt{out}(\tau_a \diamond \tau_s); B$ for representing a situation where the behavior B has to wait the consumption of the message from the receiver before being actived. We start from a network configuration:

$$(\nu \tau_s) a[\texttt{out}(\tau_a \diamond \tau_s); B]_F^R \| b[B']_F^{\tau_a \lambda s > B''}$$

Notice that how the session τ_s has been defined is not strictly relevant. For the sake f simplicity, here we assume it to be a fresh name. Under the assumption that b is the only process subscribed to topics τ_a of component a, the network can be defined by the following SC code:

$$(\nu\tau_s, \tau_{ack})\Big(a[\mathtt{out}(\tau_a \diamond \tau_s).\mathtt{rupd}(\tau_{ack} \diamond \tau_s \gtrdot B)]^R_F \| b[B']_F^{\tau_a \lambda s \gtrdot \mathtt{out}(\tau_{ack} \diamond s).B''}\Big)$$

It becomes clear that the rendez-vous application to SC components requires arrangements to the structure of both parties (publishers and subscribers), that must adhere a common strategy and explicitly support the needed message exchanging for allowing the publisher to be notified of the correct reception of the message from the subscriber.

3.5.1.3 Flow removal and reaction hiding

A pragmatic feature of the event notification paradigm relies on the possibility to obtain dynamic subscriptions to revoke the previously defined ones. The service oriented applications can benefit from the dynamic subscription feature to naturally reflect the reconfigurations of the network topology. The possibility to revoke previously defined subscriptions has not been formulated in the specification of SC language.

For this reason, we now introduce a modification to the original SC structure in order to enable the subscription removal. Since flows and reactions are our linguistic device for implementing EN subscription mechanism, the revocation of subscriptions, from the publisher side, can be achieved by applying the flow removal primitive. The syntax of SC is extended by adding the following operator:

$$\mathtt{fdel}(\tau \rightsquigarrow \vec{a}).B \qquad \textit{(Flow remove)}$$

whose semantics is given by the following rule:

$$\frac{F = F' \oplus \tau \rightsquigarrow \vec{b} \ \land \ (F')\downarrow_\tau \cap \vec{b} = \emptyset}{a[\mathtt{fdel}(\tau \rightsquigarrow \vec{b}).B|B']^R_F \longrightarrow a[B|B']^R_{F'}} \ (\textsc{FDel})$$

The (FDEL) rule isolates, from the flow interface F, a subset F' for which holds that $(F')\downarrow_\tau \cap \vec{b} = \emptyset$. This constraint ensures that the flows with topic τ starting from a will never be targeted with names in $\vec{b}$ and is needed since flow composition is idempotent.

Moreover, equally useful may result the possibility to remove a reaction from the component interface, to allow, for example, subscribers to revoke their handling task for a given topic of events. However, the reaction removal results more complex to code in SC. That is because it is not always possible to uniquely retrieve, from a given signature, the corresponding reaction. In fact, SC gives the possibility to enter several reactions triggering on the same signature in order to feature their non-deterministic activation. As consequence, the reaction removal primitive cannot be directly coded.

Nevertheless, the flow removal primitive can help to solve this issue. Suppose to have the following component:

$$a[B']_F^{\tau\lambda s \gtrdot B}$$

it can be converted into a (sub-)network of the form:

$$(\nu\tau_h, l_a)\Big(a[B']_{F \oplus \tau_h \rightsquigarrow l_a}^{\tau\lambda s \gtrdot \mathtt{out}(\tau_h \diamond s)} \| l_a[0]_F^{\tau_h\lambda s \gtrdot B}\Big)$$

Initially a new component l_a, whose restricted name known in a, is introduced. The reaction previously present on a is moved on the l_a interface and a link connecting a to l_a on the restricted topic τ_h is established. Finally, the flows of a are replicated on the l_a interface so that, if its internal behavior B involves emissions, the delivering is applied according to the a subscriptions.

This coding respects the intended meaning of original component a under the assumption that further alteration to the flows on a for topic τ are reported to the flow interface of l_a.

Now that the reaction has been delegated to the restricted component, it is trivial to implement the reaction hiding on component a. It suffices that the link $\tau_h \rightsquigarrow l_a$ is removed from a to "hide" the reaction installed on l_a, conversely for "un-hiding" the reaction, the flow must be reinstalled.

From the outside, a remains the only visible component and represents so the unique entry point of the process composition. For this reason in the following we will use the term "sub-network" to represent a configuration where part of the network is hidden.

3.5.2 Modeling the alarm controller in SC

Now we discuss how the alarm system example presented in Section 3.2 can be modeled in SC. Hereafter we will use the same structure (in terms of agents and events) of the solution reported in Code 3.2.

The SC extension with sessions allows to directly handle work-flow sessions by exploiting the session information related to signals. Here we make use of check reactions and reaction handling for coding the alarm running example. Under the assumption that the whole chain of steps performed by the *User* are

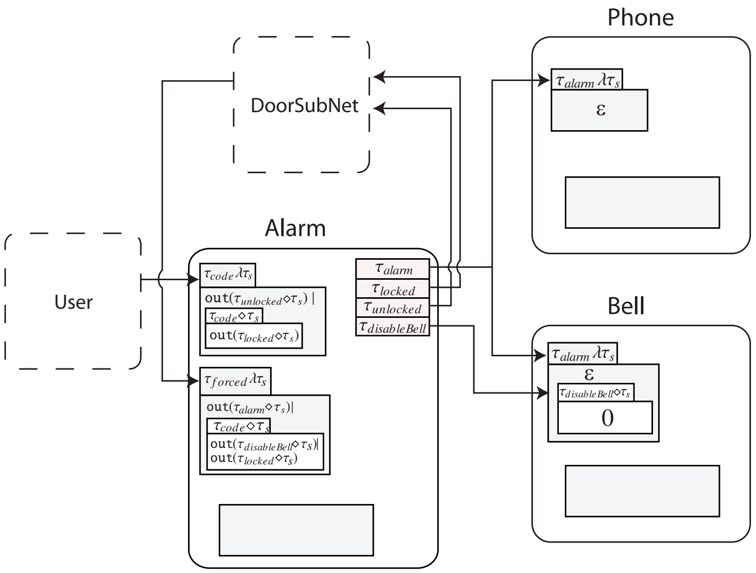

Phone
DoorSubNet
Alarm
User
Bell
$\tau_{alarm}\lambda\tau_s$
ε
$\tau_{code}\lambda\tau_s$
out($\tau_{unlocked}\diamond\tau_s$) |
$\tau_{code}\diamond\tau_s$
out($\tau_{locked}\diamond\tau_s$)
τ_{alarm}
τ_{locked}
$\tau_{unlocked}$
$\tau_{disableBell}$
$\tau_{forced}\lambda\tau_s$
out($\tau_{alarm}\diamond\tau_s$)|
$\tau_{code}\diamond\tau_s$
out($\tau_{disableBell}\diamond\tau_s$)|
out($\tau_{locked}\diamond\tau_s$)
$\tau_{disableBell}\diamond\tau_s$
0

Figure 3.3: Alarm system in SC: overall system

enclosed into the same session[1], the system is coded by applying the reactions shown in Code 3.4 and the flows of Code 3.5 to the network configuration reported in Code 3.6.

In order to give a more intuitive representation of the SC network we adopt the abstract graphical representation introduced in Section 3.5.1.1. In particular, Figure 3.3 depicts the overall network with the exclusion of the *Door* component, since, as will be shown in Figure 3.4, it involves the adoption of reaction hiding accordingly to the concepts previously exposed.

Referring to Figure 3.3, the *Alarm* interface consists of two reactions. Such component is able to handle notifications of *code* insertions (coming from *User*) and notifications of *forced* events (coming from *Door*). Once a *code* notification is caught by *Alarm*, it signals (to *Door*) that the alarm has been *unlocked* and installs a check reaction for capturing the next occurrence of *code* event. Consequently, the reception of a further *code* notification will be captured by the specialized (check) reaction that will signal that the alarm is again *locked*. The

[1]This constraint is not a limitation, it states that each time the user starts a new connection to the system, a new session will be created to distinguish the work-flow instance.

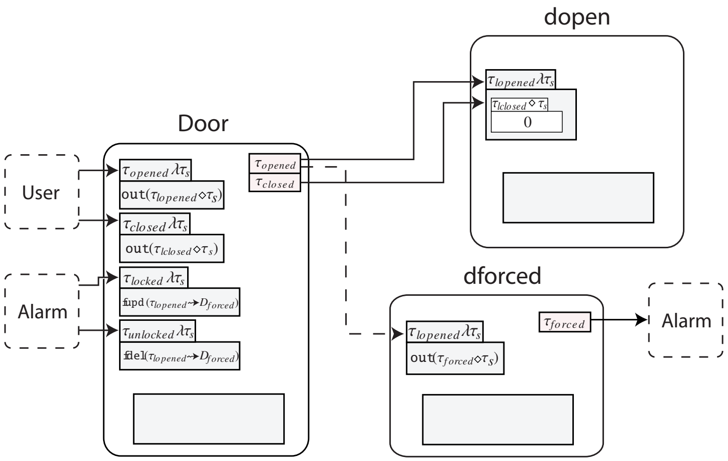

dopen
$\tau_{lopened}\lambda\tau_s$
$\tau_{lclosed}\diamond\tau_s$
0
Door
$\tau_{opened}\lambda\tau_s$
out($\tau_{lopened}\diamond\tau_s$)
$\tau_{closed}\lambda\tau_s$
out($\tau_{lclosed}\diamond\tau_s$)
$\tau_{locked}\lambda\tau_s$
fupd ($\tau_{lopened}\rightsquigarrow D_{forced}$)
$\tau_{unlocked}\lambda\tau_s$
fdel($\tau_{lopened}\rightsquigarrow D_{forced}$)
τ_{opened}
τ_{closed}
User
Alarm
dforced
τ_{forced}
$\tau_{lopened}\lambda\tau_s$
out($\tau_{forced}\diamond\tau_s$)
Alarm

Figure 3.4: Alarm system in SC: the door sub network

$$
\begin{array}{lcl}
R_{alarm} & = & \tau_{code}\lambda\tau_s \gtrdot \left(\begin{array}{l} \texttt{out}(\tau_{unlocked}\diamond\tau_s) \mid \\ \texttt{rupd}\Big(\tau_{code}\diamond\tau_s \gtrdot \texttt{out}(\tau_{locked}\diamond\tau_s)\Big) \end{array} \right) \otimes \\
 & & \tau_{forced}\lambda\tau_s \gtrdot \left(\begin{array}{l} \texttt{out}(\tau_{alarm}\diamond\tau_s) \mid \\ \texttt{rupd}\Big(\tau_{code}\diamond\tau_s \gtrdot \texttt{out}(\tau_{disableBell}\diamond\tau_s) \mid \texttt{out}(\tau_{locked}\diamond\tau_s)\Big) \end{array} \right) \\
R_{door} & = & \tau_{opened}\lambda\tau_s \gtrdot \texttt{out}(\tau_{lopened}\diamond\tau_s) \otimes \\
 & & \tau_{closed}\lambda\tau_s \gtrdot \texttt{out}(\tau_{lclosed}\diamond\tau_s) \otimes \\
 & & \tau_{locked}\lambda\tau_s \gtrdot \texttt{fupd}(\tau_{lopened} \rightsquigarrow D_{forced}) \otimes \\
 & & \tau_{unlocked}\lambda\tau_s \gtrdot \texttt{fdel}(\tau_{lopened} \rightsquigarrow D_{forced}) \\
R_{dopen} & = & \tau_{lopened}\lambda\tau_s \gtrdot \texttt{rupd}(\tau_{lclosed}\diamond\tau_s \gtrdot 0) \\
R_{dforced} & = & \tau_{lopened}\lambda\tau_s \gtrdot \texttt{out}(\tau_{forced}\diamond\tau_s) \\
R_{bell} & = & \tau_{alarm}\lambda\tau_s \gtrdot \epsilon.\texttt{rupd}(\tau_{disableBell}\diamond\tau_s \gtrdot 0) \\
R_{phone} & = & \tau_{alarm}\lambda\tau_s \gtrdot \epsilon
\end{array}
$$

Code 3.4: Alarm System Interfaces: Reactions

reception of a *forced* event is handled in *Alarm* by sending an *alarm* notification (to *Phone* and *Bell* components) and installing a reactions that, consuming the further code insertion, requires the deactivation of the signal bell and notifies that the door is locked again.

The reading of the *Phone* agent is trivial while the *Bell* contains a reaction that, after received an *alarm* notification, installs a check reaction for capturing the further request of bell deactivation.

Notice that the usage of check reaction has here solved the non deterministic activation of *code* branches arisen in π coding.

Moreover, in Figure 3.4 we give a closer look of a part of the sub-network involving the door agent and the two hidden components D_{opened} and D_{forced}. A further restricted name *lopened* is defined to implement local communications among the components in the sub-network. The *Door* agent receives from the *User* notifications for *opened* and *closed* events and acts as a proxy for the hidden components. The *opened* notification is forwarded to the *dopened* component and, depending on the configuration reached, to the *dforced* one. The connection to this last component is removed once an *unlocked* signal has been received and restored once the τ_{locked} arrives.

$$\begin{array}{lcl}
F_{User} & = & \tau_{code} \rightsquigarrow alarm \oplus \tau_{opened} \rightsquigarrow door \oplus \tau_{closed} \rightsquigarrow door \\
F_{Alarm} & = & \tau_{unlocked} \rightsquigarrow door \oplus \tau_{locked} \rightsquigarrow door \oplus \\
 & & \tau_{alarm} \rightsquigarrow \{phone, bell\} \oplus \tau_{disableBell} \rightsquigarrow bell \\
F_{Door} & = & \tau_{lopened} \rightsquigarrow dopened \oplus \tau_{lclosed} \rightsquigarrow dforced \\
F_{Dforced} & = & \tau_{forced} \rightsquigarrow alarm
\end{array}$$

Code 3.5: Alarm System Interfaces: Flows

$$\begin{array}{lcl}
DoorSubNet & = & \nu(dopened, dforced, \tau_{lopened}, \tau_{lclosed}) \\
 & & \left(door[0]^{R_{door}}_{F_{door}} \parallel dopened[0]^{R_{dopened}}_{\emptyset_F} \parallel dforced[0]^{R_{dforced}}_{F_{dforced}}\right) \\
Network & = & user[...]^{\emptyset_R}_{F_{user}} \parallel alarm[0]^{R_{alarm}}_{F_{alarm}} \parallel bell[0]^{R_{bell}}_{\emptyset_F} \parallel phone[0]^{R_{phone}}_{\emptyset_F} \parallel DoorSubNet
\end{array}$$

Code 3.6: Alarm System: Network

3.6 Concluding remarks

The adoption of sessions in the SC language has given the possibility to support the name passing facilities present in the π-calculus.

The key differences between π-calculus and SC rely on the message delivering and on the input operation.

The policies for implementing the message delivering are kept from basic SC formulation so the considerations remain the same given when comparing SC with CCS.

Reactions correspond to the π input operations on channels. Besides, SC declares two kinds of reactions.

The lambda reactions $\tau\lambda\tau' \gg B$ can be thought of as π processes of the form $\tau(\tau')B$ where τ corresponds to the channel, τ' to the formal parameter and B to the continuing process. Nevertheless, the activation of an input in π removes the operation while in SC the corresponding reaction remains installed.

The check reactions $\tau\diamond\tau' \gg B$ instead, are non persistent like π channels, still such reactions do not activate bindings. The session parameter τ' is here used to inhibit the activation of the reaction itself. Analogous results can be obtained by exploiting matching mechanisms on the formal parameter of inputs and reintroducing the message once the matching fails.

The sessioning mechanism reveals to be so relevant for SOAs that recent formal specifications, such as Muse [BLMT08] and SCC [BBC+06], to cite a few, use the session as center point of the interaction.

The introduction of sessions in the SC formalism, opens the possibility to deal with coordination patterns usual in the work-flow languages (e.g. the join) and furthermore to distinguish several running instances of the same work-flow in response to several activations.

However the calculus SC can be enriched by considering signals as tagged nested lists [AB05, HP03], which represent XML documents, and conversation schemata as abstractions for XML Schema types [xml04]. This extension of conversation schemata lead to a more general notion of reaction based on pattern matching or unification in the style of [BBM05]. A further extension can provide component and schema name passing, modelling a more dynamic scenario.

SC can describe dynamic orchestrations through reaction and flow update primitives. These primitives have effects only on the component view of the choreography, namely a component cannot update the reaction or the flow of another component. Flow management can be enriched providing a primitive to update remote flows. This primitive should spawn a *flow update envelope* into the network. The update of remote reaction is more difficult, since a reaction contains code and than is necessary to formalize and implement the code migration.

Moreover, enabling the transmission of component names, through signals, should improve the dynamicity of networks, since new components can be discovered at run-time and involved into existing network topologies.

Chapter 4

Java Signal Core Layer

*The structure of the network model defined in the Signal Calculus has been reflected in a prototypical middleware, called Java Signal Core Layer (*JSCL*), consisting of a set of Java API for programming distributed components interacting by notifying events.*

Mainly, JSCL *offers the run-time support for executing* SC *networks in a distributed and open environment where components assume the meaning of services in the more general context of SOAs.*

In order to separate the concepts related to the network infrastructure from the ones strictly related to the service coordination, JSCL *has been structured as multi layered architecture.*

The lower architectural layer exhibits network adapters that allows components to interact in an uniform manner by adhering to the SC *network specifications, independently from the underlying network infrastructure.*

Additionally, JSCL *declares a set of high level mechanisms tailored to define complex coordination patterns with the aim to reduce the development efforts.*

Notably, the middleware features data encapsulation inside exchanged notifications so that components can communicate the information needed to properly handle the raised events and features constructs for synchronizing concurrent flows of executions and enforces the concept of sessions to inhibit the delivering to components that are out of the scope of a work-flow session.

The coordination primitives presented in JSCL *can be mixed with the internal instructions of components, hiding anyway the topology of the environmental network in which services are acting. Analogously to the exception handling mechanism, the components have the possibility to notify events that happens during their computations.*

4.1 Introduction

Most of the proposed SOA solutions, present services in terms of computational units interacting by exchanging documents. The observational behavior of components involved in a coordination comes through flow of messages.

Jscl adopts an *event-driven* solution, against the *data-driven* one by shifting the attention on what happens inside components instead of on which messages they exchange with the environment. This way, it becomes necessary to explicitly supply primitives to treat those events raised by the components.

As usual in event notification paradigm, components can subscribe their interest for externally raised events or notify their internal events to interested partners. The artifacts for event notifications are *signals*. According to the SC specification, signals are tagged by topics according to the class of events they represent. Additionally, signals are able to encapsulate data.

Components participate to a coordination as *reactive* agents. Reactions implement the business logic of subscribers and declare which behavior activate according to the topic they are related to. Each component can declare the set of event topics it is interested in, and the reaction to activate once the related notification has been received. Reactions supply to the subscribers the machinery for implementing subscriptions.

Conversely, flows are the linguistic device for implementing subscriptions from the publishers side and for the notification dispatching to the subscribed components (publication). Publishers and subscribers are here denoted as *emitters* and *handlers*.

Abstractly, in an analogy with OOP paradigm, reactions are intended as methods, whose signature is given by the handled topic, while flows correspond to references to methods declared on a component interface. Moreover, in our model, reactions and flows can be declared and modified at runtime and method invocations are implemented as asynchronous single calls. In this metaphor, object types are dynamic, in the means of dynamic reconfiguration of their interfaces. *Flows* act similarly to the "delegate" mechanism introduced in modern programming languages (e.g. *C#*) but, here, components are considered in an open and distributed environment. Finally, the flow removal capability, introduced in Section 3.5.1.3, allows components to retreat their previously declared reactions and modify their connectivity.

The OOP metaphor can be interpreted as: services adopt the *subscription* mechanism to "provide" functionalities and the *publication* mechanism to "require" a functionality to subscribed services. This strategy introduces a high level of independence and adaptability of the services involved into the coordination. First, each service has no acknowledge about the other services, thus achieving loosely coupling. Furthermore, if several services are able to offer the same functionality, they can be freely rearranged (*adaptability*) by exploiting the basic EN capabilities (e.g. one service can remove its subscription and be replaced by the subscription of a new service acting in behalf of it).

JSCL has been proposed with the aim to furnish a concrete support for the development of services. Further details that were not captured by SC specification are now taken into account. In the first instance, JSCL, differently from SC, adopts a non anonymous publication mechanism so that it is possible to retrieve, from each signal, the component that raised the event (the sender). This property is useful, for example, to avoid erroneous replications of the same notification to a single handler. The subscription policies instead keeps the same characteristics of SC specifications, namely it is implemented in non anonymous way.

Flows can be specialized for a session, analogously to SC check reactions. This capability allows to express customizations of the network topology according to a work-flow session. This characteristic is not presented in SC since, at abstract level, the possibility to trace multiple instances of the same component running in concurrent work-flows is not considered.

Moreover, JSCL permits to mark flows with logical expressions that are evaluated on the actual parameters of the outgoing signals and are used to enable the conditional delivering.

This chapter is structured as follows. The JSCL architecture is presented in Section 4.2. Once provided the minimal infrastructure from implementing the concepts exposed in SC, an overview of the JSCL coding of the alarm system example, described above, is exposed in Section 4.3. The network infrastructure exposed by JSCL middleware is adapted, in Section 4.4, to handle services acting in an heterogeneous networks where the visibility of service addresses is not always guaranteed.

4.2 ARCHITECTURE

In order to achieve independence from the network infrastructure, and to separate the aspects regarding the communication primitives from the ones needed to deal with events, JSCL has been designed and implemented with a multi-layered architecture as shown in Figure 4.1.

The lower level, called *Inter Object Communication Layer* (`iocl`), acts as a bridge among the underlying networking infrastructure and the higher architectural levels. The `iocl` specifies a set of requirements the network must expose, namely the naming facilities for identifying components and the capabilities for data serialization and for asynchronous message delivering. On top of this specification networks adapters, in the means of artifacts for SC networks, can be defined and plugged at `iocl` level.

The *Signal Based Layer* (`sbl`) layer provides all the facilities to deal with SC components, signals, reactions and flows by introducing suitable Java API.

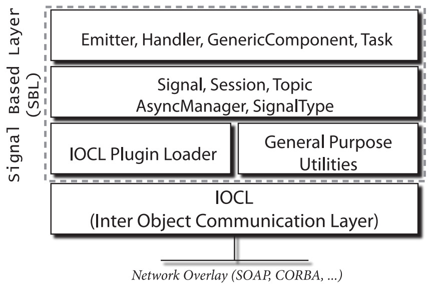

Signal Based Layer (SBL)
Emitter, Handler, GenericComponent, Task
Signal, Session, Topic AsyncManager, SignalType
IOCL Plugin Loader
General Purpose Utilities
IOCL (Inter Object Communication Layer)
Network Overlay (SOAP, CORBA, ...)

Figure 4.1: JSCL Architecture

Further insights regarding the `iocl` and `sbl` architectures can be found in Section 4.2.1 and Section 4.2.2, respectively.

4.2.1 Inter Object Communication Layer

To abstract from the particular underlying network, JSCL introduces a lower architectural layer, the Inter Object Communication Layer (`iocl`) that defines the minimal structure a network must expose to allow the distribution and the interaction of components. The primitives for creating, publishing and retrieving distributed components are defined at this layer. The `iocl` acts in behalf of the network hiding to the programmer the mechanism used to exchange messages among components. Moreover, the middleware allows several instances of `iocl` to coexist, using the addressing mechanism to identify the network infrastructures.

The `iocl` is demanded to provide message serialization facilities according to the adopted network. For example, if components are implemented as web services, signals will be serialized as XML documents using SOAP bindings. Handling of signals and their transformations, in accordance with the network overlay in use, comports several benefits.

The `iocl` networks advocate an high degree of *interoperability*. Not only services running on different platforms can access the same facilities offered by the underlying network structure, as promised by usual SOA based infrastructures (e.g. Web Services, CORBA, etc.), but also solutions coming from heterogeneous networks (e.g. mobile phones, sensor networks, etc.) can coexist and inter-operate, under the assumption that these systems adhere to a common network model.

For this purpose, it is important to establish how components are addressed, in the network overlay they are acting in, and to furnish registry facilities that give the possibility to retrieve and publish services.

Hence, the `iocls` have been developed as plug-ins that provide a common structure to the upper standing layers offering *transparency*. Ad-hoc proxy artifacts are defined at this level by offering the developer the usual object oriented facilities. The lower level treats aspects regarding the communication among components, while, at upper level, these aspect are hidden and the focus is put on what happens during a computation and on how other components react upon the reception of notifications.

The clear separation among the aspects related to the network infrastructure and the logic of components brings two main benefits: *code reusing* and *scalability*. The components can, in fact, migrate on different `iocls` without compromising their internal design. Besides, ad-hoc `iocl` implementations can benefit of the facilities offered by the underlying network and exploit them to advantage load-balancing, component distribution and message buffering depending on the infrastructure their are acting.

The `iocl` layer covers two main aspects: *i)* how components are reflected on the actual network infrastructure and *ii)* how messages are exchanged among components.

To separately treat these aspects two distinct profiles have been defined. In Figure 4.2 is reported the «iocl_core» profile that presents the interface for implementing `iocl` plug-ins and the concepts needed to address components. In Figure 4.3 we report the «iocl_comm» profile consisting of the API needed for implementing the message serialization.

We now briefly recall the main features exposed by the `iocl` layer that adhere to SC specifications. Additional constructs have been defined and further insights will be given in the following sections to provide an overall view of the JSCL middleware.

IOCL CORE

We now summarize the meta-model of the APIs of `iocl` core layer, reported in Figure 4.2.

Since the addressing of components strictly depends on the nature of the understanding network overlay, a **ComponentAddress** definition must be provided.

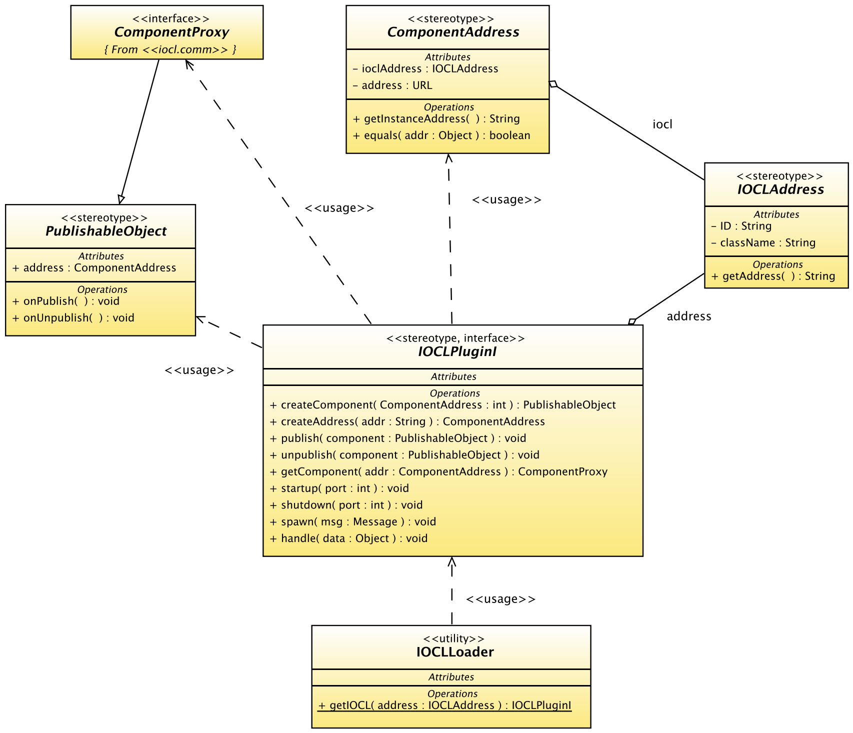

<<interface>>
ComponentProxy
{ From <<iocl.comm>> }
<<stereotype>>
ComponentAddress
Attributes
- ioclAddress : IOCLAddress
- address : URL
Operations
+ getInstanceAddress( ) : String
+ equals(addr : Object) : boolean
iocl
<<stereotype>>
IOCLAddress
Attributes
- ID : String
- className : String
Operations
+ getAddress( ) : String
<<usage>>
<<usage>>
<<stereotype>>
PublishableObject
Attributes
+ address : ComponentAddress
Operations
+ onPublish( ) : void
+ onUnpublish( ) : void
address
<<usage>>
<<stereotype, interface>>
IOCLPluginI
Attributes
Operations
+ createComponent(ComponentAddress : int) : PublishableObject
+ createAddress(addr : String) : ComponentAddress
+ publish(component : PublishableObject) : void
+ unpublish(component : PublishableObject) : void
+ getComponent(addr : ComponentAddress) : ComponentProxy
+ startup(port : int) : void
+ shutdown(port : int) : void
+ spawn(msg : Message) : void
+ handle(data : Object) : void
<<usage>>
<<utility>>
IOCLLoader
Attributes
Operations
+ getIOCL(address : IOCLAddress) : IOCLPluginI

Figure 4.2: IOCL metamodel: «iocl_core» profile

Components are identified by their address, usually an URL, in conjunction with the address of the `iocl` on which they have been published. The method *`getInstanceAddress`* provides a canonical representation of component addresses in the form *iocl@⟨compAddr⟩*. The `iocl` identifier is in the following called address prefix. Valid examples of `iocl` prefixes are *socket* to indicate components interacting via usual socket connections, as well as *mem* or *xsoap* to indicate plug-ins implemented as "in memory" or with xsoap [oCSIU] (an implementation of SOAP Web Services), respectively. Furthermore, on component addresses it is possible to singularly retrieve the `iocl` address (*`getIOCLAddress`*) or the local address representation of the component (*`getAddress`*) or to make a comparison with another address (*`equals`*).

Each `iocl` plug-in is deployed and distributed in a *Java ARchive* (**JAR**) file.

By convention, an header file, MANIFEST.MF, is utilized for declaring which package the class implementing the `iocl` plug-in interface is located in. The **IOCLAddress** keeps the association between the plug-in prefix identifier and the related implementing class. Given the address of an `iocl`, the **IOCLLoader** is able to instantiate it (*`getIOCL`*) by exploiting the reflection features.

In the following we make use of an intermediate class **IOCLRegistry**, consisting of a support utility that simplifies the translation from `iocl` aliases to their related instances (*`getIOCLByName`*). The binding among `iocl` names and their implementing plug-ins are directly retrieved from a configuration file coming with JSCL projects. Here we report a fragment of a the configuration file where are declared the `iocl` prefixes adopted and the binding with the implementing classes.

```
iocl.prefixes = xsoap, mem, socket
iocl.xsoap.classname = net.tao4ws.jscl.IOCL.XSoap.IOCLImpl
```

Consequently the access to the `iocl` plug-in can be obtained as follows.

```
IOCLRegistry.loadConfiguration("jscl.properties");
IOCLPlugin factory = IOCLRegistry.getIOCLByName ("xsoap");
```

The `iocl` plug-ins are built by implementing the **IOCLPluginI** interface. To the `iocl` is demanded the responsibility to properly build addresses (*`createAddress`*), using the prefixed form previously described, and to implement registry facilities for publishing (*`publish`*) or for revoking the publication of a component (*`unpublish`*) and for retrieving (*`getComponent`*) published components. The method *`createComponent`* is used when some code must be executed at initialization phase or when additional information, out of the scope of the middleware, must be attached to components (e.g. components must implement a particular interface defined elsewhere), though it is not mandatory. The *`startup`* and *`shutdown`* methods have the obvious meaning. Finally, the *`spawn`* method is invoked at emission of signals from the higher layers and corresponds to the SIGNAL ENVELOPE primitive of SC networks discussed in Section 3.5. This method is demanded to implement the asynchronous message delivering transparently to the `sbl` layer as previously discussed. Conversely, the method *`handle`* represents the entry point of the remote `iocl` on which is located the subscriber and is contacted during the spawn of the publisher's `iocl`. An overview of the message delivering protocol is given in Section 4.2.2.1.

The **PublishableObject** interface declares the way components are addressed on `iocl` registries (*`getAddress`*) and, when required, the triggers to fire on publication activation (*`onPublish`*) and on publication removal (*`onUnpublish`*) of components.

Iocl comm

The «iocl_comm» profile, reported in Figure 4.3, comprises the APIs needed to implement the communications among components. Once a signal is spawned into the network, the message delivering and the proper message transformations for serializing data are applied accordingly to the `iocl` prefix of the target component.

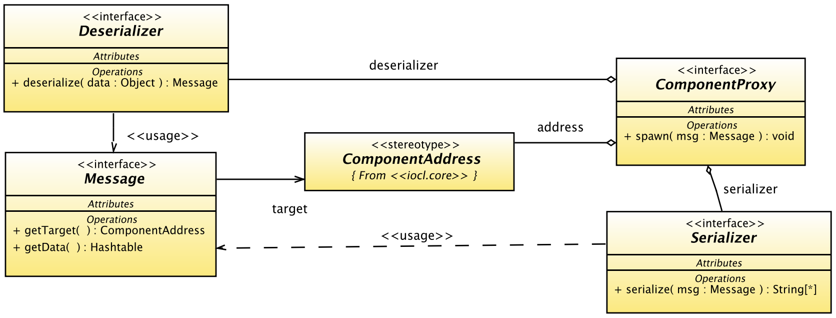


Figure 4.3: IOCL metamodel: «iocl_comm» profile

The interface **Message** declares the structure of exchanged signals. At this level two parameters are considered: the address of the target component (*`getTarget`*) and the payload (*`getData`*) containing, over than the information to retrieve the notified event, the actual carried data. The topic and the session of the spawned signal are stored under reserved keys of the data dictionary.

The `iocl` can decide to adopt proxy artifacts for remote components. In this case, the `iocl` plug-in is equipped with a class implementing the **ComponentProxy** interface. Proxies are usually employed when the `iocl` needs to move part of the message handling, for example for keeping a connection alive, or for caching the outgoing messages. The activation of a proxy is done by invoking its method *`spawn`*.

Each `iocl` plug-in comes equipped with default **Serializer** and **Deserializer** implementations, nevertheless they can customized to fit the developer needs. The serialization of a message is performed during the spawn of a message on the local `iocl`. This functionalities are invisibly accessed by the local `iocl` (or by the component proxy) in according to the target component. Conversely, at the reception of the message, the receiving `iocl` activates its deserialization and pass it to the higher architectural stacks.

4.2.2 Signal Based Layer

The Signal Based Layer (sbl) is here presented as composition of two profiles: the *i)* «sbl_data» profile, defining signals as high level representation of iocl messages, and *ii)* the «sbl_core» exhibiting the SC primitives for implementing components, flows and reactions.

Sbl data

As shown in Figure 4.4, signals are special kinds of iocl messages characterized by the notion of signal type coherently to the structured topics discussed in Section 3.5. A **Signal**, more than having a *type*, for supporting the SC notification delivering, are able to convey data. Data are accessed with the usual *getValue* and *setValue* methods. Moreover, on each signal, it is possible to retrieve the owner (*getOwner*), by means of the component that initially notified the event, and the source (*getSource*), consisting of the last sender of the notification (the last hop in a chain of successive forwards).

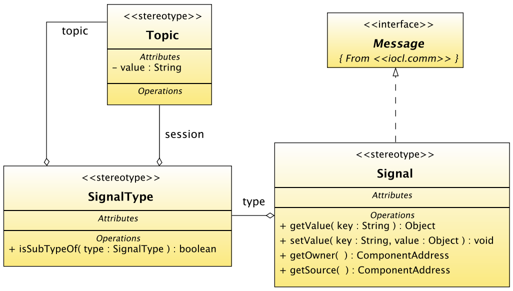


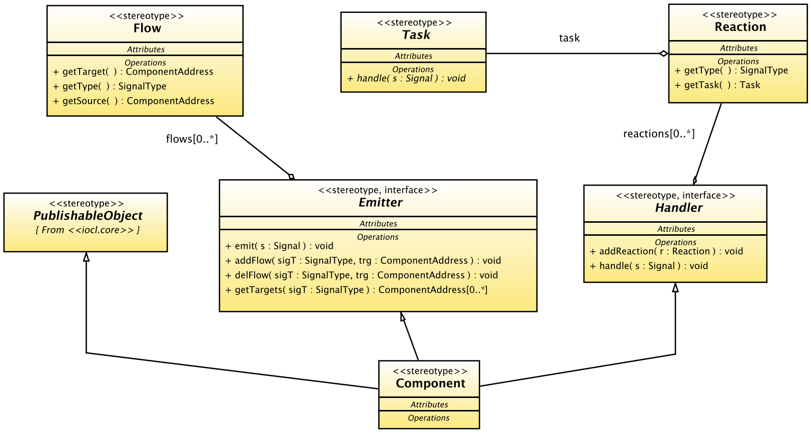
Figure 4.4: Signal Based Layer metamodel: «sbl_data» profile

The **SignalType** contains a *topic* and a *session* identifier that are accessible through the usual getter and setter mechanisms. The method *isSubTypeOf* has been introduced to enable the support of structured topics for signal types in accordance to the extensions provided in [FGST07]. More complex structures on topics, like hierarchical types, can be easily supported by re-defining this method. Reaction activation and message delivering are implemented on top of the notion of sub typing on topic structures. Signal types having the empty (*null*) session can be declared (e.g. to deal with basic SC formalism) by omitting the session parameter at instantiation phase. Besides giving the possibility to emit signals among components, we can also associate some useful parameters to them. The iocl will be responsible to properly serialize the information attached to signals and to offer to the higher level the usual getter setter mechanisms (through *getValue* and *setValue*).

SBL CORE

A **Component**, whose structure is reported in Figure 4.5, consists of a *publishable object* defined in terms of *handler* and *emitter* parts.

The **Handler** characterizes the ability of components to declare the reactions (*addReaction*), while the **Emitter** exhibits the primitives to create links to other components (*addFlow*) and to emit signals to them (*emit*). The emission will happen regardless of the component (if any) that will actually receive the emitted signals. There might even be no target component connected; the middleware will correctly deliver signals transparently to the signal emitter. Moreover, signals addressed to components that have not yet been published, become pending on the network until they become reachable.



<<stereotype>>
Flow
Attributes
Operations
+ getTarget( ) : ComponentAddress
+ getType( ) : SignalType
+ getSource( ) : ComponentAddress
<<stereotype>>
Task
Attributes
Operations
+ handle(s : Signal) : void
task
<<stereotype>>
Reaction
Attributes
Operations
+ getType( ) : SignalType
+ getTask( ) : Task
flows[0..*]
reactions[0..*]
<<stereotype>>
PublishableObject
{ From <<iocl.core>> }
<<stereotype, interface>>
Emitter
Attributes
Operations
+ emit(s : Signal) : void
+ addFlow(sigT : SignalType, trg : ComponentAddress) : void
+ delFlow(sigT : SignalType, trg : ComponentAddress) : void
+ getTargets(sigT : SignalType) : ComponentAddress[0..*]
<<stereotype, interface>>
Handler
Attributes
Operations
+ addReaction(r : Reaction) : void
+ handle(s : Signal) : void
<<stereotype>>
Component
Attributes
Operations

Figure 4.5: Signal Based Layer metamodel: «sbl_core» profile

A **Flow** is defined as a relation between two components on a signal type. Even though the SC specification considers only the topic part of the signal type, this generalized form allows to support coordination policies that relies on types differently structured (e.g. by enabling the subscriptions related to sessions). Additionally, components support the removal of flows (*delFlow*) inherently to the ideas exposed in Section 3.5.1.3. Signals pending on the network targeted to components, for which the subscription has been revoked, are not affected by the flow removal operation, namely they remain pending on the network.

A **Reaction** is built by specifying the signal type, triggering its activation, and the **Task** to activate upon the reception of a signal. Reactions are classified into *check* or *lambda* depending on the signal type parameter. Signal types having no session parameter specified are considered to have an empty session. In accordance to the SC specification, reactions having an unbound (*null*) session are considered *lambda* and the session binding will happen at activation phase. As usual they will be activated regardless of the session carried with signals and are persistent. Conversely, *check* reactions trigger on a well defined session and, once activated, are removed from the component interface.

The handler exposes the `handle` functionality for implementing the counterpart of the message spawning that is used by the middleware to activate the task responsible to handle the received signal. The `iocl` is responsible to retrieve, at the reception of a signal, the component reaction to activate and to run, in a thread space, its handling method by passing to it the signal instance. The handling task may act on the internal state of the components and on the parameters carried with the received signal. Having reactions encapsulated into components, the related tasks can access all the capabilities offered by their containers (*signal emission*, *reaction update*, etc.).

4.2.2.1 Message Delivering Protocol

In order to give a flavor of how the network abstraction has been implemented in our middleware, we present a sketch of the signal delivery protocol. This protocol is displayed in Figure 4.6.

We can identify two network partitions and group their interactions in three distinct phases. The two network partitions, $Host1$ and $Host2$, symmetrically define three main actors: (i) a component S_1 (resp. S_2), that acts as publisher (resp. subscriber), (ii) the `sbl` architectural stack, that implements the primitives of components, and, finally, (iii) the underlying `iocl` plug-in acting as network artifact.

Once a component S_1 raises a new event, the `sbl` retrieves the set of targets *trg* subscribed for the topic of the outgoing signal, analogously to the *flow projection* operation defined in the Signal Calculus. Suppose *trg* to be composed by the only component named S_2. The `iocl` identifier is extracted from the target address (Step 1). The respective local instance of `iocl` is informed of the request of implementing a message spawning to the remote `iocl`. As a consequence, the serialization of the message will be performed in conformance with the remote data representation. The message delivering is demanded to $iocl_L$ and the control

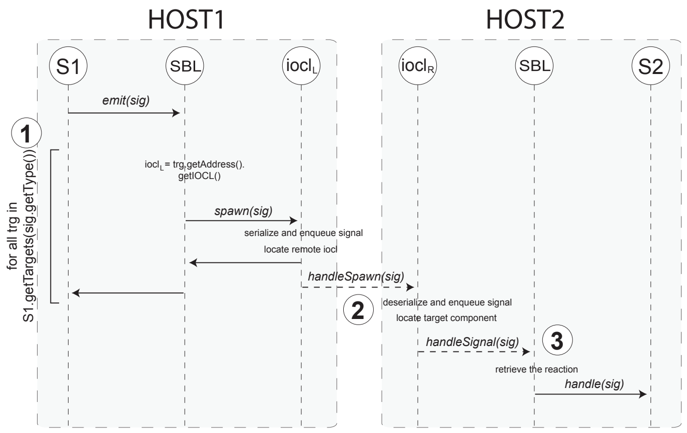


Figure 4.6: Signal delivery protocol

flow returned to the emitter.

Asynchronously, the `iocl` contacts the remote counterpart informing it that a new signal is present in the network (STEP 2). Once the message has been received, the remote `iocl` service $iocl_R$ performs the data deserialization and forwards the signal to the component $S2$ (STEP 3).

The actions presented in Figure 4.6 give an outline of the protocol adopted for implementing the communications among distributed components. The set of steps in (STEP 2) can be (partially) demanded to the **ComponentProxy** provided by the $iocl_L$. For example, here, the proxy can be adopted to keep a reference to the remote `iocl` and can be demanded to retrieve the remote component (the handler) and, in the case it is temporary busy or not available, to locally store the signals.

4.3 A RUNNING EXAMPLE: THE ALARM SYSTEM

We take into account the alarm system example presented in Chapter 3 to clarify how networks are implemented in JSCL providing some snippets of code providing an outline of how the SC ideas have been reflected at implementation level. We just focus on the core of the language in the spirit of the formalism defined in the calculus. Additional primitives declared inside JSCL will be presented in further sections.

The example may also result useful to give a mere intuition of how SC principles are reflected in the programming API. Some fragments of the JSCL coding for the *Door* agent are reported here and analogous considerations can be done for the other components. The initial interface configuration for components is boxed inside the `jscl_init` method, invoked at construction phase. We introduce, for simplicity, two dictionaries, *Global* and *Local* that store the component addresses and the topic names declared globally in the choreography or locally to each component, respectively. The values are accessed in the usual manner. The *Local* class is defined inside the same file of the component and assumed statically initialized at design time, as reported in Code 4.1. In particular, LINES 6-7 show the pattern for creating addresses and LINE 8 the one for creating new topics.

```
 1 class Locals extends HashMap {
 2   static Locals instance = new Locals();
 3   static{
 4     instance.putAddress(
 5      "dopened",
 6      IOCLRegistry.getIOCLByName("xsoap").
 7        createAddress("http://www.tao4ws.net/slocal/dopened"));
 8     instance.putTopic("lopened", new Topic("local_opened"));
 9   }
10 }
```

Code 4.1: Alarm system in JSCL: local names

The initialization consists on the definition of flows and reactions on the component interface. Flows are initialized by using the code shown in Code 4.2. Two

```
1 this.addFlow (
2   new Flow (
3     this.getAddress(),
4     new SignalType(Locals.getTopic("lopened")),
5     Locals.getAddress("dopened")));
```

Code 4.2: Alarm system in JSCL: adding flows

parameters are passed to the `addFlow` method. The signal type (LINE 4) having topic *lopened* and no session. In LINE 5, the address of the *dopened* component is retrieved by the local dictionary and passed as parameter. Notice that both names are "restricted" to the scope of the current component as previously discussed in Section 3.5.2.

The code reported in Code 4.3 appends a reaction for the topic *opened* to the component instance (line 3). The related task that must be executed at its activation is defined in block 5-9. Essentially, once received a signal, the affected component delivers a new signal with the locally defined topic *lopened* (line 7) and the same session of the entering signal (line 8).

```
 1 this.addReaction(
 2   new Reaction (
 3     new SignalType (Globals.getTopic("opened")),
 4     new Task (){
 5       public void handle (Signal s){
 6         emit (new Signal(new SignalType(
 7           Locals.getTopic("lopened"),
 8           s.getType().getSession())));
 9       }
10     }
11   )
12 );
```

Code 4.3: JSCL alarm coding: adding a reaction

Finally, components are bound to networks by requiring a publication on the `iocl` registries. Each JSCL project comes equipped with a configuration file that contains information useful to retrieve from the `iocl` meta-name the related plug-in to activate. All the information regarding the configuration are accessed through apposite utilities. For example, the class **IOCLRegistry** encapsulates the `iocl` name bindings offering the facilities from registering components. The code for publishing a component becomes:

```
IOCLRegistry.getIOCLByName("xsoap").publish (component);
```

4.4 Illustrating the network flexibility

Modern distributed systems demand not only heterogeneity but also a higher degree of adaptability and the Service Oriented Architectures provide evidence of this issue. In the SOA approach, applications are developed by coordinating the behavior of autonomous components distributed over an overlay network. At the best of our knowledge, current research and implementation efforts devoted to design and to implement middleware for coordinating distributed services (see ORC [Mis04], BPEL [Spe], WS-CDL [W3Cc] and SIENA [CRW98] to cite a few), have focused on overlay networks based on a public addressing schema, namely the *address* of each service is directly visible and reachable from any part of the network. Indeed, very few approaches address coordination of services over overlay networks where services reside on host without a public address or are hosted behind a firewall hiding their addresses. Other modern distributed systems raise similar demands with respect to the visibility of addresses. Illustrative examples are peer-to-peer networks. Coping with these issues is therefore a challenging task for the SOA paradigm.

We attempt to explore the features of the SOA approach within computing environments without a public addressing schema where visibility of service addresses is not always guaranteed. Here, we describe the adaptation of `iocl` architectural layer and its implementation for supporting service coordination where identification of services endpoints is more structured while preserving, at the same time, independence from the underlying network technologies. Hence, the assumption on public visibility of all services is relaxed. We discuss design and implementation issues analyzing the impact of our proposal over current technologies and network solutions. The experimental results and the formal model that as driven our implementation choices have been reported in [FGS06a].

4.4.1 Gateways

To offer the possibility to handle interactions among components that may reside on networks that do not guarantee public addressing, a special version of `iocl` networks has been defined, based on a *two addressing schema* methodology. Such `iocl` network exploits the concept of gateways. The registering facilities are demanded to gateways that act as bridges among several network infrastructures. Gateways, on their turn, can be published on the local machine, by invoking, as usual, the method *`iocl.publish`*, allowing each component to deliver messages to them using the `iocl` specific protocol. In order to simplify the development, the `iocl` extends the notion of proxies to gateways. Hence, internally, the sub-layer can access a remote gateway by using the method *`iocl.getGateway`* that instantiates a proxy that locally exposes the gateway remote interface.

A `Gateway` intermediates the message passing among component giving to them *public visibility*. The component registration to a gateway is obtained by invoking the method *`register`*. Internally, this operation involves the creation of a *virtual channel* between the gateway and the component. Incoming signals are delivered to the *owned* components through the method *`handle`* of the proxy supplied by the `iocl` that involves the communication on the *virtual channel* previously created.

Two kinds of addresses are defined. The gateways are identified by public addressing schema in accordance to the standard `iocl` addressing mechanisms.

Hence, they are bound to an `iocl` address. Components, instead, become visible to the outside through intermediate gateways. Their addresses are declared as couples, in the following represented in the form $G[a]$, to assert that the component addressed a is registered on gateway G. The publication of services happens, as usual, by specifying the address to which the component will be bound, regardless the knowledge of the existence of gateways. The registration to the gateway, under the assumption it has already been published and made visible, will be automatically performed during the publication of the component.

A sample code for binding a component address to a gateway for a further publication on an `iocl` registry is the following:

```
GWAddress g_addr =
  IOCLRegistry.getIOCLByName("xsoap").
    createGWAddress(
      "http://www.tao4ws.net/gateways/G"));
ComponentAddress c_addr =
  IOCLRegistry.getIOCLByName("rhttp").
    createCAddress(g_addr,
      "http://www.tao4ws.net/services/G/a"));
// publish the service
IOCLRegistry.getIOCLByName("rhttp").publish(component)
```

Two kinds of address types are defined: the **GWAddress** and the **ComponentAddress**. The structure of public addressing schema previously described is reflected in the former class definition. The latter instead, is obtained by binding a component address defined on the *rhttp* `iocl`, to a public visible gateway address (*g_addr*). When publishing a component, the `iocl` itself, is responsible to internally implement the registration on a gateway according to the information retrieved by the **ComponentAddress** structure. The publish and register facilities, and the addressing schemata, are internally implemented by an ad-hoc `iocl` plug-in called *rhttp*, whose implementation details are reported in Section 4.4.4.

4.4.2 Implementation Overview

In this section, we outline the implementation strategies adopted for the Jscl extension supporting the two level addressing. Services are supposed to be always private implying that, to be *visible* to the outside, they need to make a registration to, at least, a gateway. Gateways become the unique public visible entities in the global network. Having several `iocl` plugins, one for each network overlay, gateways need to make available on the `iocls` they are interested to operate in. In the following we only deal with two kinds of overlay networks: SOAP with standard

HTTP binding and SOAP with the binding proposed in 4.4.3. Depending on the protocol used to identify a component or a gateway, JSCL instantiate the proper `iocl` (e.g. to *rhttp* corresponds the `iocl` with multipart, etc.). Communication from a gateway to a public service hosted on the same "domain" can be obtained through HTTP binding or through more scalable and efficient ad-hoc solutions (e.g. JMS [SUNa]). Each `iocl` defines its gateway and component *proxy* structures that map the local invocation to communication primitives according to the related network.

4.4.3 X-Mixed-Replace SOAP Binding

We propose an alternative SOAP binding for HTTP 1.1 to supply an envelope transport mechanism for services that cannot open local *tcp* ports (e.g. firewalled applications), or that are executed on machines without public address (e.g. internet applications) or that are hosted in an environment that disallows socket management (e.g. Ajax and Comet applications inside a Web Browser). The proposed binding is based on the `x-mixed-replace` [Net99] mimetype and is structured as follows.

1. The service opens a HTTP 1.1 connection to a potential requester and performs a GET request specifying the information needed for the publication.

2. The requester sends back a response having mimetype `x-mixed-replace`. Usually, this mimetype informs a client that the server will send a stream of multiple versions of the same document. The client and the server must keep opened the HTTP connection, until the server terminates to deliver the stream.

3. When the requester wants to send a SOAP request to a previously published service, it sends a SOAP envelope over the active HTTP connection, as a new version of the multipart document.

4. When a new version of the multipart document is received, the SOAP envelope is extracted and the local service is invoked.

As we will see in section 4.4.4, the first and second steps are performed by the *gateway.register* method, which creates the *virtual channel* between the gateway and the component, and the third and forth steps are performed for routing signals from the gateway to the component.

4.4.4 JSCL implementation outline

Here, we outline the implementation strategies adopted for implementing the JSCL extension supporting orchestration with a two level addressing schema.
By using a *UML-like sequence diagram* notation, in Figure 4.7, are shown the steps performed by JSCL to implement the component registration to a gateway (*block* 1) and the signal exchanging between two components (*blocks* 2, 3 and 4). In the following we will use the notation P^S_X to represent proxies for an entity S (a component or a gateway) communicating through the network via protocol X. Analogously, A^S_X represents an address of the entity S over the network via protocol X.

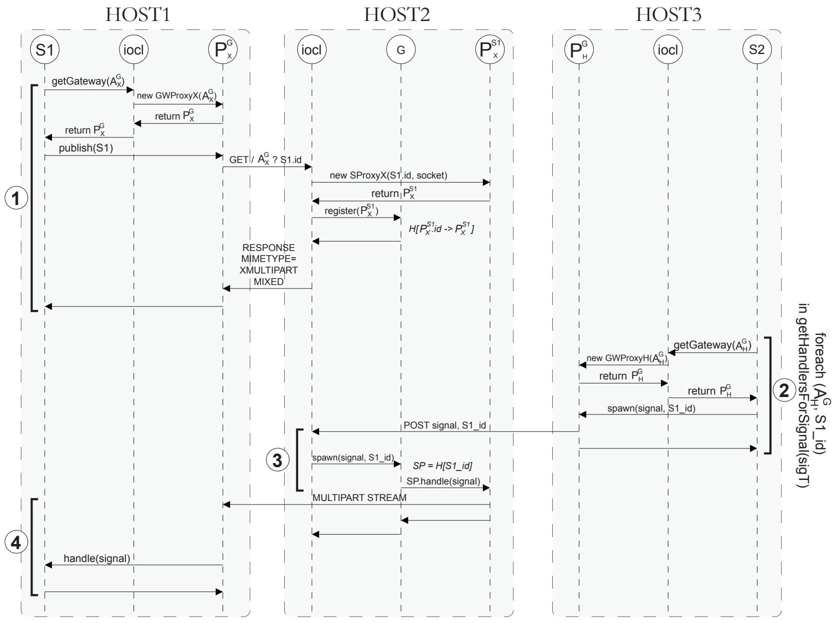


Figure 4.7: Registration and signal emission protocol

The *block* 1, defined in Figure 4.7, describes the steps performed by the component S_1, hosted on $Host_1$, to activate a registration on the gateway G, located on $Host_2$. S_1 demands to the `iocl` to create a proxy P^G_X for the gateway G, having address A^G_X, which will be encapsulated into the proxy instance. The registration method is invoked on the proxy which makes an HTTP request, specifying the encapsulated gateway address and the component identifier (see Step 1 in section 4.4.3). The request is received by the `iocl` on the $Host_2$ that creates the local proxy P_X^{S1} for the component requester. Notice that the connection (*socket*) established with P_X^G and the component identifier $S_1.id$ are stored into the component proxy. The gateway stores, into the table H, the association between the component identifier and the proxy bound to it. Finally, the `iocl` sends back an HTTP response to declare that further messages will be send on that stream (see Step 2 of section 4.4.3).

As result of a signal emission, the component S_2 retrieves the set of component link descriptors of the form $(A_H^G, S1_{id})$. For each component, S_2 requests a proxy for the intermediate gateway (P_H^G), then invokes its method `spawn` (*block* 2). The gateway proxy sends a HTTP Post request, containing the signal and the target component identifier, using standard SOAP HTTP binding. At the reception of the message, the `iocl` on $Host_2$ retrieves the proper gateway and invokes its method `spawn`. The gateway retrieves, from the table H, the proxy for the target component that has been created at the registration phase (*block* 3). The component proxy forwards the signal through the multipart stream using the previously encapsulated connection with P_X^G (see Step 3 of section 4.4.3). Finally, in *block* 4, the gateway proxy retrieves the locally registered component and demands to it the signal handling (see Step 4 of section 4.4.3).

4.5 Additional programming facilities

Besides the basic mechanisms for implementing SC, the JSCL middleware offers some additional features which are described in this section. In order to simplify the designing and the development of complex coordination patterns, we introduce special kinds of "pre-coded" components called *logical ports*. Such artifacts permit to define first-order logic based applications offering, moreover, immediate support for work-flow diagrams, as discussed in Section 4.5.1.

Additionally, we discuss the possibility to inhibit the delivering of signals through flows by annotating them with constraints that are evaluated before performing the emission to the corresponding subscribers. These constructs, called *flow guards*, trigger the emission of signals through a flow. If the logical condition expressed on the guard is not satisfied, the notification will be ignored and not externally visible. Flow guards are reported in Section 4.5.2.

4.5.1 Logical Ports

A value-added of JSCL is constituted by the introduction of *logical ports* which permit to define first-order logic based applications offering, moreover, a direct correspondence with flow-chart diagrams. The logical ports represent a generalized form of components with a pre-built behavior. They receive as input parameters corresponding to the signal topics to associate to the *boolean* representation. JSCL provides AND, OR and NOT logical ports.

A simple, yet illustrative, example is in Figure 4.8, where the AND gate synchronizes the flows originating from components *b* and *c*. Namely, when both *b* and *c* emit an *Ok* signal, the AND gate propagates an *Ok* signal to component *d*, otherwise an *Err* is emitted for *e*.

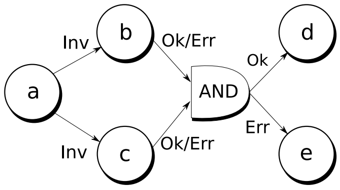


Figure 4.8: Joining two components

The strategy adopted here mainly differs from the synchronization mechanism, whose SC coding has been presented in Section 3.5.1.1, for the multiple exit flows of the AND component. In the first instance, at implementation level, JSCL takes advantage from the host language to express the internal logics of the logical port differently from the SC formalism where the state configurations were coded through the proper usage of specialized reactions. An intuition of the logical port coding is given in Section 4.5.1.1.

4.5.1.1 A sketch of logical ports

In Code 4.4 we report an sketch of how an AND port can be programmed in JSCL.

```java
1   protected class AndPort extends Task {
2     public void handle (Signal s){
3       Topic session = s.getType().getSession();
4       syncronized(session) {
5         if (!state.containsKey(session.getValue()))
6           state.set(session.getValue(), s.getTopic());
7         else if (state.get(session).equals(sigTrue) &&
8                  s.getTopic().equals(sigTrue)) {
9           emit(s);
10          state.remove(session);
11        }
12        else {
13          s.setTopic(sigFalse);
14          emit(s);
15          state.remove(session);
16        }
17      }
18    }
19  }
```

Code 4.4: Logical Ports: AND behavior

We focus on the internal logics of component to implement the required functionality. A special kind of dictionary *state* is internally used to store the received signals, grouping them by sessions, and is demanded to implement the boolean evaluation on topics. The reported sketch considers the simple case that just two components can be connected in input. The first insertion for a session is simply stored into the dictionary (LINES 5-6). The state will be TRUE if and only if all the connected components have sent signals having the TRUE topic (LINES 7-11), otherwise the state will be considered false (LINES 12-16).

4.5.1.2 Logical ports: API

Logical ports (whose API are in Figure 4.9) are instantiated by declaring two kinds of topics corresponding to the boolean TRUE and FALSE values and are able to handle only signals corresponding to signal topics associated to the boolean values. They come equipped with predefined behaviors (and reactions) specialized for the accepted topics. The installation of additional reactions is forbidden. Analogously the `addFlow` method has been overloaded to force link creation to the only boolean corresponding topics. This modification implicitly restricts the use of primitives for signals firing, so that the signals fired for different topics will be ignored.

To apply the boolean relations to the outgoing signals of several components it is sufficient to invoke the method `filter`, meaning that on the output links of notifiers will be applied a filtering. The ports can have several components connected both in input and in output.

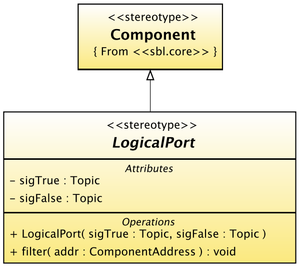


Figure 4.9: Profile «sbl_ext»: logical ports

Basically, the logical ports have been introduced for simplifying the design of service composition. These components permit the developer to easily program connections and, furthermore, they can be used in conjunction with the other primitives in order to express more complex patterns. For example, referring to the *workflow patterns* presented in [WvdADtH03b], we can describe the *parallel split* and the *synchronization* between two components *a* and *b* by adding an *and* port filtering their outgoings. In the same manner the *or* port can be adopted for implementing the *exclusive choice*.

The following example outlines the flexibility of logical ports to model distributed work-flows.

Example 4.5.1 *Consider a service for searching into several on-line book shops an item and to buy it by accessing to some payment service. The participants involved in the interactions are a buyer, search engines, a basket handler (shared by all the services), a bank service and a credit card manager. Figure 4.10 shows a control flow diagram for the application (the data flow is omitted for simplicity).*

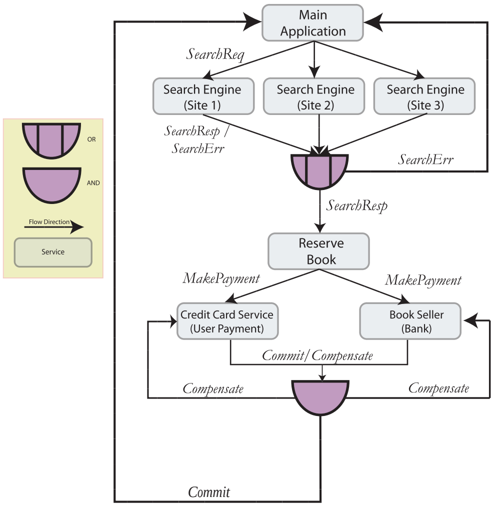


Figure 4.10: Search and buy books overview

Let us now illustrate the protocol of the Example 4.5.1. Initially the user specify the search query which is forwarded to the search engines dislocated on several nodes of the network. Each service makes a local search for the book specified by the user and responds by emitting a signal containing the results of its search. The Or port collects the results coming from the search engines and signals to the chosen book store to make the reservation for it[1]. The last step of the application is the payment, it consists of a charging operation from the credit card account of the user to the bank account of the chosen book store. The bank and credit card manager services execute their operations in parallel and, *if and only if both of them* reach a positive result the whole transaction is considered completed and so can *commit*, otherwise the And port will forward a signal to the

[1] In order to make as simple as possible the example, we suppose that the user makes only one search each time he uses the service.

previous stages in order to communicate a failure.
The signals which are exchanged among the services are respectively: *SearchReq*, containing the parameters useful for making the search (e.g. the *book title*), *SearchResp* sent by the search engines to communicate they have found the requested book. The book identifier will be conveyed through the signal data. If during its execution a search engine fails (e.g. book not found), it will emit a *SearchErr* signal. The Or port collects the signals coming from the search engines and, if at least one of them sends a *SearchResp* signal, the computation can continue otherwise the whole transaction is terminated and a *SearchErr* is forwarded to the *Search & Buy Books* service in order to inform the user that the book has not been found. Once the *Reserve Book* service receives the search response, it signals to the chosen book store to make a reservation on the item specified in the request and forward a payment request to the next two services.
The *bank* and *credit card manager* services receive all the needed informations to retrieve respectively the book store bank account and the buyer credit card number. Once received these informations, they can start their executions making the proper payment. The signals they emit are *Commit* or *Compensate*. If both of them emit a *Commit* signal the whole transaction can be considered successful executed, otherwise a rollback signal is forwarded backward to inform the previous stages to *undo* their modifications. A snippet of the Jscl code for the Or port is given in Code 4.5. In lines 4-6 there is the declaration of the filtered components and the remaining blocks (lines 7-10 and lines 11-14) connect the outputs for the true and false corresponding topics, respectively.

```
1  public SearchOrPort extends OrPort {
2    // ...
3    protected void jscl_init(){
4      this.filter (Globals.getAddress("search1"));
5      this.filter (Globals.getAddress("search2"));
6      this.filter (Globals.getAddress("search3"));
7      this.addFlow (
8        new Flow (this.getAddress(),
9          this.sigTrue,
10         Globals.getAddress("reserveBook")));
11     this.addFlow (
12       new Flow (this.getAddress(),
13         this.sigFalse,
14         Globals.getAddress("entry"));
15   }
16 }
```

Code 4.5: Search & Buy Books: Or port

4.5.2 Guarded Flows

The *guards* give the possibility to attach, during the creation of a flow, a constraint that must be evaluated before sending the signal on it. The signal is sent through the flows providing that the evaluation of the guard is *true*, otherwise it is removed from the system leaving no trace. Guards are built by extending the abstract class **Guard** and implementing the method `processSignal`. The API of guarded flows are given in Figure 4.11.

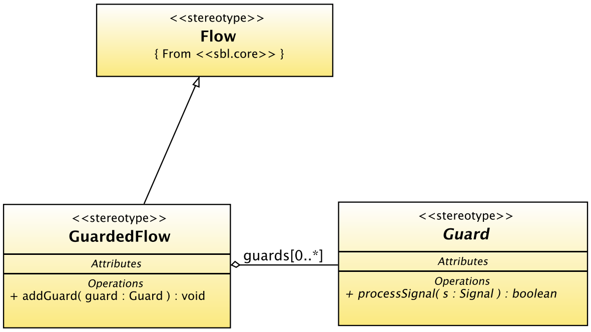


Figure 4.11: Profile «sbl_ext»: guarded flows

Example 4.5.2 *The code below, attaches on a flow a guard that checks that no "creditCard" parameter is present in the data part of the signal before sending it.*

```
flow.addGuard (
  new Guard() {
    public boolean processSignal (Signal s){
      if (s.getValue("creditCard") == null)
        return true;
      return false;
    }
  };
);
```

Often, this construct results useful when the delivering to a particular subscriber must be inhibited by the occurrence of a particular configuration of the raised event or of the internal state of the component itself. Notice that the guards discriminate the emission on a flow declared for a well defined topic and targeted to a single subscriber so that, for the same topic, the other subscriptions are not affected. Similar results can be obtained by applying the flow removal primitive and implementing the logics internally to components.

4.5.3 The dark side of serializers

An distinguishing feature of WSs stack relies in its modularity. Due to the adoption of open and platform independent specification languages for defining all the stacks of the architecture, the WSs can be easily extended to support functionalities. For example by introducing higher architectural levels that extend the structure of the messages to support additional features. A typical example is the WS-Security [OAS06] that states:

> “ *This specification is intended to provide a flexible set of mechanisms that can be used to construct a range of security protocols; in other words this specification intentionally does not describe explicit fixed security protocols.* ”

Namely the specification establishes a policy on the structure of SOAP messages declaring a common understandable way to exchange encrypted messages among services. Briefly, it suggests a pattern for storing secure messages and conversely for retrieving them, by declaring the path they are located (e.g. in the header). This strategy does not impacts the lower layers of the WSs platform. Services that does not support this feature will simply ignore the encrypted part and access the remaining data.

Similarly, the adoption of JSCL serializers allows to increment the flexibility of the framework to support new capabilities. For example, proper serializers (resp. deserializers) can be equipped with an `iocl` plug-in so that they will encrypt (resp. decrypt) the messages before sending them on the network (resp. to the handler component).

Other fields of applications is the coexistence with services non JSCL compliant. For example the activation of a SOAP based web services that is not implemented on JSCL based networks can be achieved by applying XSLT translating rules that translate signals in SOAP messages. Actually this goal can be achieved by adopting intermediate wrappers that exploit the `iocl` serialization features and act as proxy in behalf of other JSCL components. So that the request to an external service is implemented by transforming the signals in the proper message and once received the response (if any) it is converted into a signal so that can be communicated to other participants. In the future we are investigating for giving an intermediate structure on signals that allows the automatic translation into different domains (e.g. for retrieving the direct encoding from WSDL to signals).

4.6 Concluding remarks on Jscl

Jscl proposes a framework which provides the minimal primitives for creating components which interact by notifying event occurrences.

The event notification pattern has been reinterpreted in order to build a fully distributed solution differently from great part of analogous tools proposed (e.g. SIENA [Sof05, CRW98]). The approach proposed in Jscl differs from them since it does not rely on an engine that rules the execution of a composed process. Instead the conversational primitives are translated into coordination patterns among services over a middleware establishing a sort of choreography (as defined in ws-cdl [W3Cc]) among involved services.

Jscl primitives combine aspects related to the service definition and the ones strictly related to their composition and communication, presenting a complete framework useful for describing all the aspect relevant for defining business processes.

More complex scenarios can be built upon Jscl to treat more complex aspects. In [BFM+05] it was presented a specialization of Jscl focused on describing the transactional aspects related to the long running transactions (LRTs) in the spirit of Naïve Sagas [BMM05], a specialization of Sagas [GMS87]. This example better remarks the flexibility of Jscl proposed as generic middleware for describing patterns which for their nature can be described in a signal passing style.

The middleware proposed, in fact, do not want to replace existing tools and languages for *business processes* but, instead, it suggests a general purpose *signal based* framework. As an example, Jscl can be adopted for developing a BPEL engine or moreover for mapping Coloured Petri Nets [BRR87] constructs.

As counterpart, Jscl uses statefull services to keep tracks of inter-services connections and force the developing of several copies of the same service if used in different instances of the transaction. This constraint is essential if we want to avoid centralized orchestration.

Chapter 5

Programming Environment

The initial version of JSCL *was intended to provide a run-time support for* SC *networks; later, it has been extended with a set of facilities oriented to yield a complete and easy-to-use programming environment. In fact,* JSCL *has been equipped with a user-friendly interface in the form of a set of Eclipse plug-ins that offer a graphical and a textual representation of* SC *networks.*

These functionalities offer two different perspectives of the network. The graphical representation presents a global view of the choreography by considering the components and their interconnections, without detailing their internal logics. The textual notation offers a closer view of components allowing designers to focus on the behavioral aspects.

In a model driven metaphor, the aspects treated at these different levels of abstraction share a common meta-model so that it is easy to pass from a level to another and to use the resulting target model to automatically generate the runnable JSCL *code.*

The implementation choices and the capabilities offered by our programming framework are exposed in this chapter.

5.1 EVENT BASED SERVICE COORDINATION

In Chapter 4 we have presented JSCL as a run-time support for SC networks. Now we present the *event based service coordination* (ESC) framework that provides programming facilities for designing and implementing JSCL services from a higher level of abstraction. The framework is constituted by different plug-ins that, from several perspectives, offer to the designer all the constructs present in the JSCL middleware. The main advantages of this approach stands in its simplicity of use and in the separation of the concepts at different levels of design.

From the one hand, a graphical toolkit is used to design the components and their interconnections. At this level components expose their interfaces in terms of flows and reactions without detailing their internal behaviors. On the other hand, a textual representation can be used to specify how components are internally implemented by declaring their behavioral properties. Finally, the resulting model can be compiled and executed on top of JSCL middleware.

5.1.1 JSCL Graphical Notation

The JSCL graphical notation captures the sequence of activities via the description of the network topology of involved components.

The JSCL graphical notation mainly differs from BPMN on the way interactions are modeled. BPMN directly defines the flow of messages exchanged among components using a flow mechanism. JSCL, instead, defines the correlation among services so that the message sequences depend of the component internal behaviors and are not directly caught at design time.

In JSCL, components are implemented as services and can represent BPMN *pools* on the assumption that services are used to implement the participants. JSCL reactions can be interpreted as BPMN activities, because they are the handlers of messages and implement *atomic business logics*. Similarly to BPMN activities, which can be depicted into BPMN pools, reactions must be encapsulated inside JSCL components. Links in BPMN define the activity sequence, while they define publishers/subscribers relationships in JSCL. Notice that a JSCL diagram cannot contain events that are raised inside components. Moreover, links are artifacts for SC flows and specify to which components deliver events according to their topics. Even though flows are machinery for subscriptions and do not have a channel correspondence, abstractly they represent the communication strategies of EN; for this reason we will refer to flows as links or channels.

The graphical editor gives the possibility to directly use logical ports (defined in Section 4.5.1) that are here exposed as components with a pre-defined internal logic and a fixed set of reactions. This constructs are similar to the notion of BPMN gateways.

In Figure 5.1 we show a simple protocol with three components a, b, c. In our notation, the outermost boxes (labelled with components' names) represent components. Inside each components, at the bottom, we find a circle that represents the anchor point for outgoing flows, represented, in turn, as arrows tagged with the conveyed topic (t_1, t_2, etc.). Finally, the yellow boxes nested inside components represent the reactions installed on a component at initialization phase.

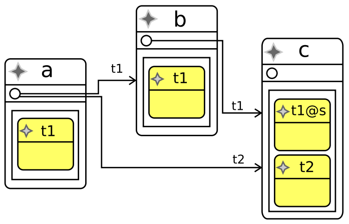


Figure 5.1: JSCL diagram of the example

The signature of reactions is given accordingly to the SC specification so that reactions labeled as $t_1@s$ represent check reactions, conversely the ones simply labeled with the topic name (e.g. t_2) represent lambda reactions.

Notice that, since reactions can be installed and modified at run-time, just the initial state is depicted. The initial configuration of the example shown in Figure 5.1, is the following: component b is subscribed for events of topic t_1 notified by a, while c is subscribed on topic t_2 notified by a and on topic t_1 notified by b.

Notice that some relevant details can be derived by the scenario depicted in the example. Initially, the component a has an installed lambda reaction for topic t_1 that is apparently unused. Since flows, akin reactions, can be dynamically programmed, links could be attached at run-time to such reaction, so that no assumptions can be derived from the initial definition of the network.

Moreover, the network can be thought of as a sub-network acting in the regards of a more structured network, as a single service hiding its internal structure and exposing, for example, the only entry points of a to the outside. This idea is at the base of the concept of compositionality of services.

Additionally, high level constructs, like the logical ports, are presented as generic components for which the designer is demanded to generate the suitable SCL coding and to install the proper constraints.

The plug-in has been implemented using GMF [Ecla], that provides a generative infrastructure for developing editors based on EMF [Eclb], as model representation, and on GEF [Eclc], as graphical support.

5.1.2 Signal Core Language

The graphical editor plug-in is paired with the textual representation given as a Domain Specific Language (DSL), called SCL after *Signal Core Language*. SCL provides a compact view of components distribution and enables to implement components by detailing their reactions. The language has been implemented as a textual plug-in for the Eclipse environment supporting code completion, error checking and code generation (some of which are described in Section 5.1.3).

The main elements are the *components*, described in terms of “reactive” software modules declaring the class of events they are interested to and the way they react at the occurrence of events. Components are defined inside a *network*.

A typical network is represented as follows:

```
 1 restricted: s1,s2;
 2 global: t1, t2, t3;
 3 component a {
 4  local: lt1, lt2;
 5  flows: [t1->a], [lt1->b];
 6  knows: s1,b;
 7  reaction lambda (t1@ws){
 8   addFlow ([ws->b]);
 9   addReaction (
10    reaction check (lt1@lt2){
11     emit (t1@lt1);
12    }
13   );
14   nop;
15   do {/*behavior*/} or {/*behavior*/}
16   split {/*behavior*/} || {/*behavior*/}
17   with (nlt1){/*behavior*/}
18   skip;
19  }
20 }
21 protected component b {
22  knows: s1;
23  main {
24    // behavior
25  }
26 }
```

The example shows a network composed by two components *a* and *b*, defined in the LINES 3-20 and 21-26, respectively. Topic names can be declared as **global** and shared among components as defined in LINE 2. Moreover, topics can be declared in a private scope of a component using the primitive **local** (LINE 4) or during the computation through the primitive **with** (LINE 17). However topic names can be declared restricted to more than one component so that the primitive **restricted** (LINE 1) is used. Similarly, component names can be declared restricted by tagging components with the **protected** clause (LINE 21). Components can insert restricted names inside their scope (with the exception of the names declared with *local* and *with* clauses) by using the **knows** primitive (LINES 6 and 22).

Components are uniquely identified by a name (e.g. *a*) and declared with the **component** keyword. Components contain a set of local topics, a set of flows and a set of reactions declaring the topics that can be handled and the tasks to perform.

The couple $t_1@t_2$ represents SC signal types $(t_1 \diamond t_2)$, where t_1 is the event topic and t_2 the work-flow session in which it has been declared. Sessions identifiers and topics are freely interchangeable, as show in LINES 7-8 where the session *ws* received by the lambda reaction is afterwards used as topic name for connecting the component *a* to the component *b*. Components declare their entry points by installing reactions. Two kinds of reactions can be defined: the **reaction lambda** (see LINE 7) that is activated for a topic regardless its related session, and **reaction check** (see LINE 10) that triggers within a specific session. As consequence, the session identifier used in lambda reactions must be a "fresh" identifier, while the one of check reactions must be already defined. **Flows** and **reactions** can be defined at initialization phase (LINES 5 and 7, respectively) or added at run-time if required (LINES 8 and 9-13, respectively).

The computational steps described inside reactions, declare their *behaviors*. The basic behavioral instructions are: **emit** (LINE 11), used to send out notification for an occurred event, **addFlow** and **addReaction** previously described, and **nop** (LINE 14) to indicate an instruction externally defined through host language instructions that do not interfere the coordination patterns (e.g. the access to the database). The **skip** (LINE 18) represents the empty action (the SC silent action). Furthermore, behaviors can be composed in sequence (using, as usual the semicolon) or with **do-or** (LINE 15) and **split** (LINE 16) constructs. The former constructs is used to implement the non deterministic execution of two branches. The latter construct allows the parallel composition of two behavioral activities.

Notice that component *b* declares a **main** block (LINES 23-25). It is used to define the initial behavior of a component.

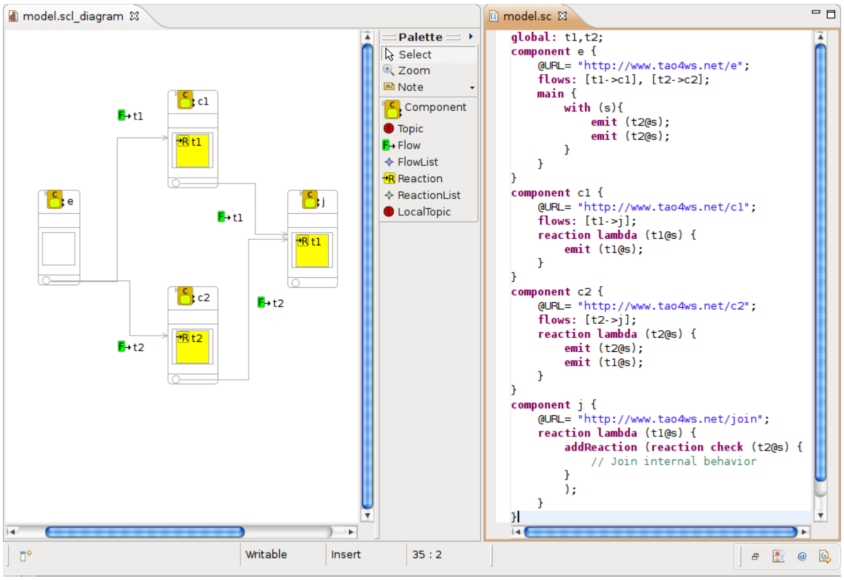
The textual editor has been implemented by using OpenArchitectureWare (oAW) [OAW], a modular MDA/MDD generator framework.

5.1.3 Basic Facilities

The ESC platform presents a set of facilities that we group into three categories: (*i*) constraint checking, (*ii*) code generation and (*iii*) model synchronization. We now detail these aspects.

Constraint checking The graphical editor fixes some constraints on the structure of the networks and is responsible to check at designing time whether they are respected. Syntactical checks verify that the names and the addresses of the components are unique and there are no multiple instances of flows starting from a component having the same target and topic. Logical ports are considered valid if there are at least two components attached in input. More precisely, for the AND port it suffices that on the *true* corresponding topic there are the entering links, since the synchronization is applied just for this kind of topic, while the *false* branch is not blocking. The correct usage of names inside components and flows are checked by the editor that evaluates the correct scope of each element. Moreover, on logical ports the installation of new reactions is avoided and the outgoing flows can correspond uniquely to the boolean respective topics.

Model synchronization The meta-models managed by the designer and the textual editor are automatically synchronized so that modifications to a model are immediately reflected in the other one. In the same way, errors (or warnings) arisen during checking are automatically reported in both models. In Figure 5.2 are given the two different views of the join example discussed in Section 3.5.1.1.



```
global: t1,t2;
component e {
    @URL= "http://www.tao4ws.net/e";
    flows: [t1->c1], [t2->c2];
    main {
        with (s){
            emit (t2@s);
            emit (t2@s);
        }
    }
}
component c1 {
    @URL= "http://www.tao4ws.net/c1";
    flows: [t1->j];
    reaction lambda (t1@s) {
        emit (t1@s);
    }
}
component c2 {
    @URL= "http://www.tao4ws.net/c2";
    flows: [t2->j];
    reaction lambda (t2@s) {
        emit (t2@s);
        emit (t1@s);
    }
}
component j {
    @URL= "http://www.tao4ws.net/join";
    reaction lambda (t1@s) {
        addReaction (reaction check (t2@s) {
            // Join internal behavior
        }
        );
    }
}
```

Figure 5.2: ESC design: the join pattern

Property	Description
ID	An (unique) alias for the component.
Address	The `iocl` representation of the address (URL).
Classname	The class name to use for generating code.
Package	The package in which the generated classes will be stored.
Iocl plugin	The `iocl` network adapter to use for publication.
Generate	If the code must be generated (bool)
Publish	If the generated component must be published(bool)

Table 5.1: SCL DIAGRAM: component properties

Code generation At the end of the modeling phase, it is possible to generate the Java source code. The code generation creates components and a network orchestration artifact. According with the component and network properties given in Table 5.1 and in Table 5.2, the source code is generated as follows:

- An *orchestrator class* is generated. This class has a static *main* method that creates the coordination. The orchestrator instantiates each component and, if their *publish* property is true, it publishes them to the proper addresses, otherwise it obtains a remote proxy for them.
- For each component having *generate* property enabled, a component class is created. Component flows are declared accordingly to the designed network. The component property file is generated with the local and global names as discussed in Section 4.3.
- For each reaction a proper class is created. This class implements the handler for messages of the specified type. To implement the internal *business logic*, in correspondence of the **`nop`** instructions, the body of the handle method must be filled.

Each generated class allows for each method to specify the *generated* annotation. Source code with this annotation will be overridden by further code generations. To prevent this behavior the annotation must be disabled to make the environment aware that the source code has been specialized. The environment supports a simple navigation mechanism: starting from the diagram view, the contextual menu *Edit code* opens in the default Eclipse Java editor the corresponding source file.

Property	Description
Address	The `iocl` representation of the address (URL).
Classname	The class name to use for generating code.
Package	The package in which the generated classes will be stored.
Collector	The class name to use for generating code.
Iocl plugin	The `iocl` network adapter to use for publication.

Table 5.2: SCL DIAGRAM: network properties

5.1.4 Synchronizing behaviors

Here we present a general purpose construct that will be used in the late as synchronization machinery for describing the logical port coding in SCL. The construct is in the following referred to with the behavioral primitive **synchronized** that declares a critical section that regulates the flow of messages coming from the producer component. Moreover, with the aim to give a "non invasive" construct, in the means that it does not comprises the intended behavior of the original producer component, we make use of intermediate components that act as proxies for the message producer. Two additional components QUEUE and SYNC are introduced as shown in Figure 5.3. With MSGS we represent the queue of messages coming from the producer. The CONSUMER is encapsulated inside a sub-network and represents the only externally visible component.

The links reported in Figure 5.3 represent the flow configurations of components.

```
restricted n,rdy,done;

protected component queue {
  knows: rdy;
  flows: [msg->sync],[rdy->sync];
  reaction lambda (msg@s) {
    emit (rdy@s);
    emit (msg@s);
  }
}
```

Code 5.1: Synchronizing behaviors in SCL: queue

As detailed in Code 5.1. The QUEUE receives from the producer the messages. Once a message *msg*@*s* is received, QUEUE informs SYNC that a new message in the session *s* is ready to be consumed (line 7) and promptly forwards the message to it (line 8).

```
 1  protected component sync {
 2    knows: n,done,rdy;
 3    flows: [msg->consumer];
 4    reaction lambda (rdy@s) { // from queue
 5      addReaction (reaction check (done@n){ // from consumer
 6        addReaction ( reaction check (msg@s) {
 7          emit (msg@s);
 8        } );
 9      });
10    }
11  }
```

Code 5.2: Synchronizing behaviors in Scl: sync

As shown in Code 5.2, once sync receives notification that a new message is ready (line 4), it waits until the consumer is ready to receive the new message (line 5) and consequently consumes the message sent by queue (line 6) and forward it to the consumer (line 7).

```
 1  component consumer {
 2    knows: queue, sync, done,n;
 3    flows: [done->sync],[msg->...];
 4    reaction lambda (msg@s) {
 5      // Synchronized behavior
 6      addReaction (reaction check (msg@s){...}
 7      // End of sync
 8      emit (done@n);
 9    }
10    main {
11      emit (done@n);
12    }
13  }
```

Code 5.3: Synchronizing behaviors in Scl: consumer

Finally, the Scl representation for the consumer is given in Code 5.3. The block 5-7 is considered a critical section since the installation of the check reaction must be installed before a new message is sent to the consumer. Once executed the critical section, the sync agent in informed that the consumer is ready to receive a new message (line 8). The topic *done* is internally used by consumer to inhibit the emission of signals from the SYNC component. At initial phase the consumer is able to receive a message from the system as given in BLOCK 10-12.

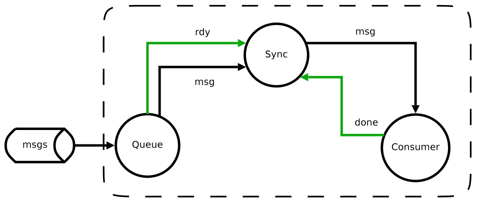


Figure 5.3: Synchronizing reactions installation

5.1.5 Logical Ports in SCL

At design level is offered the possibility to make use of JSCL logical ports. Furthermore, to keep coherence with the textual representation, once a new input port is created, the designer converts the high level construct in concrete SCL primitives. For example, the SCL representation of a binary AND port is reported in Code 5.4.

```
 1 global: tt,ff;
 2 component And {
 3   reaction lambda (tt@s){
 4     synchronized {
 5       addReaction (reaction check (tt@s){
 6         emit(tt@s);
 7       });
 8     }
 9   }
10   reaction lambda (ff@s){
11     synchronized {
12       addReaction (reaction check (ff@s){
13         skip;
14       });
15     }
16     emit(ff@s);
17   }
18 }
```

Code 5.4: Binary AND port in SCL

The boolean corresponding topics are globally declared (LINE 1). The AND component is initialized with two reactions for handling *tt* and *ff* signals (resp. BLOCKS 3-9 and 10-17). The former block reinstalls a further reaction (BLOCK 5-7) that triggers the next reception of a *tt* signal valid for the session *s*. The latter block instead immediately forwards the *ff* signal since no synchronization is required in this case and installs a reaction (BLOCK 12-14) for consuming the next occurrence of the *ff* signal for the same session.

5.2 A CASE STUDY

In this section we illustrate how the SENSORIA car repair scenario [WCG+06] can be developed by using the ESC framework. The formal description and the considerations that lay around this coding have been exposed in [FGST08]. Here we only use it to show a possible use of the JSCL framework to model the following problem:

You rented a car to visit some of the beautiful savage landscapes you like most and you ever dreamed of visiting. Everything is fine, but the car has a fault...in the middle of nowhere! What if the car could arrange for a quick rescue? Probably what you would like to have is a track towing you to a close garage and another car to proceed your trip.

5.2.1 The car repair scenario

A car manufacturer offers a service that, once a user's car breaks down, the system attempts to locate a garage, a tow truck and a rental car service so that the car is towed to the garage and repaired meanwhile the car owner may continue his travel.

The interdependencies between the service bookings make it necessary to have an orchestration with compensations. Before any service lookup is made, the credit card is charged with a security amount. Before looking for a tow truck, a garage must be found as it poses additional constraints to the candidate tow trucks. If finding a tow truck fails, the garage appointment must be revoked. If renting a car succeeds and finding either a tow truck or a garage appointment fails, the car rental must be redirected to the broken down car's actual location. If the car rental fails, it should not affect the tow truck and garage appointment.

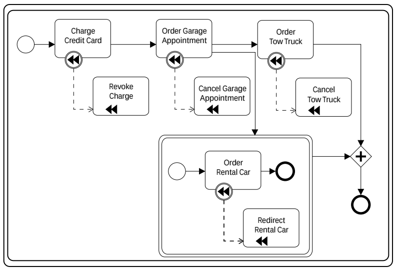


Figure 5.4: Car repair scenario: the BPMN model

The BPMN model of this scenario is presented in Figure 5.4. Notice that the model exploits the transactional and compensation facilities of BPMN$_{tr}$ and that the car rental service is a sub-transaction, since it does not affect other activities.

5.2.2 Designing the Car Repair Scenario

To model the car repair scenario as a JSCL diagram we assume that each participant is represented by a JSCL component. We use two types of signals: *forward* and *rollback*. The first event type is used to inform a component that all the previous activities have been completed, while the second one is used to inform a component that an exception has been occurred, and that all following activities have completed their compensation. Each component has two reactions: one that handles forward signals, executing the corresponding main activity, and the other one that handles rollback signals, executing the corresponding compensation if the main activity has been previously completed without errors.

In Figure 5.5 the JSCL model of the car repair scenario is presented. In the environment, we use the black color for forward links (that are oriented from the left to the right) and red one for the other ones (oriented in the opposite verse). The model contains two instances of the AND logical port defined in Section 5.1.5. The two logical ports are differently instantiated. Namely, the And_1 applies a synchronization on the rollbacks (the backward flow before the GARAGE compensation) so that it is initialized with *rollback* topic assigned to the *true* value. Conversely, the And_2 considers the *forward* topic corresponding to the *true* topic. Moreover, the *And* ports have two roles: *i*) to synchronize the forward flow, and *ii*) to execute the compensation of RENTALCAR if both the ORDERTOWTRUCK fails and the RENTALCAR main activity has been successfully completed.

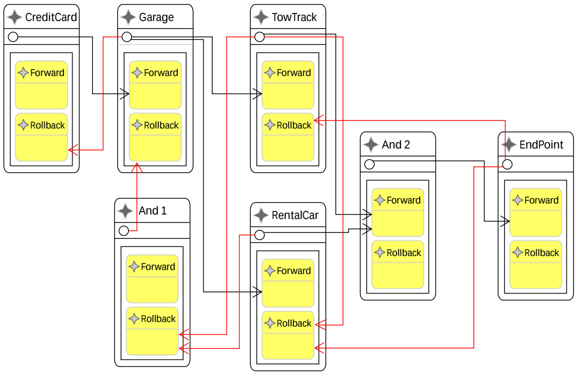


Figure 5.5: Car repair scenario the graphical JSCL representation

Notice that, in this scenario, the ORDERTOWTRUCK compensation is not executed if the RENTALCAR fails. The ENDPOINT component represents the BPMN final state and allows to reuse the obtained composition as a sub-network inside other designs. This component simply forwards the received signals to the components externally connected.

Chapter 6

Experimenting Long Running Transactions

In this section, in order to highlight the flexibility of the SC *proposed methodology to deal with practical aspects that are relevant in the SOA scenario, we present a coding of* BPMN$_{tr}$ *constructs. In particular, we utilize Sagas specification trying to establish a connection among the theoretical model presented in [BMM05] and the* SC *design model. The choice falls to* Sagas *since it naturally abstracts away from low level computations and communication patterns, while highlighting the composition structure of transactional processes. We map high level transactional primitives into concrete orchestration patterns by observing the events that are raised inside Sagas activities.*

The exposed ideas have been adopted in a model transformation tool that starting from a BPMN$_{tr}$ *construct gives the corresponding* SCL *code that is subsequently mapped on its paired graphical notation.*

6.1 From BPMN to SC (informally)

The success of BPMN consists in the possibility to describe the work-flow of business processes and their transactional activities from a global point of view. Hence, the resulting model abstracts from the distribution of processes, from the communication policies and from the underlay network technologies. As counterpart, BPMN neglects distribution aspects that are instead captured in SC network model. Notably, SC components correspond to services artifacts so that the computations are enclosed inside component boundaries while for BPMN activities it can happen, at implementation level, that several activities correspond to different stages of computations performed on the same service, or conversely, that single work units are spawned, at run-time on several services that implements part of the required tasks.

The Sagas offer a suitable framework for detailing the formal semantics of BPMN$_{tr}$ (the transactional subset of BPMN reported in Section 2.5.1) and we take advantage from its specification to provide an outline of how such components can be automatically coded in SC language.

We discuss the implementation choices in terms of topics of exchanged signals and internal communications among components.

The basic units of BPMN$_{tr}$ are *compensable activities*, namely pairs of main activities and compensations that can be composed through *sequence* or *parallel* blocks or can be enclosed (isolated) inside *transactional boundaries*. The corresponding Sagas constructs are: *step*, *sequence*, *parallel* and *saga*.

Referring to the Sagas semantics, reported in Section 2.5.2, our focus is not on the executed activities but rather on their results.

For this reason, differently from Sagas, we distinguish two groups of possible observable events by means of *i)* the results obtained during the execution of internal steps and *ii)* the results observed by the outstanding processes. Two distinguished event topics f and r are globally used (after Sagas terminology for forward and rollback flow, respectively) to signal successful termination of processes. The observation of internal activities is instead modeled through local topic names as clarified in the further sections.

6.1.1 Compensable activity

Compensable activities of BPMN$_{tr}$ can be interpreted in SC by applying the semantical specification given in Sagas for STEPS. Figure 6.1 gives a pictorial intuition of the coding.

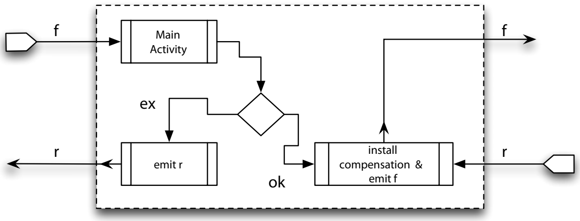


Figure 6.1: Internal view of SC compensable activities

Initially, signals of type f trigger the execution of the main activity, hereafter referred as *Task*, and install the reactions to manage its continuation. Indeed, *Task* will eventually emit either an *ok* or an *ex* signal that are two distinguished topics restricted to the transactional component depending on the whether its termination has been successful or not. In the former case, the compensation is installed and the f signal is propagated outside the component, otherwise a r signal is emitted to the previous stages of the transaction. Notice that, as required by sagas, rollback signals can be consumed only by components that successfully executed their main activity (and therefore installed their compensations).

$$
\begin{array}{lcl}
TC(a, A, B, prev, next) & = & (\nu ok, ex)\Big(a[0]^{f\,\lambda s \gtrdot A \,|\, \mathtt{rupd}(R_{res})}_{ok \rightsquigarrow a \oplus ex \rightsquigarrow a \oplus f \rightsquigarrow next \oplus r \rightsquigarrow prev}\Big) \\
R_{res} & = & ok \diamond s \gtrdot \Big(\mathtt{rupd}(r \diamond s \gtrdot B) | \mathtt{out}(f \diamond s)\Big) \otimes \\
 & & ex \diamond s \gtrdot \mathtt{out}(r \diamond s)
\end{array}
$$

Code 6.1: SC coding of BPMN$_{tr}$: compensable activity

A BPMN$_{tr}$ compensable activity is expressed by the SC component reported in Code 6.1, where $Task = \epsilon.\mathtt{out}(ok \diamond s) + \epsilon.\mathtt{out}(ex \diamond s)$ [1] and $Comp = \epsilon.\mathtt{out}(r \diamond s)$, respectively, represent the main activity and the compensation of the transactional component, and *next* and *prev* represent the forward and the backward flows.

To give a flavor of how SC primitives are reflected in the SCL meta-model we approach the definition of compensable activities in both formalisms. The corresponding SCL representation is reported in Section 6.2.1.

6.1.2 Sequence

The sequential composition of two transactional components is obtained by suitably connecting the forward and rollback flows as shown in Figure 6.2.

The SC term modeling the BPMN$_{tr}$ sequence design of Figure 2.5(a) is graphically represented in Figure 6.2 where a (resp. b) embeds $Task1$ and $Comp1$ (resp. $Task2$ and $Comp2$). We use solid (resp. dashed) arrows to represent the forward (resp. backward) flow.

Notice that the compositionality is obtained by simply rearranging their interconnections. Namely, given two transactional entities $Comp_a$ and $Comp_b$ of the form:

$$
\begin{array}{lcl}
Comp_a & \triangleq & TC(a, A_a, B_a, prev_a, -) \\
Comp_a & \triangleq & TC(b, A_b, B_b, -, next_b)
\end{array}
$$

[1] Notice that SC is not directly equipped with non-deterministic choice. However non-determinism can be easily encoded exploiting the non-deterministic activation of reactions as discussed in Section 3.3.

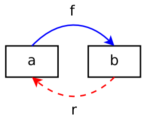


Figure 6.2: SC sequential composition

the corresponding SC sequential composition is given by:

$$Comp_a; Comp_b \triangleq TC(a, A_a, B_a, prev_a, b) \mid TC(b, A_b, B_b, a, next_b)$$

The component a on the left side of the composition is connected for forward flows to the component b on its right and conversely for the backward flows b is connected to a.

6.1.3 Parallel composition

Parallel composition of components requires two auxiliary components called *dispatcher* and *collector* to model the fork and join entry points. Dispatchers are responsible to collect notifications of the forward flow (signals of topic f) and redirect them to the parallel components. Symmetrically, dispatchers bounce rollback signals of topic r when the backward flow is executed. Analogously, collectors propagates forward and backward flows by sending the signals of topic f or r as appropriate. Figure 6.3 yields a pictorial representation of how the forward and backward flows of the dispatcher d and collector c of parallel components a and b are coordinated using the f and r signals. Notice that a and b have rollback flows connecting each other; in fact, the semantics of saga prescribes that, when the main activity of a parallel component fails, the other components must be notified and start their compensations.

A further topic n is internally used for implementing the synchronization. Abstractly, this topic is used to implement a "private channel" through which d communicates to the collector c the new received session s to inform that a new parallel branch is going to start. Subsequently, the signal is forwarded to the internal parallel stages to initiate their tasks. The collector is responsible to receive through the channel n the work-flow session identifier s and to install the suitable reactions for consuming signals sent by the parallel stages (the synchronization protocol is implemented coherently to the ideas exposed in Section 3.5.1.1). In Section 6.2.2 we report the respective Scl coding for parallel composition applied to the case study introduced in Section 5.2.

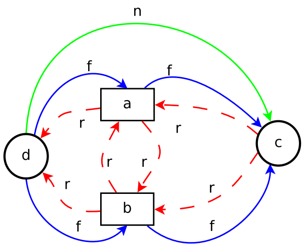


Figure 6.3: SC parallel composition

When needed, the usage of name communicating capabilities can be adopted for implementing "session nesting" technique. For example the entry point of a composed network (e.g. *d*) can assign a new session name to the signals before delivering them to the internal stages so that the exit point (e.g. the collector *c*) is informed of the renamed session. The internal components (e.g. *a* and *b*) will act on restricted work-flow session arising conflicts with any other external concurrent components out of the actual work-flow session.

6.1.4 Transactional enclosure

The intended meaning of saga construct is that its internal failure does not affect other activities. For this reason, regardless the outgoings of transactional activity *a* the collector will receive a forward (*f*) signal. If from the outside *c* receives a rollback, the component *a* must be informed and activate its compensation. Two cases are possible: *i) a* has previously successful terminated, so it has a compensation installed *ii) a* internally failed and no compensations are needed.

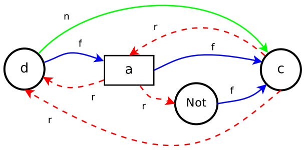


Figure 6.4: SC transactional enclosure

Under the assumption that *d* has to consume two instances of *r* signals before activating the backward flow while *c*, for the same session, consumes only a *f* signal (and ignores the further instances of *f*), the corresponding SC network is given in Figure 6.4.

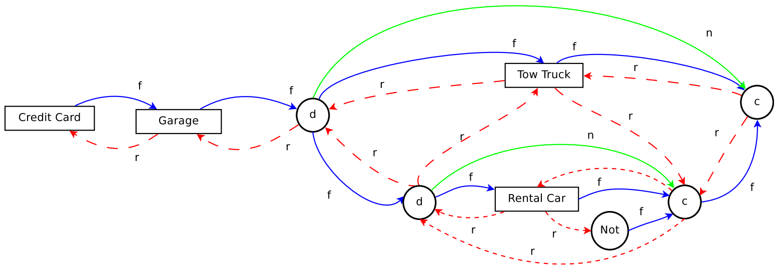


Figure 6.5: The generated SCL network

Similarly to the parallel encoding previously exposed, the topic n is used from d to inform c that a new work-flow instance has been initiated so that the latter component can install the proper check reactions to consume two distinct instances of f signals coming from a.

By applying the graphical representation of SC coding for BPMN$_{tr}$ constructs, the case study introduced in Section 5.2 can be pictorially defined as in Figure 6.5.

6.2 BPMN TO SCL MODEL TRANSFORMATION

The framework permits to transform the platform independent BPMN$_{tr}$ model to the platform specific SCL model. We describe this transformation exploiting the case study explained section 5.2. We recall that our framework supports a subset of the BPMN specification, that has been formalized by the Saga process algebra. The SCL implementation of transactional behaviors exploits two public names, fw and rb, respectively for *forward* and *rollback* signals. Forward signals propagate the successful completion from an activity to the next ones in the work-flow. Backward signals are emitted on failures to trigger compensations. In the first step the model transformation generates an SCL component for every BPMN atomic process (aka an activity and the corresponding compensation).

Each step can generate *glue* components and change the flows of the components, however the behavior of components generated in the previous steps cannot be altered.

This permits to transform a BPMN process to an SCL network independently by the context, and reuse it as building block just changing its connections (SCL flows).

6.2.1 BPMN atomic process

In Code 6.2 we report the SCL coding of transactional activity GARAGE.

```
 1 component garage {
 2  local: ok, ex;
 3  flows: [(ok->garage), (ex->garage)];
 4  reaction lambda (fw@s) {
 5   nop;
 6   split {
 7     do {emit <ok@s>;} or {emit <ex@s>;}
 8   } || {
 9     addReaction (reaction check (ok@s) {
10       split {
11         emit <fw@s>;
12       }||{
13         addReaction (reaction check (rb@s) {
14           nop;
15           emit <rb@s>;
16         });
17       }
18     });
19   } || {
20     addReaction (reaction check (ex@s) {
21       emit <rb@s>;
22     });
23   }
24 }
```

Code 6.2: BPMN to SCL transformation: compensable activity

The component declares two private topics, *ok* and *ex*, (LINE 2) that will be used to verify the termination of the corresponding main activity. Notice that all event raised by the component having these topics will be delivered to the component itself (LINE 3). Since these topics are restricted in the scope of the component, we refer them as local topics. Initially the component can react only to fw signals (BLOCK 4-23). Upon the reaction of forward signals, the component bounds (receives) the session identifier s and execute the task prescribed by the BPMN$_{tr}$ main activity (LINE 5). We do not implement explicitly this activities, but assume that, if successfully terminating they issue an *ok* signal, otherwise they issue an *ex* signal. Concurrently with the main activity the component installs the reactions to check its termination (BLOCKS 9- 18 and 20-22). A successful activity (LINE 6) propagates fw signal to the next stages of the work-flow (LINE 11) and a check reaction for a further rollback notification is installed (LINE 13-16). When a *rb* signal for the session s is received, the compensation is executed and the rollback signal is propagated to previous stages (LINE 15). If the execution of the activity fails (LINE 20), the handler simply starts the backward flow, raising a rollback signal (LINE 21). Since the transformation of an atomic task generates only one SCL component, this component is both the entry point and the exit point of the generated network. Notice that the generated component has only flow to itself, since it is generated independently by the context.

6.2.2 Parallel composition

The parallel composition of two BPMN_{tr} processes is transformed by generating the networks corresponding to the two processes and two other components, referred to dispatcher and collector. Now we report the SCL code for the collector and dispatcher generated to implement the parallel composition of the TowTruck and RentalCar services.

```
 1  component dispatcherPar {
 2    flows: [fw->TowTruck],[fw->RentalCar],
 3           [rb->Garage],[n->collectorPar];
 4    reaction lambda (fw@s) {
 5      split {
 6        emit (fw@s);
 7      } || {
 8        emit (n@s);
 9      } || {
10        addReaction (reaction check (rb@s) {
11          addReaction (reaction check (rb@s) {
12            emit (rb@s);
13          });
14        });
15      }
16    }
17  }
```

Code 6.3: BPMN to SCL transformation: parallel dispatcher

The dispatcher (c.f. Code 6.3) represents the entry point of the parallel branch. Basically, it activates the forward flow of next components, and synchronizes their backward flows. Upon reactions to forward signals (LINE 4), the collector emits two events: one having topic fw (LINE 6) and the other one having topic n (LINE 8). The former event is delivered to the components representing the parallel activities. The latter event is delivered to the collector, informing it of the received session that will be later used by it to implement its synchronization.

Concurrently, the collector activates its the synchronization mechanism by installing two nested reactions for the topic *rb* in the work-flow session *s* (LINES 10 and 11). When the synchronization of the backward flow takes place, the emitter backwardly forwards the rollback signal (LINE 12).

```
 1 component collectorPar {
 2  flows: [rb->TowTruck],[rb->RentalCar],[fw->...];
 3  reaction lambda (n@s) {
 4     addReaction check (fw@s) {
 5       addReaction check (fw@s) {
 6         split {
 7           emit <fw@s>;
 8         } || {
 9           addReaction check (rb@s) {
10             emit <rb@s>;
11           }
12         }
13       }
14     }
15   }
16 }
```

Code 6.4: BPMN to SCL transformation: parallel collector

Similarly, the collector component (in Code 6.4) is responsible to implement the synchronization mechanism for the forward flows (LINES 4 and 5) and to activate the backward flows of the parallel components when a *rb* signal is received (BLOCK 9-11). Once both the internal components have sent their forward messages, the collector sends out a fw signal (LINE 7). Notice that the collector exploits a *n* signal to get information about the session *s* of the work-flow (LINE 3). After the generation of the new components, the flows of the two networks are updated (the flow for fw in LINE 2). Moreover the backward flow is suitable connected to the internal parallel components as given in LINE 2). The Figure 6.3 depicts the flows generated for two nested parallel branches of the proposed scenario. The dispatcher and the collector components represent the entry and exit point of the parallel component, respectively.

6.2.3 BPMN sub-transaction

A BPMN sub-transaction is compiled as a SCL network that does not effects the computation of tasks performed out of the sub-transaction itself. The transformation generates a dispatcher and a collector. The generated SCL code for the sub-transaction containing the RentalCar component is provided by three internal components according to the schema given in Figure 6.4.

```
 1  component dispatcherTrans {
 2   flows [n->collectorTrans],[fw->RentalCar];
 3   reaction lambda (fw@s) {
 4     emit (n@s);
 5     emit (fw@s);
 6     addReaction (reaction check (rb@s){
 7       addReaction (reaction check (rb@s){
 8         emit (rb@s);
 9       });
10     });
11   }
12  }
```

Code 6.5: bpmn to Scl transformation: saga dispatcher

The dispatcherTrans (c.f. Code 6.5) receives from the external activities the forward signals (line 3), informs the collectorTrans that a new transactional session has been initiated (line 4), redirects the forward signal to the RentalCar (line 5) and installs the rollback handler for the current session (block 6-10). Notice that, the rollback will be sent out (line 8) after the reception of two *rb* notifications.

```
component Not {
 flows [fw->c];
 reaction lambda (rb@s) {
   emit <fw@s>;
 }
}
```

Code 6.6: bpmn to Scl transformation: saga not

The Not port has the obvious meaning, it inverts the topic from *rb* to fw, without altering the session, as given in Code 6.6.

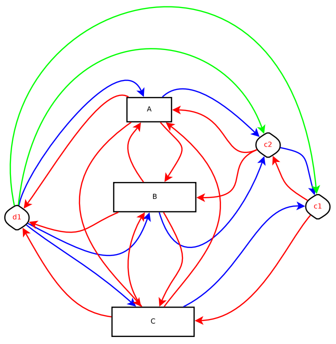
The CollectorTrans (c.f. Code 6.7) waits until the dispatcher communicates the new working session (line 3). Consequently, it installs the handler for the fw notifications coming from the RentalCar (block 4-9). Once received the fw it is delivered outside (line 5) and an handler for the rollback coming from the outside is installed (block 6-8).

```
component collectorTrans {
 flows: [fw->TowTrack],[rb->RentalCar],[rb->dispatcherTrans];
 reaction lambda (n@s) {
   addReaction( reaction check  (fw@s) {
     emit(fw@s);
     addReaction (reaction check (rb@s) {
       emit (rb@s);
     });
   });
 }
}
```

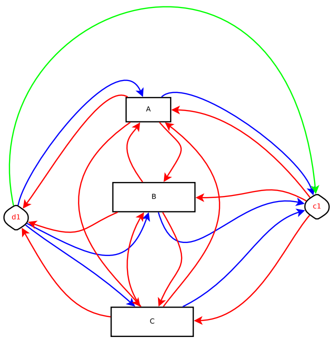
Code 6.7: BPMN to SCL transformation: saga collector

6.3 SCL MODEL REFACTORING

The BPMN$_{tr}$ translation in SCL networks is given by the transformation roles discussed before. Anyway, if for example we consider the parallel branches, two additional components are introduced, the collector and the dispatcher. The further parallel composition with a third component produces the network in Figure 6.6(a). Sometimes, it could be useful to merge the additional components as shown in Figure 6.6(b).



(a) Original Coding



(b) Merged Collectors

Figure 6.6: SCL refactoring: merging parallel collector

These refactoring roles can be provided in the graphical environment under the assumption that they respect the initial intended behavior. The refactoring roles on BPMN$_{tr}$ coding have been studied in [Gua09] and as future work we plan to integrate the results in our programming platform.

6.4 Concluding remarks

The SC-JSCL framework has been design to support the specification, the implementation and verification of coordination policies for services oriented applications. Our main goal is to provide general facilities to implement high-level languages for service oriented architectures (e.g. BPEL4WS [AGK+03], BPML [OMG02],WS-CDL [W3Cc]). The strict interplay between SC and JSCL permits to drive and verify implementation of such languages.

A number of approaches have been introduced to provide the formal foundations of standards for service orchestrations and service choreographies. The SC-JSCL framework differs from these approaches (COWS [LPT07], Global Calculus [CHY07], λ_{req} [BDFZ07] ORC [Mis04], SCC [BBC+06], SOCK [GLG+06] to cite a few), since it focus on a lower level of abstraction, merging the theoretical formalization with the implementation requirements. Indeed, the emphasis in SC-JSCL is just on designing general facilities to program coordination patterns on services by exploiting the notion of event notification. The EN mechanism features abstraction from the communication layer providing the capabilities for implementing multi-cast communications.

There are a number of directions that we are pursuing for the future development of our framework. In [FGST07], we introduced an algebraic structure over topics. This allows us to implement complex coordination logics directly inside the signal type. Moreover, this provides the foundational description of BPMN-like gateways. We intend to investigate this issue in order to design a BPMN work-flow engine based on the SC/JSCL framework. Furthermore, we plan to extend the SC/JSCL framework with facilities for reasoning and proving properties of coordination policies. On the one hand, we are extending the compilation facilities so to generate both the source JSCL code and the SC specification out of the JSCL graphical notation. On the other hand, we plan to integrate in our environment toolkits that provide verification and analysis capabilities for Java programs and other semantic checker (e.g. bisimulation and model checkers) for the SC specification.

Bibliography

[AACP04] Marco Aiello, Mikio Aoyama, Francisco Curbera, and Mike P. Papazoglou, editors. *Service-Oriented Computing - ICSOC 2004, Second International Conference, New York, NY, USA, November 15-19, 2004, Proceedings*. ACM, 2004.

[AB05] Lucia Acciai and Michele Boreale. Xpi: A typed process calculus for xml messaging. In Martin Steffen and Gianluigi Zavattaro, editors, *FMOODS*, volume 3535 of *Lecture Notes in Computer Science*, pages 47–66. Springer, 2005.

[AGK$^+$03] Tony Andrews, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. Business process execution language for web services. Version 1.1. BEA, IBM, Microsoft, SAP AG and Siebel Systems, May 2003.

[All98] Paul Allen. A practical framework for applying uml. In Jean Bézivin and Pierre-Alain Muller, editors, *UML*, volume 1618 of *Lecture Notes in Computer Science*, pages 419–433. Springer, 1998.

[BB92] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, April 1992.

[BBC$^+$06] Michele Boreale, Roberto Bruni, Luís Caires, Rocco De Nicola, Ivan Lanese, Michele Loreti, Francisco Martins, Ugo Montanari, António Ravara, Davide Sangiorgi, Vasco Thudichum Vasconcelos, and Gianluigi Zavattaro. Scc: A service centered calculus. In *WS-FM*, volume 4184 of *Lecture Notes in Computer Science*, pages 38–57. Springer, 2006.

[BBF$^+$05] Roberto Bruni, Michael J. Butler, Carla Ferreira, C. A. R. Hoare, Hernán C. Melgratti, and Ugo Montanari. Comparing two approaches to compensable flow composition. In Martín Abadi and Luca de Alfaro, editors, *CONCUR*, volume 3653 of *Lecture Notes in Computer Science*, pages 383–397. Springer, 2005.

[BBM05] Michele Boreale, Maria Grazia Buscemi, and Ugo Montanari. A general name binding mechanism. In Rocco De Nicola and

Davide Sangiorgi, editors, *TGC*, volume 3705 of *Lecture Notes in Computer Science*, pages 61–74. Springer, 2005.

[BCF+08] Massimo Bartoletti, Vincenzo Ciancia, Gianluigi Ferrari, Roberto Guanciale, Daniele Strollo, and Roberto Zunino. LâĂŹorientamento ai servizi. *Mondo Digitale*, March 2008.

[BDFZ07] M. Bartoletti, P. Degano, G. Ferrari, and R. Zunino. Secure service orchestration. In *FOSAD*, volume 4667 of *Lecture Notes in Computer Science*. Springer, 2007.

[BEK+00] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. *Simple Object Access Protocol (SOAP) 1.1*. W3C Recommendation, http://www.w3.org/TR/2000/NOTE-SOAP-2000058/, 2000.

[BF04] Michael Butler and Carla Ferreira. An Operational Semantics for StAC, a Language for Modelling Long-Running Business Transactions. In Rocco De Nicola, Gianluigi Ferrari, and Greg Meredith, editors, *International Conference on Coordination Models and Languages*, volume 2949 of *Lecture Notes in Computer Science*, pages 87–104. Springer-Verlag, 2004.

[BFM+05] Roberto Bruni, Gian Luigi Ferrari, Hernán C. Melgratti, Ugo Montanari, Daniele Strollo, and Emilio Tuosto. From Theory to Practice in Transactional Composition of Web Services. In Mario Bravetti, Leïla Kloul, and Gianluigi Zavattaro, editors, *EPEW/WS-FM*, volume 3670 of *Lecture Notes in Computer Science*, pages 272–286. Springer, 2005.

[BLMT08] Roberto Bruni, Ivan Lanese, Hernán C. Melgratti, and Emilio Tuosto. Multiparty sessions in soc. In Doug Lea and Gianluigi Zavattaro, editors, *COORDINATION*, volume 5052 of *Lecture Notes in Computer Science*, pages 67–82. Springer, 2008.

[BLZ03] Laura Bocchi, Cosimo Laneve, and Gianluigi Zavattaro. A calculus for long-running transactions. In Elie Najm, Uwe Nestmann, and Perdita Stevens, editors, *FMOODS*, volume 2884 of *Lecture Notes in Computer Science*, pages 124–138. Springer, 2003.

[BMM04] Roberto Bruni, Hernán C. Melgratti, and Ugo Montanari. Nested commits for mobile calculi: Extending join. In Jean-Jacques Lévy, Ernst W. Mayr, and John C. Mitchell, editors, *IFIP TCS*, pages 563–576. Kluwer, 2004.

[BMM05] Roberto Bruni, Hernán Melgratti, and Ugo Montanari. Theoretical foundations for compensations in flow composition languages. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 209–220, New York, NY, USA, 2005. ACM Press.

[Bou92] Gerard Boudol. Asynchrony and the pi-calculus. Technical Report RR-1702, INRIA Sofia-Antipolis, May 1992.

[BPE] BPEL open issues. http://www.choreology.com/external/WS_BPEL_issues_list.html.

[BPS97] Tim Bray, Jean Paoli, and Chris M. Sperberg-McQueen. Extensible Markup Language (XML). *The World Wide Web Journal*, 2(4):29–66, 1997.

[BRR87] Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors. *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part I, Proceedings of an Advanced Course, Bad Honnef, 8.-19. September 1986*, volume 254 of *Lecture Notes in Computer Science*. Springer, 1987.

[CCMW01] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (wsdl) 1.1. http://www.w3.org/TR/wsdl.html, March 2001. W3C Note.

[CFGS08] Vincenzo Ciancia, Gian Luigi Ferrari, Roberto Guanciale, and Daniele Strollo. Checking correctness of transactional behaviors. In Kenji Suzuki, Teruo Higashino, Keiichi Yasumoto, and Khaled El-Fakih, editors, *FORTE*, volume 5048 of *Lecture Notes in Computer Science*, pages 134–148. Springer, 2008.

[CFSG08] Vincenzo Ciancia, Gianluigi Ferrari, Daniele Strollo, and Roberto Guanciale. Global coordination policies for services. In *FACS08 International Workshop on Formal Aspects of Component Software*, ENTCS. Elsevier, 2008. In print.

[CG98] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In Maurice Nivat, editor, *FoSSaCS*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer, 1998.

[CGL86] Nicholas Carriero, David Gelernter, and Jerrold Leichter. Distributed data structures in linda. In *POPL*, pages 236–242, 1986.

[CGV$^+$02] Mandy Chessell, Catherine Griffin, David Vines, Michael Butler, Carla Ferreira, and Peter Henderson. Extending the concept of transaction compensation. *IBM Syst. J.*, 41(4):743–758, 2002.

[CHY07] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. In *ESOP 2007*, volume 4421 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2007.

[CKM$^+$03] Francisco Curbera, Rania Khalaf, Nirmal Mukhi, Stefan Tai, and Sanjiva Weerawarana. The Next Step in Web Services. *Communications of the ACM*, 46(10):29–34, 2003.

[CRW98] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design of a scalable event notification service: Interface and architecture. Technical Report CU-CS-863-98, Department of Computer Science, University of Colorado, August 1998.

[CRW00] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Annual Symposium on Principles of Distributed Computing PODC*, pages 219–227, 2000.

[CW02] Antonio Carzaniga and Alexander L. Wolf. Content-based networking: A new communication infrastructure. In *IMWS ’01: Revised Papers from the NSF Workshop on Developing an Infrastructure for Mobile and Wireless Systems*, volume 2538 of *Lecture Notes in Computer Science*, pages 59–68, London, UK, 2002.

[CW03] Antonio Carzaniga and Alexander L. Wolf. Forwarding in a content-based network. In Anja Feldmann, Martina Zitterbart, Jon Crowcroft, and David Wetherall, editors, *Proceedings of* *the ACM SIGCOMM 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 25-29, 2003, Karlsruhe, Germany*, pages 163–174. ACM Press, 2003.

[Ecla] Eclipse Foundation. Eclipse Graphical Modeling Framework. Technical report. http://www.eclipse.org/gmf/.

[Eclb] Eclipse Foundation. Eclipse Modeling Framework. Technical report. http://www.eclipse.org/modeling/emf/.

[Eclc] Eclipse Foundation. Graphical Editing Framework. Technical report. http://www.eclipse.org/modeling/gef/.

[EFGK03] Patrick Th. Eugster, Pascal Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[EG04] Patrick Th. Eugster and Rachid Guerraoui. Distributed programming with typed events. *IEEE Software*, 21(2):56–64, March/April 2004.

[FGS06a] Gian Luigi Ferrari, Roberto Guanciale, and Daniele Strollo. Event based service coordination over dynamic and heterogeneous networks. In Asit Dan and Winfried Lamersdorf, editors, *ICSOC*, volume 4294 of *Lecture Notes in Computer Science*, pages 453–458. Springer, 2006.

[FGS06b] Gian Luigi Ferrari, Roberto Guanciale, and Daniele Strollo. Jscl: A middleware for service coordination. In Elie Najm, Jean-François Pradat-Peyre, and Véronique Donzeau-Gouge, editors, *FORTE*, volume 4229 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 2006.

[FGSTa] Gianluigi Ferrari, Roberto Guanciale, Daniele Strollo, and Emilio Tuosto. Debugging distributed systems with causal nets. In *PNGT 2008*.

[FGSTb] Gianluigi Ferrari, Roberto Guanciale, Daniele Strollo, and Emilio Tuosto. Refactoring long running transactions. In *WSFM 2008*.

[FGST07] GianLuigi Ferrari, Roberto Guanciale, Daniele Strollo, and Emilio Tuosto. Coordination via types in an event-based framework. In John Derrick and Jüri Vain, editors, *FORTE*, volume 4574 of *Lecture Notes in Computer Science*, pages 66–80. Springer, 2007.

[FGST08] Gian Luigi Ferrari, Roberto Guanciale, Daniele Strollo, and Emilio Tuosto. Event-based service coordination. In Pierpaolo Degano, Rocco De Nicola, and José Meseguer, editors, *Concurrency, Graphs and Models*, volume 5065 of *Lecture Notes in Computer Science*, pages 312–329. Springer, 2008.

[Gel85] David Gelernter. Generative communications in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, volume ISBN 0-201-63361-2. Addison-Wesley, 1995.

[GLG$^+$06] Claudio Guidi, Roberto Lucchi, Roberto Gorrieri, Nadia Busi, and Gianluigi Zavattaro. A calculus for service oriented computing. In *ICSOC*, volume 4294 of *Lecture Notes in Computer Science*, pages 327–338. Springer, 2006.

[GMS87] Hector Garcia-Molina and Kenneth Salem. Sagas. In Umeshwar Dayal and Irving L. Traiger, editors, *SIGMOD Conference*, pages 249–259. ACM Press, 1987.

[Gra81] Jim Gray. The transaction concept: virtues and limitations (invited paper). In *VLDB '1981: Proceedings of the seventh international conference on Very Large Data Bases*, pages 144–154. VLDB Endowment, 1981.

[Gro02] OMG Group. Business Process Modeling Notation. http://www.bpmn.org, 2002.

[Gua09] Roberto Guanciale. *The Signal Calculus: beyond message based coordination for services*. PhD thesis, IMT Institute for Advanced Studies, Lucca, 2008/09.

[GZ97] David Gelernter and Lenore D. Zuck. On what linda is: Formal description of linda as a reactive system. In David Garlan and Daniel Le Métayer, editors, *COORDINATION*, volume 1282 of *Lecture Notes in Computer Science*, pages 187–204. Springer, 1997.

[HG06] Yi Huang and Dennis Gannon. A comparative study of web services-based event notification specifications. In *ICPP Workshops*, pages 7–14. IEEE Computer Society, 2006.

[HP03] Haruo Hosoya and Benjamin C. Pierce. Xduce: A statically typed xml processing language. *ACM Trans. Internet Techn.*, 3(2):117–148, 2003.

[HT91] Kohei Honda and Mario Tokoro. On asynchronous communication semantics. In Mario Tokoro, Oscar Nierstrasz, and Peter Wegner, editors, *Object-Based Concurrent Computing*, volume 612 of *Lecture Notes in Computer Science*, pages 21–51. Springer, 1991.

[ibma] IBM Redbooks. Patterns: Service-Oriented Architecture and Web Services. http://www.redbooks.ibm.com/abstracts/sg246303.html.

[IBMb] IBM. Business processes and workflow in the Web services world. http://www-128.ibm.com/developerworks/webservices/library/ws-work.html.

[IBMc] IBM. Web Services Flow Language (WSFL) Specification. http://www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf.

[IBM05] IBM. Web services transactions specifications. http://www.ibm.com/developerworks/library/specification/ws-tx/, 2005. Technical Report.

[Lit03] Mark Little. Transactions and web services. *Commun. ACM*, 46(10):49–54, 2003.

[LP03] Ying Liu and Beth Plale. Survey of publish subscribe event systems. Technical Report TR574, Computer Science Department, Indiana University, 2003.

[LPT07] Alessandro Lapadula, Rosario Pugliese, and Francesco Tiezzi. A calculus for orchestration of web services. In *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2007.

[LZ05] Cosimo Laneve and Gianluigi Zavattaro. Foundations of web transactions. In Vladimiro Sassone, editor, *FoSSaCS*, volume 3441 of *Lecture Notes in Computer Science*, pages 282–298. Springer, 2005.

[MG05] Manuel Mazzara and Sergio Govoni. A case study of web services orchestration. In Jean-Marie Jacquet and Gian Pietro Picco, editors, *COORDINATION*, volume 3454 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2005.

[MH05] Jan Mendling and Michael Hafner. From inter-organizational workflows to process execution: Generating bpel from ws-cdl. In Robert Meersman, Zahir Tari, Pilar Herrero, Gonzalo Méndez, Lawrence Cavedon, David Martin, Annika Hinze, George Buchanan, María S. Pérez, Víctor Robles, Jan Humble, Antonia Albani, Jan L. G. Dietz, Hervé Panetto, Monica Scannapieco, Terry A. Halpin, Peter Spyns, Johannes Maria Zaha, Esteban Zimányi, Emmanuel Stefanakis, Tharam S. Dillon, Ling Feng, Mustafa Jarrar, Jos Lehmann, Aldo de Moor, Erik Duval, and Lora Aroyo, editors, *OTM Workshops*, volume 3762 of *Lecture Notes in Computer Science*, pages 506–515. Springer, 2005.

[Mica] Microsoft. Distributed Component Object Model (DCOM). http://msdn.microsoft.com/en-us/library/ms878122.aspx. Technical Specifications.

[Micb] Microsoft. Web Services for Business Process Design, XLANG. http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm.

[Mic05] SUN Microsystems. JavaSpaces Service Specification. http://java.sun.com/products/jini/2.0/doc/specs/html/js-spec.html, 2005. Part of Jini Specifications.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[Mil92] Robin Milner. The polyadic pi-calculus (abstract). In Rance Cleaveland, editor, *CONCUR*, volume 630 of *Lecture Notes in Computer Science*, page 1. Springer, 1992.

[Mil93] Robin Milner. The polyadic π-calculus: A tutorial. In Friedrich L. Bauer, Wilfried Brauer, and Helmut Schwichtenberg, editors, *Logic and Algebra of Specification, Proceedings of International NATO Summer School (Marktoberdorf, Germany, 1991)*, volume 94 of *Series F*. NATO ASI, Springer-Verlag, 1993. Available as Technical Report ECS-LFCS-91-180, University of Edinburgh, October 1991.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.

[Mis04] Jayadev Misra. A programming model for the orchestration of web services. In *SEFM*, pages 2–11. IEEE Computer Society, 2004.

[ML04] Manuel Mazzara and Roberto Lucchi. A framework for generic error handling in business processes. *Electr. Notes Theor. Comput. Sci.*, 105:133–145, 2004.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100(1):1–77, 1992.

[Net99] Netscape. An Exploration of Dynamic Documents. http://wp.netscape.com/assist/net_sites/pushpull.html, 1999.

[OAS02] OASIS. Business transaction protocol. http://www.oasis-open.org/committees/download.php/1184/2002-06-03.BTP_cttee_spec_1.0.pdf, June 2002. Technical Report v1.1.

[OAS04] OASIS. Web services reliable messaging. http://docs.oasis-open.org/wsrm/ws-reliability/v1.1/wsrm-ws_reliability-1.1-spec-os.pdf, November 2004. Technical Report v1.1.

[OAS06] OASIS. Web services security. http://www.oasis-open.org/committees/download.php/16790/wss-v1.

1-spec-os-SOAPMessageSecurity.pdf, February 2006. Technical Report v1.1.

[OAW] OAW. OpenArchitectureWare MDA/MDD generator framework. http://www.openarchitectureware.org/.

[oCSIU] Department of Computer Science. Indiana University. XSoap. www.extreme.indiana.edu/xgws/xsoap/.

[OMG02] OMG. Business Process Modeling Language. http://www.bpmi.org, 2002.

[Pap03] Mike P. Papazoglou. Service-Oriented Computing: Concepts, Characteristics and Directions. In *Web Information Systems Engineering (WISE'03)*, Lecture Notes in Computer Science, pages 3–12. Springer-Verlag, 2003.

[PS98] Frantisek Plasil and Michael Stal. An architectural view of distributed objects and components in corba, java rmi and com/dcom. *Software - Concepts and Tools*, 19(1):14–28, 1998.

[QW98] Paola Quaglia and David Walker. On encoding $p\pi$ in $m\pi$. In *Foundations of Software Technology and Theoretical Computer Science*, pages 42–53, 1998.

[RF] David Recordon and Brad Fitzpatrick. *OpenID Authentication 1.1*. Available at http://openid.net/specs/openid-authentication-1_1.html.

[Sof05] Software Engineering Research Laboratory. Siena (Scalable Internet Event Notification Architectures). http://serl.cs.colorado.edu/~serl/dot/siena.html, 2005.

[Spe] OASIS Bpel Specifications. OASIS - BPEL. http://www.oasis-open.org/cover/bpel4ws.html.

[SS83] Dale Skeen and Michael Stonebraker. A formal model of crash recovery in a distributed system. *IEEE Transactions on Software Engineering*, 9(3):219–228, 1983.

[Ste04] Stephen A. White - IBM. Introduction to BPMN. http://www.bpmn.org/Documents/Introduction%20to%20BPMN.pdf, May 2004.

[Ste05] Stephen A. White - IBM. Mapping BPMN to BPEL Example. http://www.bpmn.org/Documents/Mapping%20BPMN%20to%20BPEL%20Example.pdf, February 2005.

[SUNa] SUN. Java Message Service (JMS). http://java.sun.com/products/jms/.

[SUNb] SUN. Java Remote Method Invocation (Java RMI). http://java.sun.com/products/jdk/rmi/.

[SW02] Davide Sangiorgi and David Walker. *The π-Calculus: a Theory of Mobile Processes*. Cambridge University Press, 2002.

[TAJ03] David Tam, Reza Azimi, and Hans-Arno Jacobsen. Building content-based publish/subscribe systems with distributed hash tables. In Karl Aberer, Vana Kalogeraki, and Manolis Koubarakis, editors, *Databases, Information Systems, and Peer-to-Peer Computing*, volume 2944 of *Lecture Notes in Computer Science*, pages 138–152, 2003.

[Tea] OMG Team. CORBA (Common Object Request Broker Architecture). http://www.omg.org.

[TW97] D. Thompson and D. Watkins. Comparisons between corba and dcom: Architectures for distributed computing. In *TOOLS (24)*, pages 278–283. IEEE Computer Society, 1997.

[UO01] UN/CEFACT and OASIS. ebXML Business Process Specification Schema. http://www.ebxml.org/specs/ebBPSS.pdf, 2001.

[vdAtHKB00] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. Web Page, 2000.

[W3Ca] W3C. Web Service Choreography Interface (WSCI). http://www.w3c.org/TR/wsci.

[W3Cb] W3C. Web Services. http://www.w3.org/2002/ws/.

[W3Cc] W3C. Web Services Choreography Description Language (v.1.0). Technical report.

[W3C00] W3C. Universal Description, Discovery and Integration (UDDI). Technical report, 2000.

[WCC04] G.C. Wells, Alan Chalmers, and P.G. Clayton. Linda implementations in Java for concurrent systems. *Concurrency and Computation: Practice and Experience*, 16(10):1005–1022, August 2004.

[WCG+06] Martin Wirsing, Allan Clark, Stephen Gilmore, Matthias M. Hölzl, Alexander Knapp, Nora Koch, and Andreas Schroeder. Semantic-based development of service-oriented systems. In *FORTE 2006*, volume 4229 of *Lecture Notes in Computer Science*, pages 24–45. Springer, 2006.

[WfM02] WfMC. Workflow process definition interface - XML Process Definition Language. http://www.wfmc.org/standards/docs/TC-1025_10_beta_xpdl_073002.pdf, 2002.

[WvdADtH03a] Petia Wohed, Wil M. P. van der Aalst, Marlon Dumas, and Arthur H. M. ter Hofstede. Analysis of Web Services Composition Languages: The Case of BPEL4WS. In Il-Yeol Song, Stephen W. Liddle, Tok Wang Ling, and Peter Scheuermann, editors, *ER*, volume 2813 of *Lecture Notes in Computer Science*, pages 200–215. Springer, 2003.

[WvdADtH03b] Petia Wohed, Wil M.P. van der Aalst, Marlon Dumas, and Arthur H.M. ter Hofstede. Pattern Based Analysis of BPEL4WS. Technical report, Department of Computer and Systems Sciences Stockholm University/The Royal Institute of Technology, Sweden, nov 2003.

[xml04] Xml schema. Technical report, W3C, 2004.

[Yos96] Nobuko Yoshida. Graph types for monadic mobile processes. In *Foundations of Software Technology and Theoretical Computer Science*, pages 371–386, 1996.