# Dynamic Reconfiguration of Service-oriented Architectures

Manel Fredj

N$^{O}$ Ordre de la thèse :

THÈSE

présentée

DEVANT L'UNIVERSITÉ DE PARIS VI

pour obtenir le grade de:

*DOCTEUR DE L'UNIVERSITÉ DE PARIS VI*

Mention: **Informatique**

PAR

**Manel FREDJ**

Équipe d'accueil: **INRIA Paris-Rocquencourt, Équipe-Projet ARLES**

École Doctorale: **Informatique, Télécommunications et Électronique de Paris**

TITRE DE LA THÈSE:

# Reconfiguration dynamique des architectures orientées services

SOUTENUE LE 10 / 12 / 2009 devant la commission d'examen

COMPOSITION DU JURY

| | | |
|---|---|---|
| **Pierre SENS** | Pr, Université de Pierre et Marie Curie - Paris VI, France | Président du jury |
| **Nicole LEVY** | Pr, Université de Versailles Saint-Quentin-en-Yvelines, France | Rapporteur |
| **Apostolos ZARRAS** | Pr, Université de Ioannina, Grèce | Rapporteur |
| **Samir TATA** | Pr, Telecom SudParis Evry, France | Examinateur |
| **Valérie ISSARNY** | DR, INRIA Paris-Rocquencourt, France | Directrice de thèse |
| **Nikolaos GEORGANTAS** | CR, INRIA Paris-Rocquencourt, France | Co-Directeur de thèse |

*To My Parents*

Doctor of Philosophy Dissertation of Manel FREDJ ARLES/INRIA/Paris VI

# Dynamic Reconfiguration of Service-oriented Architectures

## Abstract

Runtime service reconfiguration is put forward as one of the means by which we may provide dependable service-oriented architectures (SOA), and more precisely, continuity in service provisioning, and robustness in the presence of change. Indeed, with the advent of wireless networks, computing environments are becoming highly dynamic. From a user-oriented point of view, this dynamics comes at the price of dependability, due to runtime variations in terms of (1) service availability, and (2) network connection/infrastructure availability, according to user/service mobility. In this context, the main focus of this thesis is to incorporate support for dynamic reconfiguration in SOA systems, in order to tolerate runtime variations and ensure continuity in service provisioning for the users. In particular, we focus on *middleware* support for runtime service reconfiguration. Our main contribution consists in enabling service continuity by (1) *substituting* a service that becomes unavailable at runtime with a semantically similar one, and (2) *translating* and *transferring* the current state of interaction to the substitute service in order to resume the execution after its interruption. The need for state translation is due to the environments' heterogeneity, since the unavailable and substitute services are not assumed to be identically *implemented*, nor are they identically *described*. However, state translation may not be sufficient to guarantee state compatibility between the substitute service and the unavailable one: in some cases, the substitute service may be compatible with an earlier state of interaction, instead of the last one. Hence, we need to invalidate a set of results performed by the unavailable service, in order to synchronize the state of the substitute service.

Indeed, the execution of the substitute service may provide different results from those provided by the execution of the unavailable service. In the case of service composition, still-available services –involved in the SOA system– may be affected by the substitution due to their data dependencies with the substituted service. Thus, the middleware synchronizes the state of still-available services according to the state transferred to the substitute service.

The outcome of our contribution is SIROCCO (ServIce Reconfiguration upOn serviCe unavailability and Connectivity lOss), a middleware infrastructure that enables transparent runtime reconfiguration of SOA systems upon service unavailability. The middleware discovers candidate substitute services that can be used in the place of the service that becomes unavailable. It then tries to identify the best service amongst these candidates that can be used as an actual substitute. In the best case, the selected substitute service must be such that its current state can be synchronized with the *last* state of the service that is substituted. In the case of service composition, the middleware also checks data dependencies with the still-available services and synchronizes their state with respect to the transferred state. The above concepts of SIROCCO are discussed along with an experimental evaluation of our prototype. Our findings show that SIROCCO provides the necessary means for achieving dynamic service reconfiguration, where the gain in close-to-seamless continuity in service provisioning outweighs the computing and communication overhead on the execution of the SOA system.

**Keywords:** Dependability, Middleware, Service Unavailability, Dynamic Substitution, Service-oriented Architecture.

# Preface

This thesis describes the work carried out in the Project-Team ARLES at the French National Institute for Research in Computer Science and Control / Research Centre INRIA Paris-Rocquencourt, between October 2005 and January 2009.

INRIA Paris - Rocquencourt Research Centre
Domaine de Voluceau
BP 105, 78153 Le Chesnay Cedex - France

# Acknowledgements

I would like to express my gratitude to all those who gave me the possibility to complete this thesis. First of all I would like to begin with my advisors Valérie Issarny and Nikolaos Georgantas, who always provided me with right guidance and support. It has been a great pleasure to work with them, both from a professional and a personal points of view. I am deeply indebted to Valérie whose help, suggestions and advices helped me in all the time of research and writing of this thesis. Valérie, thank you for welcoming me in your project-team, giving me the opportunity to develop this experience abroad and challenging me every day to make me a better researcher. Your writing skills and diligence for the corrections of my manuscript made it a better document. Nikolaos, it was always a pleasure to share with you new ideas and results and your constant patience, interest and wisdom allowed me to push through and complete this thesis.

Many thanks go to my reviewers and examiners of my PhD defense: Nicole Lévy, Apostolos Zarras, Pierre Sens and Samir Tata. It is a great honor to have you all as part of the examination committee.

During those 3 years of PhD, I have met a lot of people outside and inside the work sphere that contributed to make this adventure possible and enjoyable, warm thanks go to all of them. In particular, I would like to thank the people with whom I shared the office room: Sonia Ben Mokhtar that I first met when I entered the INRIA, and Sandrine Beauche. We have had some great time together, and I am sure that there is still a lot more to come. A special thank goes to Roberto as we shared both hard and nice experiences of doing a PhD. I have enjoyed working with many colleagues, members of the project-team Arles, in particular, my colleague and friend Amel Bennaceur for the various discussions we had during daily travel to work and back home. Among the people I met there, I would like to thank Emmanuelle Grousset for easing the administrative tasks, and also for the running training we enjoyed during the sunny days.

My warmest thanks naturally go to my family, my parents Dalenda and Abdelmajid for having always stood behind me through the good and bad times. I would probably not have gone so far in my studies without their encouragements and the education that they gave me. My last –but not least– thanks go to Salem Rabiaa for his steady support during the last two years. Doing this PhD has required some sacrifices and I am deeply grateful to him not only because he accepted them, but because he fully and wholeheartedly endorsed them.

# Contents

# List of Figures

*Our computers should be like an invisible foundation that is quickly forgotten but always with us, and effortlessly used throughout our lives.*

Mark, Weiser

# 1 Introduction

Nowadays, user-centric computing envisions to ease the use of computing facilities in order to help users achieving their daily tasks. This vision was first articulated by Mark Weiser in 1988 at the Computer Science Lab at Xerox PARC. He envisioned ubiquitous computing (UbiCom) environments, where humans are surrounded by computing and networking technologies unobtrusively embedded in their surroundings. Twenty years later, a trend towards computing ubiquity is being realized through pervasive computing (PerCom) [Satyanarayanan, 2001], which focuses on integrating computing facilities in dynamic environments brought about by the convergence of mobile and tiny devices. With more than 2 billion terminals in commercial operation world-wide in 2005 [Buckley, 2006], wireless and mobile technologies have enabled a first wave of pervasive systems and applications. The Ambient Intelligence (AmI) paradigm goes even a step beyond computing ubiquity, focusing on adding smart behaviors to computing devices so that they can adapt themselves to the user's context.

The underlying feature of the above vision is *freedom*; the freedom of *mobility*, *accessibility* and *interaction*. The freedom of *mobility* is enabled with the advent of numerous handheld devices and new Radio Access Technologies (RATs) such as 3G, WiFi, that interconnect mobile and stationary devices. The freedom of *accessibility* is enabled with the cohabitation of infrastructure-based and infrastructure-less (wireless ad hoc) networks; new combined networks such as Beyond-3G (B3G), enable users to access networked resources (e.g., data and computation) on demand, anywhere, and from various devices supporting different communication capabilities. The freedom of *interaction* with the networked resources is enabled by overcoming the heterogeneity of the underlying technologies such as, hardware platforms, operating systems, programming languages, interaction protocols.

Contributing to the realization of the above freedom, service-oriented architectures (SOA) [Papazoglou and Georgakopoulos, 2003] abstract heterogeneous networked resources and computing facilities as *services*. Service-oriented applications are built on top of services which have well defined interfaces, composing them into loosely-coupled structures. The advantage of loose coupling is that services can be made generally accessible to a large community of clients, as opposed to being specifically developed for a limited group of clients, as it was the case in conventional, CORBA-style integration.

Still, the SOA abstraction contributes, but is not sufficient to realize the above freedom. Indeed, from a user-oriented point of view, this freedom comes at the price of dependability, due to runtime variations in terms of (1) services availability, and (2) network connection/infrastructure availability according to the user/services mobility. The main focus of this thesis is to incorporate support for runtime reconfiguration into SOA systems in order to tolerate runtime variations and contribute to ensuring dependability.

The rest of this chapter is organized as follows. In Section 1.1, we elaborate on the motivation of our work for supporting dynamic reconfiguration in service-oriented architectures. Then, Section 1.2 outlines our contributions, and Section 1.3 presents the structure of this document.

## 1.1 Supporting Dynamic Reconfiguration in SOA Systems

Due to the advantages provided by the SOA paradigm, SOA systems have proliferated and evolved in the last few years. Functionalities provided by services are becoming more sophisticated; they support complex interactions with the client with sequences of message exchange (conversations), as opposed to a simple request/response interaction. Across these interactions, the service may hold a state in order to avoid data redundancy in the exchanged messages. Furthermore, depending on the services available in the user environment, a single service functionality may not be sufficient to serve an advanced client request. In such a case, it is required to compose multiple service functionalities to provide a full response to a client request.

To this aim, an SOA realization should support:

1. Discovering the service functionalities available in the user environment,
2. Providing access to the service providers,
3. Potentially composing the service functionalities to serve an advanced user request, and
4. Enabling the correct consumption of these functionalities according to the service-supported conversations.

The above listed facilities should further account for stateful services and complex interactions with clients. Service discovery and composition may be based on either (1) *syntactic matching* between the descriptions of the user request and the service functionality, assuming that users and services use a common syntax for denoting their respective requests and functionalities and their semantics, or (2) *semantic matching* [Paolucci et al., 2002], introducing thereby more openness that overtakes syntactic heterogeneity.

Handling the above facilities using middleware technologies appears to be the right approach that provides transparent, reusable functionalities for both services and clients. However, a service-oriented middleware (SOM) is not sufficient to overcome all the issues that todays' computing environments present. Indeed, while performing their daily tasks, users frequently move from one environment to another, which provides them with various, independent resources/-computing facilities. These facilities are deployed over heterogeneous networks (e.g., 3G, WiFi, wireless ad hoc). However, since users and services join and leave the networked environment at their convenience, services may easily become unavailable in an unpredictable way; it is hence difficult to guarantee the reliability (i.e., continuity in service provisioning) of applications engaging these services. Therefore, the environment's dynamics raises several issues illustrated through the following scenario.

### 1.1.1 Illustrating Scenario

*Richard goes five days a week to his work using public transportation. In the train station (Figure 1.1-Env1), in addition to paid Internet access, certain free services are provided via the wireless LAN* [1]*. Free services enable end-users to (1) check train timetables, and (2) easily book train tickets for a specific destination.*

[1] Local Area Network

*On Friday morning, waiting for the city train to depart, Richard is thinking about visiting his parents for the weekend; he decides to use the free service for train ticket reservation in order to book a ticket to his parents' city.*

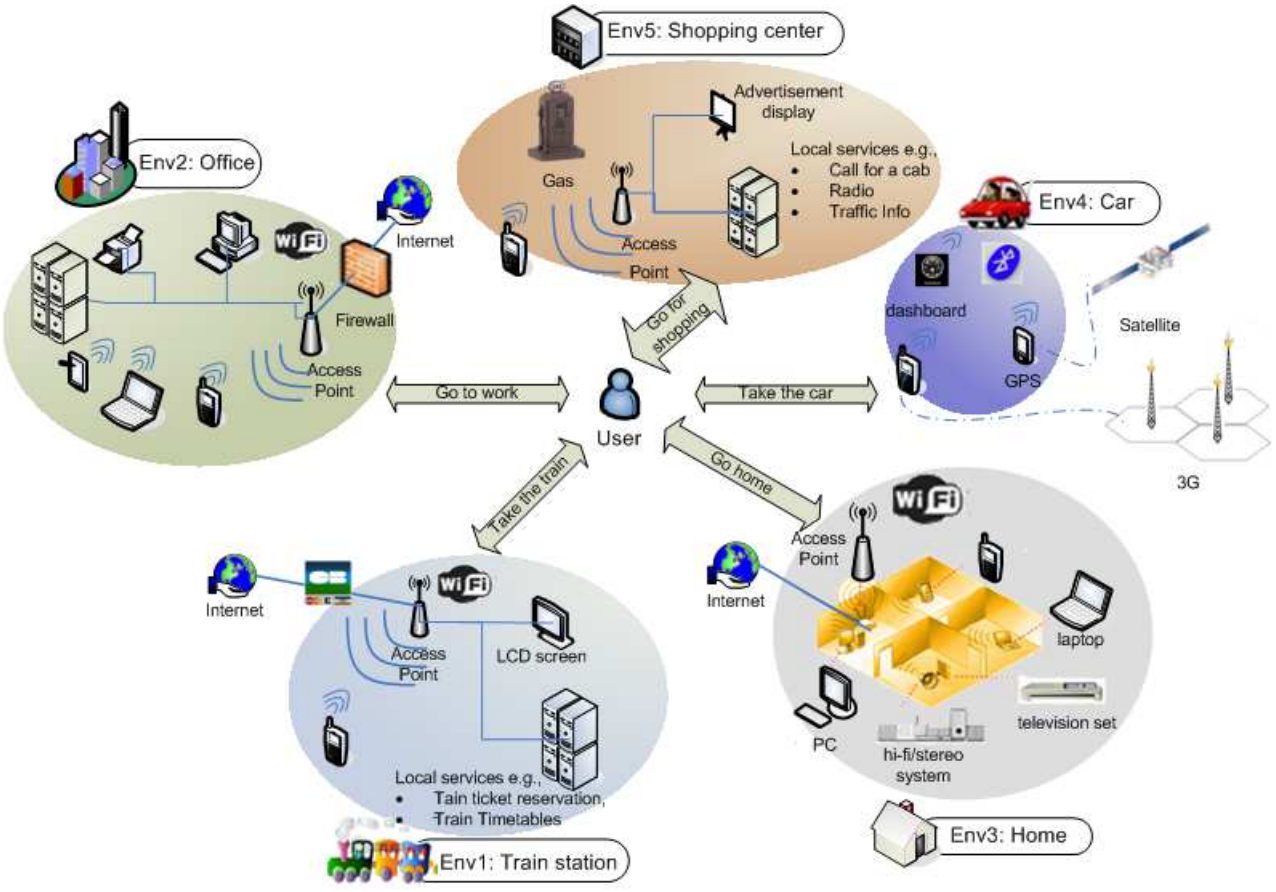



Figure 1.1: Pervasive scenario

*However, as Richard's train is leaving the station, the wireless network connection enabling the interaction with the "train ticket reservation" service is interrupted. To save the connection cost of the 3G communication, Richard decides to book his train ticket at his office, using his enterprise-provided Internet access (Figure 1.1-Env2). However, the enterprise firewall restricts the access to specific Internet services, including the ticket reservation service that Richard was using in the train station. As, Richard does not have a particular benefit from using the same service instance for ticket reservation as the one in the train station, he decides to book his tickets using another service. Some subcontracted Web services for train ticket reservation are reachable via the restricted Internet connection, enabling Richard to resume and complete his train ticket reservation.*

*Later on the same day, Richard hears a sudden rail strike announcement, thus he finally decides to drive to his parents' city. After work, he gets back to his place (Figure 1.1-Env3) to take some clothes, and he also connects –in an ad hoc way– his smartphone to his home server – which is connected to the Internet– in order to download information about the highway traffic and the weather forecast for the weekend. Once he is ready to go, he leaves his house without completing the entire information download.*

*In his car (Figure 1.1-Env4), Richard activates the 3G network interface of his smartphone to*

*resume the information download, in the absence of any alternative free communication means. He also turns on his GPS*[2]*. According to the traffic information and the current GPS coordinates of the car, Richard has launched on his smartphone a service (called "route optimization") that selects the shortest way to his parents' city, avoiding at the same time the traffic jams. "Route optimization" is a composite service that dynamically integrates the functionalities of (1) the GPS, (2) the traffic information service, and (3) a service (called "route provision") that is deployed on Richard's smartphone, and which graphically provides the directions to reach a specific destination.*

*On the way to his parents' place, Richard stops at a gas station to refuel his car (Figure 1.1-Env5). He first goes to the nearby store to buy a sandwich. In the store, Richard discovers a set of free services. Among them, there is a service that provides information about the highway traffic, which is similar to the one participating in the "route optimization" service composition. "Route optimization" integrates that service in order to update the traffic information, and consequently the directions in case of traffic jam. Richard's smartphone shifts from the 3G network radio interface to the WIFI one in order to reduce the charges induced by the 3G connection use. Hence, Richard enjoys eating his sandwich while updating the directions to his parents' place. He then refuels the car, and gets to his destination using the directions provided by the "route optimization" service.*

The above scenario points out two main challenges:

- First, in various situations, Richard had to use a new service replacing the one he was using due to network disconnection, as it was the case for the disconnection from his home server. In other situations, the substitution was voluntary to reduce the cost of using the 3G connection, as it was the case when Richard entered the gas station store. In these situations, the networked environment provided Richard with a connection to a new service instance that could replace the disconnected one. Two requirements emerge: (1) the service substitute has to provide similar functionalities to the unavailable service, and (2) it should resume, if possible, the interaction with Richard from the point it was interrupted, *taking over* all the computation performed before the disconnection, and trying to synchronize accordingly, even in case that the two services are not identical.

- Second, some of the services available in Richard's networked environment (traffic information, GPS and "route provision") are composed and integrated dynamically to perform an advanced functionality ("route optimization"). However, substituting the traffic information service with a similar service (accessible *via* the wireless network of the store) may have an impact on the other services execution state (e.g., route provision service). The issue here is to identify the affected services involved in the service composition and restore their state according to their data dependencies with the state of the substituted service.

Section 1.1.2 discusses briefly the limits of existing solutions in dealing with the above two challenges and elaborates further on these challenges to set the goals of the present thesis.

### 1.1.2 Current Solutions and Challenges

As illustrated in the above scenario, in open, dynamic and heterogeneous environments as those of ubiquitous computing, several incidents may interrupt the interaction between the user and SOA systems, disrupting thereby the service continuity: networked services may become unavailable at runtime without beforehand notification due to (1) network problems (e.g., connectivity

[2]Global Positioning System

loss), or (2) service problems (e.g., service undeployment, or service failure). In such cases, the computation performed by the now unavailable service is left incomplete, and is lost.

Regarding the *network problems*, the connection with networked services cannot be guaranteed over time due to users and services mobility. To deal with connectivity loss, relevant network-based and nomadic solutions have been introduced to maintain the interaction with the same service provider despite the disconnection.

Traditionally, (network-based) mobility management solutions [Rappaport, 2001] rely on the core network to apply handoff [Ruggaber and Seitz, 2001] (vertical or horizontal), multi-homing [3] and mobility prediction [Rosa et al., 2005] solutions. In such solutions, users are able to continuously access the same service instance through methods such as access-point switching within the same network and data traffic redirection between different networks. However, these solutions are not suitable for infrastructure-less networks where mobile nodes communicate in an impromptu manner and networks are established on the fly.

Nomadic computing systems, such as Rover [Joseph et al., 1995], deal with intermittent connection between the client and a remote service provider due to insufficient wireless network coverage or limited bandwidth shared between multiple users. Specifically, they manage the disconnection time using caching and offline working techniques. The key assumption in these solutions is that the client will probably reconnect to the same server or some replica of the server. Then, the objective is to enable users to use their mobile devices even during periods of low or non-connectivity. The distinctive feature that differentiates todays' ubiquitous SOA systems from nomadic ones is that services available in the user environment do not have a beforehand knowledge of each other. Bindings with/between services are ad hoc and temporary. Thus, after a disconnection, a client is unlikely to reconnect to the same service provider or even a replica of it. It will rather connect to another provider that provides him/her with the required functionalities. Nevertheless, disconnection has been deeply studied in nomadic computing systems, which makes the experience gained from the nomadic system solutions useful.

Similarly for *service problems*, services may be undeployed at anytime, without beforehand notification according to their providers' decision. Services may also fail. To deal with service unavailability, replication-based approaches (passive [4] or active [5]) enable to substitute an unavailable service with an exact replica of it. By definition, an exact replica is able to *interpret* and *use* the state of the unavailable service in order to resume the execution from the point it was interrupted, ensuring thereby service continuity. For instance, several approaches [Salatge and Fabre, 2007, Maamar et al., 2008] rely on the construction of fault tolerant service groups out of unreliable services. The formulation of fault-tolerant groups of services [Salatge and Fabre, 2007] seems difficult in practice when considering that the constituent services may be offered by independent or even competitive organizations or businesses. In a realistic scenario, an agreement between independent businesses is required in order to register their online services in a group that realizes active replication, knowing that this will involve devoting precious resources to the group without any actual direct benefit (e.g., many instances of the same ticket reservation made by the same user to the active replicas, while only one of them will be validated at the end by the protocol that realizes the reservation process through active replication). Other approaches tackled the issue by substituting an entity with another dedicated backup entity

[3] Multi-homing allows a mobile node to set up at the same time multiple radio interfaces and multiple IP addresses, and therefore allows applications to perform switchovers between different radio interfaces during vertical handoffs without interrupting data transfer.

[4] In *passive replication*, each single request is processed on a single replica, and then its state is transferred to the other replicas

[5] *Active replication* is performed by processing the same request at every replica

using, e.g., built-in replication. However, the availability of such backup entities is not guaranteed in open and heterogeneous environments as the ones we are interested in. To overcome the heterogeneity of the environment, recent approaches [Calore et al., 2007, Mokhtar et al., 2008b] rely on computing the semantic similarities between a user request and a provided service. These efforts enable dynamic service provisioning and composition in heterogeneous environment, but do not ensure the runtime service continuity, as they do not consider state transfer. In the case that a service becomes unavailable, a semantic-similar service substitute is discovered and the interaction with the client is restarted from the beginning, even in the case of service composition.

The problem that we are considering is too complex to be solved by directly applying the above solutions. The difficulty lies in the *runtime* substitution of the now unavailable service, and particularly, in *saving* the computation performed up to the time of execution interruption. More specifically, an approach that accomplishes service continuity in dynamic, open and heterogeneous environments is faced with the following challenges. Firstly, it should be suitable for deployment in infrastructure-less (e.g., wireless ad hoc) networks and thus, cannot rely on any network infrastructure as handoff-based solutions do. Secondly, it should be flexible enough in selecting the candidate substitute service, without being restrictive to exact replicas, but only providing similar functionality to the unavailable service's one. This induces a third challenge of ensuring a correct substitution of the unavailable service, including (1) the *provision* and (2) the correct *interpretation* and *use* of the unavailable service state by the substitute service in order to resume the interaction with the client from the point at which it was interrupted. In the case of impossibility of interpreting or using the transferred state, a mechanism should be set for replaying, and potentially adapting, the sequence of exchanged messages and the messages content according to the substitute service logic. A fourth challenge emerges in the case of service composition: a mechanism should identify the impact of service unavailability and substitution on the services participating in the service composition, and if required, it restores their state consistency according to the data dependencies between them. Finally, an approach that complies with the above challenges should also be re-usable and application-independent; it should be adaptable according to the user requests and the services that are available in the user environments.

## 1.2 Contributions

As discussed in the previous section, existing solutions that enable continuity in service provisioning either rely on the network infrastructure, or constrain the services by requiring an exact replica as substitute service. Other more flexible solutions enable semantic service substitution to ensure service provisioning, but not runtime service continuity, as they do not deal with state transfer and state interpretation. In this thesis, we take advantage of the above solutions while restricting the least the user environments. We neither rely on the network infrastructure, nor assume the availability of an exact replica. We perform semantic service substitution while preserving the service continuity. Our main contribution consists in enabling service continuity by (1) *substituting* a service that becomes unavailable at runtime with a semantically similar one, and (2) *transferring* and *translating* the state of interaction to the substitute service in order to resume the execution from the point it was interrupted. The unavailable and substitute services are neither identically *implemented*, nor identically *described*. We propose a user-transparent [6] middleware approach that tolerates runtime service unavailability and reconfigures the client and the service(s) to ensure service continuity. The reconfiguration process supported by the

[6] to the degree possible

middleware consists in:

1. Proactively storing the service state in order to provide it to a substitute service if the service becomes unavailable.

2. Filtering the candidate substitute services out of a set of available services providing a functionality that can be used in the place of the functionality of the now unavailable service. This filtering is based on comparing the semantics of the provided functionality of the candidate service, with the one of the unavailable service.

3. Selecting amongst the candidate services the one that can be used as an actual substitute; in the best case, the selected substitute service will be such that its state can be synchronized with the last state stored of the service that is substituted. The decision is made by computing a compatibility degree between the candidate services and the service being substituted, i.e., the ability of the former services to interpret and use a state provided by the latter. In some cases, a candidate service may not be able to fully synchronize with the *latest* state of the unavailable service; the synchronization is then tried using an *earlier* state than the last stored one.

4. Alternatively, in the case of impossibility of state synchronization, adapting and replaying the sequence of messages that have been exchanged between the unavailable service and the client in order to reproduce the effect of the computation –this time– with the substitute service, transparently to the user to the degree possible (avoiding thereby the user intervention).

5. Finally, in case of service composition, limiting the side effects of service unavailability. This includes to restore the state consistency of the still-available services involved in the service composition with respect to data dependencies with the transferred state of the substituted service, and between their respective states.

Satisfying the above features, our contribution is SIROCCO (ServIce Reconfiguration upOn serviCe unavailability and Connectivity lOss), a middleware that enables runtime reconfiguration of SOA systems upon service unavailability.

## 1.3 Thesis Structure

This thesis is organized in three parts. Part I presents the fundamental concepts of SOA and dependability. It also presents a set of relevant approaches for runtime reconfiguration of SOA systems. Part II formalizes our approach for runtime reconfiguration of SOA systems. Part III presents the architecture that realizes our approach as well as the performance assessment of the runtime reconfiguration.

Part I includes two chapters: Chapter 2 presents the necessary background on service-oriented architectures (SOA), including an overview of SOA, the definition of their basic concepts, and the middleware solutions that enable the deployment of such architectures. Furthermore, it provides an overview of Web service technologies, which are the most commonly-used realization of SOA systems. Chapter 3 presents the basic concepts of dependability along with the type of failure we are interested in (i.e., service unavailability). We detail the existing techniques to deal with component unavailability in closed distributed systems and their limits of applicability in open, dynamic SOA systems. This stresses the need for providing a middleware solution that ensures

dependability in SOA-based environments. In the same chapter, we present a study of the state of the art of the existing solutions that target continuity in service provisioning, focusing more specifically on service substitution.

Part II includes four chapters: Chapter 4 presents an advanced service model that integrates the necessary concepts for supporting dependability. Based on this service model, Chapter 5 provides a formal definition of service substitution and the constraints under which a service is able to serve as a substitute for another one. To select a service substitute, Chapter 6 provides the rules for checking the compatibility between a candidate service and the unavailable one. The compatibility check is based on the definition of a compatibility degree, the value of which determines whether a service candidate is able to serve as a substitute for the unavailable one, or not. In the case of multiple candidates for substitution, the value of the compatibility degree associated with each candidate service enables classifying the set of candidates. This classification enables to select the service that best reduces the reconfiguration time and/or the loss of computation. To deal with the case that the substituted service makes part of a service composition, Chapter 7 presents the strategies and rules that enable to preserve the data coherence between the states of all the services participating in the service composition.

Part III includes two chapters: Chapter 8 details the architecture and the implementation of our middleware approach. It then assesses the use of our runtime reconfiguration through the means of a Web service implemented scenario and by measuring the performance of the runtime reconfiguration. Chapter 9 summarizes the work achieved in this thesis. It provides our conclusions and our future research directions as a continuity of, and beyond, this thesis.

# Part I

# Fundamentals and State of the Art

*Architecture starts when you carefully put two bricks together. There it begins.*
Ludwig Mies van der Rohe

# 2 Service-oriented Architectures

Traditional application design depends upon a tight interconnection of all its components, often running in the same process [Papazoglou and Dubray, 2004]. The complexity of these connections requires that the developers thoroughly understand and have the control over both ends of a connection. Moreover, once established, it is very difficult to extract one element and replace it with another. Service-oriented architecture (SOA) is an evolution of distributed computing based on the request/reply design paradigm for synchronous and asynchronous applications. Components' functions are modularized and presented as services for client applications. A key characteristic of these services is their loosely coupled nature. The service interface is defined independently of the technology implementing the service. Consequently, loosely coupled systems enable a simpler level of coordination and allow more flexible reconfiguration as opposed to tightly coupled systems that require agreement and shared context between the communicating components.

This chapter presents an overview of service-oriented architectures and introduces their key concepts. Section 2.1 first presents the basic roles within service-oriented architectures and an overview of their interactions with each other. To support these interactions, Section 2.2 stresses the need for a service-oriented middleware which enables easier and more effective (1) creation, (2) deployment, and (3) management of services across distributed infrastructures. Section 2.3 focuses more in detail on the notion of service. We propose a basic service model that integrates the basic concepts of a service along with their dependencies. One of the most popular realization of SOA systems is provided by Web services. In particular, Web services have standardized languages enabling to describe service concepts. Section 2.4 provides a background on the languages that enable to describe Web service functionalities. Section 2.5 integrates some of the concepts that are defined in the WS domain in our basic service model. Finally, Section 2.6 summarizes this chapter.

## 2.1 Basics of Service-oriented Architectures

Service-oriented architectures rely on three important roles, namely, the service provider, the service consumer and the service registry [Papazoglou, 2003].

A service provider is the entity that owns and implements the business logic that underlies the service. From an architectural perspective, it is the platform that hosts and controls the access to one or several services. A service consumer (or client) is the entity that requires a function to be executed. From an architectural perspective, it is the application that is looking for, and subsequently invoking a service. As illustrated in Figure 2.1, a service registry puts the two above roles in contact. It is a "searchable" directory where services publish their descriptions,

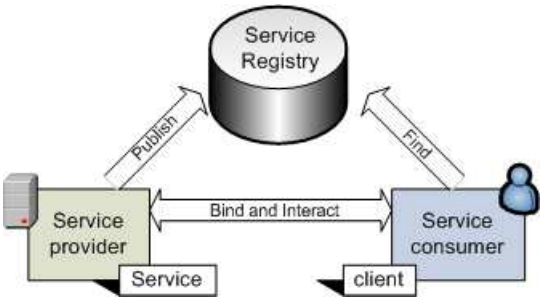



Figure 2.1: Service-oriented architectures

and clients find these descriptions, select services, and obtain binding information. In particular, the service selection is complicated when numerous services from various providers exist, all claiming to fulfill users' needs, or when each of the services uses a different naming taxonomy. To solve these problems, a service basically has to use expressive semantic means for describing its functional properties and non-functional ones such as quality of service (QoS). Then, clients provided with semantic search capabilities are able to search distributed registries for services with certain functional and QoS specifics.

Furthermore, an advanced request of a client may not be served by a single service, but requires to compose service functionalities. The service composition may be coordinated by the client, involving services that are not aware of each other. In such case, the service composition is called an *orchestration*. Alternatively, the service composition may be set up and performed seamlessly to the client, in peer-to-peer way, coordinated by the services involved in the composition. In this case, the composition is called *choreography*. Both orchestration and choreography specify the coordination of the services participating in the service composition. However, compared to orchestration, choreography requires an additional effort in order to distribute the interaction over the participating services [Su et al., 2008]. In our study, we essentially consider the case of service orchestration as it represents the most widely-used pattern. Extending our study to choreographies is one of our future research directions.

## 2.2 Service-oriented Middleware

Middleware is the most common solution widely used to facilitate interoperability and coordination in the presence of dynamics and heterogeneity. As illustrated in Figure 2.2, middleware is a software layer that stands between the networked operating system and the application and provides well known reusable solutions to frequently encountered problems like heterogeneity, interoperability, security and dependability [Issarny et al., 2007]. The authors in [Issarny et al., 2007] present a survey of the different families of middleware according to their coordination model. Among them, service-oriented middleware (SOM) supports the abstraction related to SOA systems. A SOM implements a set of functionalities that are essential to enable (1) discovering

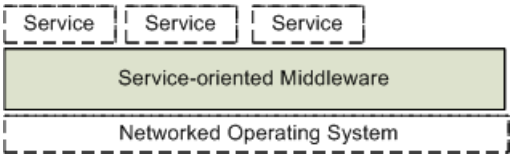



Figure 2.2: Service-oriented middleware

services, and (2) integrating them to serve users needs. *Service discovery* dynamically finds services that are available in the user environment and accesses them in order to consume their functionalities. *Service composition* enables to serve a user request by composing the functionalities of the available services.

Both service discovery and composition can be carried out by syntactically matching between the client request and the service functionality, assuming that clients and providers use a common service description syntax for denoting the service access protocols and service semantics such as in WSAMI [Issarny et al., 2005]. However, assuming that service developers and clients describe, respectively, services and service requests with identical terms cannot hold in open and heterogeneous environments such as the ones that we are considering. To cope with syntactic heterogeneity, clients and services may provide a semantic description, respectively, of their functionalities and requests. The semantic description can provided through the use of, e.g., ontologies [Singh and Huhns, 2005], which have their origins in the domain of artificial intelligence (AI). In this way, combining semantics and service-orientation, semantic-based SOM presents an efficient way to automatically and unambiguously discover, compose and consume services in heterogeneous environments, such as pervasive computing ones [Mokhtar et al., 2007].

## 2.3 Basic Service Model

In this section, we detail the notion of service and introduce a basic model that integrates the main concepts making up a service. These concepts are retrieved from the most common elements that we find in the related literature [Bell, 2008]. Using the service model, we aim at providing an accurate definition of its concepts as well as the dependencies among them, in order to depict a global view of our understanding of a service along with its constituents.

Online dictionaries give various definitions for the term *service* including "*useful labor that does not produce a tangible commodity*" [1] and "*a facility providing the public with the use of something*" [2]. In line with dictionary definitions, the authors in [Krafzig et al., 2004] underline in their definition the *autonomy* and *reusability* properties of the service functionality. They state that a service is "*a meaningful activity (of a certain complexity) that a computer program performs on request for another computer program* (...)*, i.e., a remotely accessible, self-contained application module* (...) *that is not designed for one specific customer, but instead (...) provides a functionality that is reusable in different applications*".

Other definitions emphasize the *collaboration* and *interoperability* facilities provided by a service. For instance, the authors in [Papazoglou and Georgakopoulos, 2003] state that "*services perform encapsulated business functions. Functions can be from simple request-reply to full business process interactions* (...). *They can be mixed and matched to create complete enterprise processes* (...). *They enable dynamic integration of applications across diverse technologies and between organizations*". Furthermore, this definition stresses that a service may perform complex functionalities requiring multiple interactions with the client as well as with other services.

In this thesis, the term *service* follows the above definitions, emphasizing more specifically the consumption facilities that a service provides to its clients. Inspired from [Papazoglou, 2003], we define a service as follows:

**Definition 1. Service**
A *service* is an autonomous entity that performs a single or a set of functionalities that can be consumed independently of each other. It is implemented by a software program. It is wrapped

[1] Merriam Webster's Dictionary: http://www.merriam-webster.com
[2] Dictionary.com: http://dictionary.reference.com

within a formal *service description* that is well known, and known how to be used not only by the service designers, but also by entities (i.e., other services and clients) that do not know about how the service has been designed and implemented, and yet want to access and use it.

In the above definition, the *service description* is a fundamental concept making up a service. It is used to advertise the description of (1) the service interface, (2) the service capabilities, (3) their expected behaviors, and (4) the service quality. Figure 2.3 represents a class diagram for the service model, highlighting the basic concepts of a service along with their inter-dependencies. This model serves as a basis that will be enriched all along this document as new concepts definitions are required.

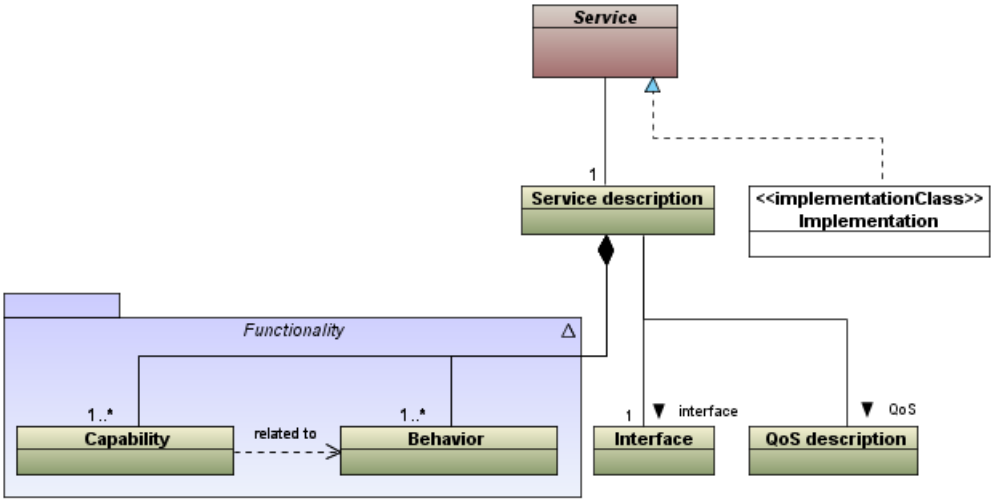



Figure 2.3: Service model

**Service interface** The description of the *service interface* publishes the service signature, which consists of the available operations, their input/output/error parameters, the data-types and the access protocols, in a way that other software entities (i.e., clients or services) can determine what the service does, how to invoke its operation(s), and what results to expect in return. In addition, the service interface may be enriched with a semantic description of its operations as well as their in/out parameters in order to provide them with richer description that goes beyond the limits of the syntactic description.

In this way, the service interface can be modeled as illustrated in Figure 2.4. It essentially includes the descriptions of the operations that the service provides, where an operation description is composed out of two elements: a semantic description and a signature. The service interface includes other concepts, including the binding and transport protocols; these concepts are oriented to the access and communication issues, and are not relevant to the issue of modeling the service interface.

**Service functionality** The service interface is able to describe only simple functionalities that are described using a single operation, and which require a single request/response exchange to perform a unit of work. Complex functionalities that require multi-step exchange (i.e., a conversation) between the service and the client, and which involve the invocation of multiple operations, cannot be described in the service interface. Hence, the service description is extended

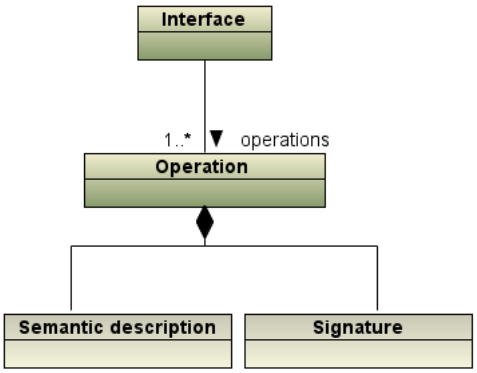



Figure 2.4: Service interface

in two ways to represent both complex and simple service functionalities: service capability and service behavior.

**Service capability** The first way provides the description of a *service capability* which states the conceptual purpose of a service functionality, its required data and its expected results, by using terms or concepts defined in an application-specific or commonly agreed upon taxonomy, using, e.g., WordNet [3]. Using such taxonomy, services such as "Voyages-SNCF" [4] may be defined as instances of concepts that represent their capabilities. In its simplest form, a capability may be realized by a single operation of the service interface.

Inspired from the OWL-S specification [W3C, 2004a], we model a capability description (in Figure 2.5) as a composition of three concepts: the *functional purpose*, (0..*) *required inputs* and (0..*) *provided outputs*, where the functional purpose of the service functionality represents the transformation that the service produces, which results in the production of outputs from a set of inputs.

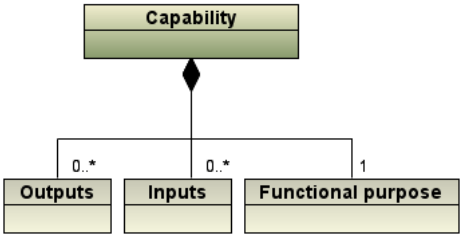



Figure 2.5: Modeling a capability

**Service behavior** The second way to represent a functionality is to describe its *behavior*. The behavior defines how a system entity changes over time [OMG, 2001]. It represents the business logic that realizes the functionality. As in the Model Driven Architecture (MDA [OMG, 2001]) defined by the OMG [5], we model the service behavior as a workflow/process that defines the set of operations realizing the functionality, their order of execution (i.e., the control flow), along with their data dependencies (i.e., the data flow). For example, the behavior associated with

[3] http://wordnet.princeton.edu/
[4] Voyages-SNCF is a French online travel agency, http://www.voyages-sncf.com/
[5] Object Management Group.

the "train ticket booking" capability can be set as follows. It includes a first operation that requests for the departure and destination places, and the departure and arrival dates. Then, a number of possibilities are proposed to the user, from which s/he selects a train that fits her/his schedule. Afterwards, another operation requests for a valid credit card number, and produces a state transition where (1) the train ticket will be edited, then (2) the credit card will be charged. We introduce in Figure 2.6 a model for service behaviors that integrates its basic concepts.

We distinguish in our service model three features for the service behaviors that have impact on the behavior execution. These features are: (a) the *structure* of the behavior, (b) its *pre- and post-conditions*, which have to be satisfied respectively by the user inputs and the service outputs, and (c) potentially the *service state* maintained during the behavior execution.

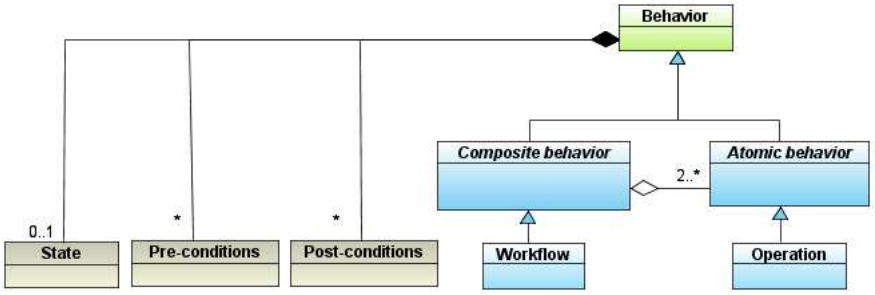



Figure 2.6: Modeling a service behavior

Using the terminology of the OWL-S [W3C, 2004a] specification, behaviors that are composed of a single operation are *atomic*, and those that compose more than one operation are *composite*. We model composite behaviors as an aggregation of (2..*) atomic behaviors.

In addition, a behavior may define (0..*) of pre- and post-conditions (Figure 2.6), which are set to ensure the consistency of the behavior execution. *Pre-conditions* are logical formulæ that need to be satisfied (ensured to be true) by a client before the execution of the behavior. Examples of pre-conditions are that a credit card should be valid, or that the account should not be overdrawn, and so forth. The execution of a behavior may also result in changes of the state of the world, which are called post-conditions or effects. *Post-conditions* are logical formulæ that state what will be true upon successful execution of the behavior. An example of post-condition is the process/behavior that charges a credit card. As a result of the execution of the process, a credit card is charged and the money in the account reduced.

Finally, the behavior execution of a functionality can be stateless or stateful. A service provides a *stateless* behavior, if the behavior can be performed without requiring that the service maintains a context or state; the service does not hold any state across or after the interaction with a client when executing the behavior. Each message sent by the client to the service must contain all necessary information for the latter to process it.

In contrast to stateless behaviors, if the execution of a behavior requires its context to be preserved, then the behavior is called *stateful*. A service that provides a *stateful* behavior maintains a state before, during, and/or after the behavior execution. A composite behavior is a typical example of a stateful behavior. It involves the exchange of the messages with the client in order to execute the operations the behavior composes. Hereafter, we define the service state associated to a given service instance that executes a stateful behavior.

**Definition 2. Service state**
Assuming that the execution of the service behavior has started at a time $t$, the *service state* at a

time ($t+\Delta t$) includes all the data that the behavior execution generates and/or uses during $\Delta t$. In the case that multiple instances of the service are running for multiple clients, we consider the state of each instance separately from the rest of other instances. We define the state of a service executing a stateful behavior into three parts, which may overlap:

1. The data related to the interaction with the specific client of the running instance, i.e., all the data that are included in the exchanged messages with the client as well as intermediate data used in the workflow. We call this part, the *workflow* or *observable state* of the service.

2. The data that are maintained internally by the service implementation, such as program counters and alike. This part includes all the temporary data the behavior execution generates in the time interval $[t, t + \Delta t]$, and which are required in order to achieve the behavior execution from $(t + \Delta t)$ till its completion. We call this part, the *implementation state* of the service.

3. The external data that are used and/or manipulated during the behavior execution. This part includes all the data that are potentially shared with the running instances of the behavior, as well as, with external entities to the service (i.e., other services and clients). As in shared databases, using and manipulating these data are governed by the service internal logic using specific rules, such as mutual exclusion (mutex) and locks. We call this part the *resources state* of the service as the shared data can be considered as external resources that the implementation manipulates. As each service instance is concerned with a subpart of the external resources, we consider that the resources state as a limited view of the overall shared resources of the service, which includes only the information required by the running instance.

At the instantiation of the service, the observable and implementation states are initialized with the user inputs. The resources state is included in the service state only when external (potentially shared) data are required and used at runtime.

**Service model** The two ways of modeling functionalities (i.e., using capabilities and behaviors) are complementary. The description of a capability helps to select a service provider that complies with a *user request*. It facilitates the discovery process since the matching process between a user request and a capability description is reduced to comparing concepts of a given taxonomy. The description of a behavior shows how a service produces its results. It enables to foster the correct consumption of the service functionalities by invoking the service operations in the correct order with respect to data dependencies between invocations.

Taking advantage of the two representations of the service functionalities, we integrate both of them in our service model using the concepts "*capability*" and "*behavior*" in order to provide a complete description of the service functionalities. A capability is associated with the behavior through the dependency 'related to', to denote that the two representations are related to the same functionality (as modeled in Figure 2.7).

The service model integrates the concepts that are included in the capability and behavior descriptions, presented respectively in Figures 2.5 and 2.6. In particular, the behavior description can be of two types: a workflow description in the case of composite behavior, or an operation in the case of atomic behavior. The semantic description of an operation represents aspecification of its capability. The behavior includes (0..*) pre- and post-conditions, and a state description in the case of stateful behavior.

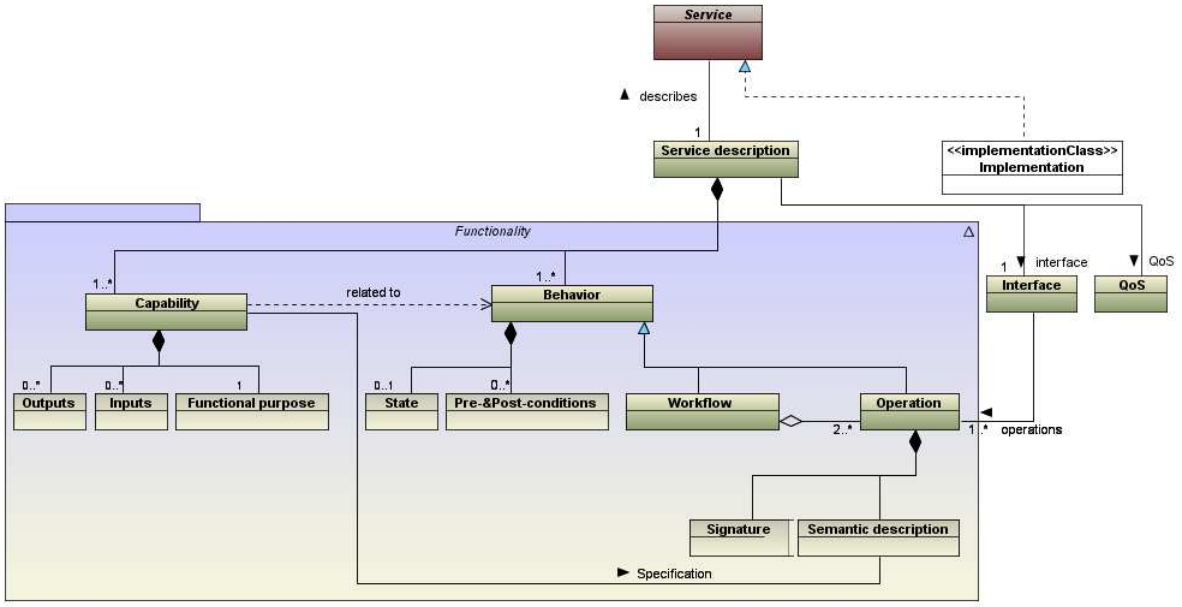



Figure 2.7: Integrating capability and behavior concepts in the service model

Finally, the service description may be extended with a *Quality of Service* (QoS) description, which publishes important non-functional service quality attributes, such as service cost, performance metrics (e.g., response time), security attributes, transactional properties, reliability, scalability, and availability [Glinz, 2007].

## 2.4 Web Service Description Languages

Web services (WS) is the *de facto* realization of the SOA paradigm. They define a model for service-oriented distributed computing, in which services are described with and interact by exchanging XML documents. In addition, Web service standards are evolving rapidly to provide a foundation for interoperation between Web services. Web service description is based on a number of standardized languages, each of them focusing on a specific concept of the service model introduced in the previous section. It typically includes the interface definition and the transport-level properties, both specified in the Web Service Description Language (WSDL [W3C, 2007c]). It further includes the behavior description [Bultan et al., 2003, Benatallah et al., 2004, Benatallah et al., 2003]. Behaviors can be specified using the Web Services Business Process Execution Language (WS-BPEL [OASIS, 2007]) or any of the many other formalisms developed for this purpose, such as the Web Service Conversation Language (WSCL [6]). In particular, WS-BPEL is an industrial standard that is supported by a significant number of tools in the domain of Web services. As service behaviors become more complex, maintaining and manipulating a state across multiple message exchanges with external entities (i.e., clients and services), the Web Service Resource Framework (WSRF [OASIS, 2006b]) has been standardized in order to describe the service state when services require to maintain a state across multiple interactions with a client. WSRF further provides standardized means to manipulate the whole or a part of the service state.

[6] WSCL: http://www.w3.org/TR/wscl10/

Coupled with WS description languages, Semantic Web technologies [Berners-Lee et al., 2001] proliferate, as the need for rich semantic specifications of Web services grows. They enable fuller automation of service provision and use, based on well-founded semantic reasoning about services. The Web Ontology Language (OWL [W3C, 2004b]) is considered as one of the fundamental technologies underpinning the Semantic Web. In order to ensure correct consumption of service functionalities, Semantic Web Rule Language (SWRL [W3C, 2004c]) enable to describe the service rules such as the pre- and post conditions of the service behaviors. In this way, combined with the syntactic Web service description languages, the semantic description of Web services enables automated service selection, and the translation of message content between heterogeneous interoperating services. It also makes easier service composition, service monitoring and recovery from failures.

In the following sections, we present the current background on a set of Web service languages that serve the purpose of this thesis. Section 2.4.1 briefly presents OWL, and illustrates its use through a train ticket booking example. Section 2.4.2 presents the use of OWL semantic relationships between the semantic concepts, in representing and matching between a user request and service capabilities. Section 2.4.3 presents WSDL and its extension with semantic annotations. Section 2.4.4 provides an overview of WS-BPEL and its use to describe service behaviors as well as service compositions. Section 2.4.5 presents WSRF in order to illustrate how a service state can be described, and our extension of WSRF with semantic annotations. Finally, Section 2.4.6 briefly presents SWRL for semantically describing the pre-and post-conditions of service behaviors.

### 2.4.1 OWL

The semantic Web paradigm adds a machine-interpretable semantics to the current Web by referring to hierarchically structured vocabularies of terms, i.e., ontologies, representing a specific area of knowledge. Since the 1990s, a number of research efforts have explored how the knowledge representation (KR) from AI [7] could be made useful on the World Wide Web. These included languages based on HTML (called SHOE), XML (called XOL, later OIL), and various frame-based KR languages (e.g., FLogic, OKBC, and KM) and knowledge acquisition approaches [Corcho and Gómez-Pérez, 2000, Gómez-Pérez and Corcho, 2002]. Ontology languages such as the W3C-recommended Web Ontology Language (OWL [W3C, 2004b]) support formal descriptions and machine reasoning on concept hierarchies. OWL is a language for making ontological statements, developed as a follow-on of (1) RDF and RDF Schema (RDF-S) by providing richer vocabulary along with a formal semantics, as well as, (2) earlier ontology language projects including OIL, DAML and DAML+OIL. An OWL ontology comprises classes, individuals and properties, where a class represents a concept, an individual represents an instance of a class, and a property represents a relation between classes or individuals.

To illustrate the use of ontologies in a service description using WS languages, we employ in the rest of this chapter the service for train ticket booking, picked from the scenario presented in the previous chapter.

*A train ticket booking service enables users to book a train ticket according to their preferences such as, journey type (i.e., one-way or round-trip), seat class (i.e., business or economy) and side (i.e., window or aisle), and alike. Let's suppose that a user tries to book a round-trip train ticket using a service providing train tickets to his/her specific destination. S/he would like an economy class seat, located at the window side.*

[7] Artificial Intelligence.

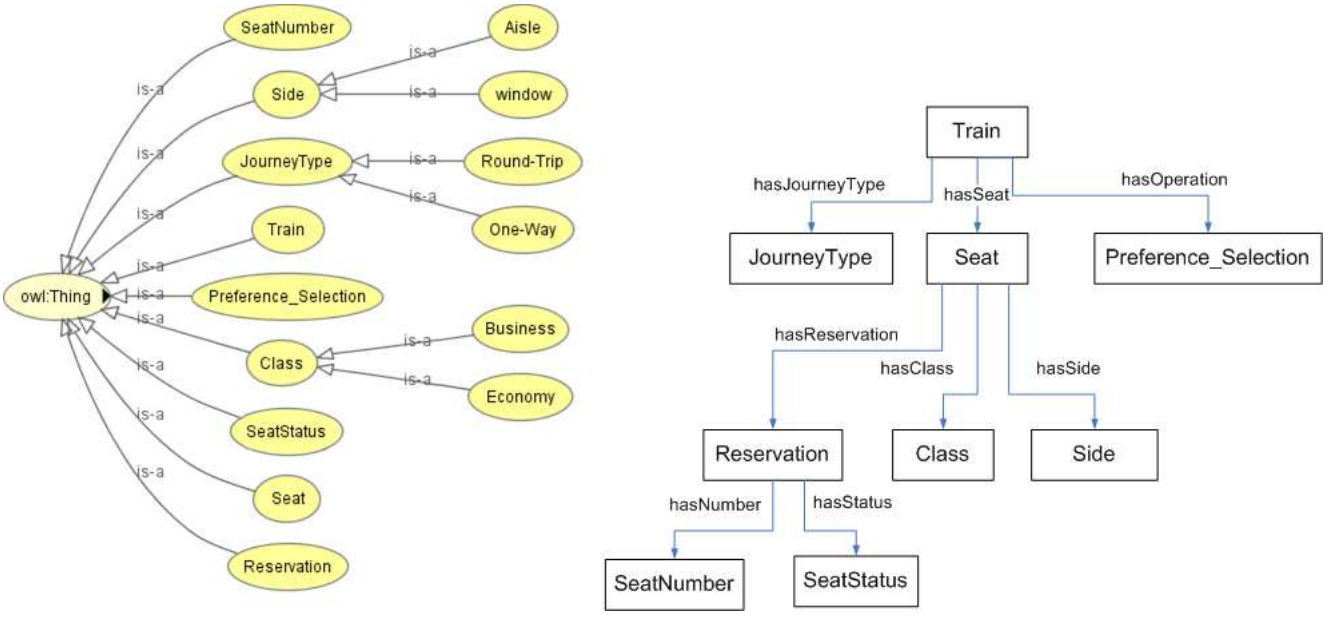



Figure 2.8: Ontology for train ticket booking

We consider an ontology for train ticket booking that takes into consideration the users' preferences. Figure 2.8 depicts the hierarchy of the train ticket booking ontology [8] and the properties connecting its classes. The 'Train' class is connected to 'Preference_Selection' by the object property 'hasOperation' in order to declare the operations that enable users to select their preferences. The 'Train' class is also connected to the 'Seat' and 'JourneyType' classes by the object properties 'hasSeat' and 'hasJourneyType'. The 'Seat' class is connected to the 'Class', 'Side' and 'Reservation' classes by using, respectively, the properties 'hasClass', 'hasSide' and 'hasReservation'. The property 'hasReservation' associates to a 'Seat' a 'Reservation' that includes a 'SeatNumber' and 'SeatStatus' using, respectively, the properties 'hasNumber' and 'hasStatus'. The instance of the seat status determines whether the seat is booked or not.

OWL is used by almost all the WS description languages that we present in the following in order to provide, in addition to the syntactic description, a semantic description of the concepts to which they are dedicated. Among their several benefits, OWL ontologies can be used to describe service capabilities. Hereafter, we illustrate how we use OWL ontologies to describe service capabilities. This representation serves as a basis for matching between a user request and a service capability.

### 2.4.2 Matching Service Capabilities

As mentioned in Section 2.3, a capability describes the conceptual purpose of a service functionality, its required data as input and its expected results as outputs. These elements can be described using semantic concepts included in OWL ontologies. For example, the inputs of a functionality that enables to select user preferences for a train seat should include an instance of the class 'Side' and an instance of the class 'Class' that are in a relationship with the train ticket being booked (hasSeat). The outputs should be an instance of the class 'Reservation'. The functional description of a functionality that enables to select user preferences for a seat should

[8] Created using Protégé, a tool enabling to create, edit and export OWL ontologies. For more details visit http://protege.stanford.edu/

be an instance of the class 'Preference_Selection' included in the 'Train' ontology. Hence, the capability description of a functionality that enables to select user preferences for a seat includes three elements:

1. Functional purpose: 'Preference_Selection'
2. Inputs: 'Side' and 'Class'
3. Output: 'Reservation'

*Capability matching* then compares the capabilities advertised by services with the capabilities needed by the requester. The goal is to find the advertiser that produces the results required for the client. In general, it hardly happens that capability offered will exactly match the client request. Several algorithms [Guo et al., 2005b, Paolucci et al., 2002] have been proposed in order to compute the degree of matching between a requested capability and a provided one. Different degrees of match are detected depending on whether the advertised capability and the requested one describe the same capability or whether one subsumes the other [Ben Mokhtar, 2007]. Going one step further, other efforts have been focusing on matching capabilities expressed using heterogeneous ontologies [Guo et al., 2005a].

In the present document, we take advantage of the existing efforts that enable to match between a user-requested capability and a service-provided one. We concentrate on the relationships that define similarities between the capabilities in order to enlarge the scope of search for the services that may serve a user request. In the following, we briefly present the different semantic relationships supported by OWL [W3C, 2004b]. We then gather these relationships into a single relation denoting the semantic inclusion between semantic concepts.

#### Background on OWL-supported relationships

To serve a user request, the service's provided capability has to be semantically similar to the user's requested one. To denote similarities over semantic concepts, OWL supports a set of relationships that are presented in the following section.

To present the relationships over semantic concepts, we consider a finite set of semantic concepts $\zeta$ across a finite set of ontologies $\theta$. In the present thesis, we will consider the case where $\theta$ is reduced to a single ontology. Nevertheless, relevant efforts have been investigated to connect and merge the concepts of multiple ontologies. These efforts can be integrated in our work in order to enlarge the scope of the semantic relationships between semantic concepts. Furthermore, we assume that both $\zeta$ and $\theta$ are described using OWL [W3C, 2004b].

- *Instance-concept relationship.*
  In OWL, a concept $C \in \zeta$ is represented as a class. The instances of a class (i.e., objects) are represented as individuals. The instance-concept relationship (denoted $I\text{-}C$) associates the individuals with the class they instantiate. For instance, a 'Train seat' is an instance of the class 'Seat'. We denote the relation $C_\tau \xrightarrow{I\text{-}C} C : \zeta \times \zeta$ between two semantic concepts, defining that $C_\tau$ is an instance of $C$. In practice, the instance-concept relationship is described in the class of the concept $C_\tau$ using the built-in OWL property *owl:oneOf*.

- *Equivalence relationship.*
  A concept $C_\tau$ is equivalent to a concept $C_\sigma$ (denoted $\simeq$) if and only if each instance of the concept $C_\tau$ is an instance of the concept $C_\sigma$ and *vice versa*. In practice, the built-in OWL property *owl:equivalentClass* between two semantic concepts $C_\tau$ and $C_\sigma$ denotes

their semantic equivalence. A special case of equivalence is the identity. The built-in OWL property *owl:sameAs* (denoted $=$) links two individuals, denoting that they are identical.

- *Hierarchical or subsumption relationship.*
  A hierarchical relationship, also known as subsumption or subtyping relationship, takes place between classes corresponding to two concepts, when the classes of the two concepts are linked using a set of properties, such as 'is a', 'can be', 'type of'. In OWL, its meaning is exactly the same: if the class description $C_\sigma$ is defined as a superclass of class description $C_\tau$, then the set of individuals in the class extension of $C_\tau$ should be a subset of the set of individuals in the class extension of $C_\sigma$. A class is by definition a subclass of itself (as the subset may be the complete set). We denote the relation $C_\tau \xrightarrow{SuperClass} C_\sigma : \zeta \times \zeta$ between two semantic concepts, defining that $C_\sigma$ is a superclass of $C_\tau$. In practice, the hierarchical relationship is described in the class of the concept $C_\tau$ using the construct *rdfs:subClassOf*.

- *Compositional relationship.*
  A composition relationship, also known as a part-whole relation [9], occurs between two distinct concepts when one is a part, or component, of the other. For instance, considering the 'Train' ontology, a seat is composed of 'reservation', 'class' and 'side'. We denote $C_\tau \xrightarrow{Comp} C_\sigma : \zeta \times \zeta$, defining that $C_\tau$ is a component of $C_\sigma$. In practice, the compositional relationship is described in the class of the concept $C_\tau$ using the built-in OWL property *owl:unionOf*.

Other relationships may be defined between two concepts such as the complement relationship, where each instance of the one is not an instance of the other and *vice versa*, and the union of the two concepts constitutes the universe. Also, the intersection relationship defines the common properties between two concepts, but it does not guarantee that all the properties are preserved by the included concept. These relationships do not serve our need for identifying similarities between semantic concepts, and thus are out of the scope of our interest.

### Semantic inclusion

Several efforts have been focusing on semantic service discovery and selection, where the service selection is based on discovering a service that offers a capability with the same OWL class as the requested one, or an equivalent OWL class. In addition to the equivalence over semantic concepts of the required and provided capabilities, some effort (e.g., [Ben Mokhtar, 2007]) support the subsumption relationship between the provided and required capabilities. However, extending discovery to other relationships than equivalence or subsumption increases the chances to find a substitute service.

Following this direction, we go one step further by allowing more flexible semantic relationships than the semantic equivalence and subsumption between the concepts of the user-requested and service-provided capabilities. In order to take fully advantage of the richness of the environment in terms of available capabilities, we allow semantic inclusion between semantic concepts (denoted $\subseteq^s$). The semantic inclusion is realized when a capability is '*a part of*' a more *generic* capability. For instance, considering the train ticket booking scenario, the semantic inclusion enables to select for a 'train ticket reservation' user requested capability, a richer capability such as, 'TrainTicketReservation&Weather' capability that enables to book a seat in a train while providing information details on the weather forecast. Indeed, we assume that if a service provides

[9] Simple part-whole relations in OWL Ontologies. W3C Editor's Draft 11 Aug 2005, available at http://www.w3.org/2001/sw/BestPractices/OEP/SimplePartWhole/

more than required, its candidacy for service selection is worth to be considered in case that no equivalent capability is available. Hereafter, we define the semantic inclusion relationship that we consider over semantic concepts.

**Definition 3. Semantic inclusion over semantic concepts $C_\tau$ and $C_\sigma$** (denoted $C_\tau \subseteq^s C_\sigma$) if for each property or attribute of the class $C_\tau$, $C_\sigma$ defines the same or equivalent property or attribute.

The instance-concept, hierarchical, compositional and equivalence relationships are examples of a semantic inclusion. We have: $\forall R \in \{I\text{-}C, SuperClass, Comp, \simeq, =\}, C_\tau \xrightarrow{R} C_\sigma \Rightarrow C_\tau \subseteq^s C_\sigma$. Note that the semantic inclusion relation over semantic concepts is a partial order.

#### $\subseteq^s$ is a partial order

A partial order is a binary relation over the set of concepts $\zeta$. $\forall C_\tau, C_\sigma, C_\gamma$ three distinct semantic concepts or instances of semantic concepts included in $\zeta$, $\subseteq^s$ is

1. reflexive: $C_\tau = C_\tau \Rightarrow C_\tau \subseteq^s C_\tau$.
2. antisymmetric: if $(C_\tau \subseteq^s C_\sigma) \wedge (C_\sigma \subseteq^s C_\tau)$, then $C_\tau$ is equivalent to $C_\sigma$, as $(C_\tau \subseteq^s C_\sigma)$ implies that $C_\sigma$ defines all the properties and attributes of $C_\tau$, and $(C_\sigma \subseteq^s C_\tau)$ implies that $C_\tau$ defines all the properties and attributes of $C_\sigma$. Hence, $C_\tau$ and $C_\sigma$ define the same properties and attributes, thus they are equivalent.
3. transitive: if $(C_\tau \subseteq^s C_\sigma) \wedge (C_\sigma \subseteq^s C_\gamma)$, then $C_\tau \subseteq^s C_\gamma$ , as $(C_\tau \subseteq^s C_\sigma)$ implies that $C_\sigma$ defines all the properties and attributes of $C_\tau$, and $(C_\sigma \subseteq^s C_\gamma)$ implies that $C_\gamma$ defines all the properties and attributes of $C_\sigma$, and thus all the properties and attributes of $C_\tau$.

The advantage of the partial order relation is to infer semantic inclusion between concepts that are not necessarily in direct relationship, increasing thereby the chances to serve a user request.

### 2.4.3 SAWSDL

Web services are described using the Web Service Description Language [W3C, 2007c] (WSDL). WSDL is an XML-based document describing a service that offers a set of operations through the exchange of messages. The operations and messages are described syntactically, and then bound to a concrete network protocol and message format to define an endpoint. However, as the WSDL description is purely syntactic, two services may have similar descriptions while meaning totally different things, or they may have very different descriptions, and yet similar meaning. To resolve such ambiguities, the W3C has standardized the Semantic Annotations for WSDL and XML [10] Schema (SAWSDL) [W3C, 2007a]. A semantic annotation in a document (e.g., WSDL or XML schema) is additional information that identifies an OWL concept in an ontology (i.e., *semantic model*, to use SAWSDL taxonomy) in order to describe a part of that document. Semantic annotations are of two kinds: (1) explicit identifiers of OWL concepts, and (2) identifiers of mappings from WSDL to concepts or *vice versa*.

The former enables to describe XML elements with a reference to a semantic concept. The latter relates the data defined by an instance of an XML schema document with semantic data defined by a semantic model. A graphical representation of the types of annotations is provided in Figure 2.9. We detail these two types in the following.

[10] XML schema: http://www.w3.org/XML/Schema

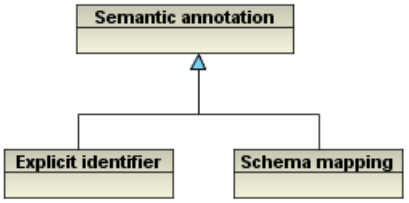



Figure 2.9: Semantic annotations in SAWSDL

### Explicit identifiers

Explicit identifiers specify the association of a concept in some semantic model with WSDL elements, such as operations, in/out parameters, and in/out messages. This association is useful when considering the operations as functionalities with atomic behaviors; the explicit identifiers then enable to describe the capability of each operation.

The association is specified using an additional attribute to the syntactic description of WSDL elements, denoted $< sawsdl : modelReference... >$. This attribute makes reference to the semantic concept related to the WSDL element. The reference is made explicit through a *URI* [11], which is a concatenation of (1) the URI of the ontology, (2) the '#' character, and (3) the name of the concept in the ontology.

```
1  <wsdl:definitions of a service A enabling train ticket booking...>
   ...
3  <xs:element name="SeatPreferences">
     <xs:complexType>
       <xs:all>
6        <xs:element name="class" type="integer"
7         sawsdl:modelReference="http://URI_Train_Ontology/Seat#Class"/>
8        <xs:element name="side" type="integer"
9         sawsdl:modelReference="http://URI_Train_Ontology/Seat#Side"/> ...
       </xs:all>
     </xs:complexType>
   </xs:element>
   ...
14 <wsdl:message name="Input">
15   <wsdl:part name="body" element="xsd:SeatPreferences"
16    sawsdl:modelReference="http://URI_Train_Ontology/Train#Seat"/>
   </wsdl:message>
   ...
19 <wsdl:operation name="SelectTypeOfSeat"
20  sawsdl:modelReference= "http://URI_Train_Ontology/Train#Preference_Selection">
21   <wsdl:input message="tns:SeatPreferences"/>
     <wsdl:output message=.../>
   </wsdl:operation>
   ... </wsdl:definitions>
```

Figure 2.10: Web service (Service A) for train ticket booking

For example, consider a Web service (denoted *Service A*) for train ticket booking. *Service A* provides –among others– an operation that enables to select the clients' preferences, advertised in *Service A*'s SAWSDL. The listing in Figure 2.10 describes the related part of the SAWSDL

[11] Uniform Resource Identifier.

description of *Service A*. *Service A* declares the operation 'SelectTypeOfSeat' (Line 19) that enables a client to select his/her seat class and side. Using the train ticket booking ontology (described in Section 2.4.1), the 'SelectTypeOfSeat' operation is annotated with the semantic concept '$Train\#Preference\_Selection$' (Line 20). Furthermore, the operation input, i.e., 'SeatPreferences' (Line 15) is annotated with '$Train\#Seat$' (Line 16). The XML element 'SeatPreferences' is composed out of two parameters namely, class (Line 6) and side (Line 8), respectively annotated with the concepts '$Seat\#Class$' and '$Seat\#Side$' (Lines 7, 9).

These annotations are useful for understanding and reasoning on the elements described in the WSDL. They enable, for instance, to match between the semantics of the user-provided data and the service-required ones. However, in case of syntactic mismatch, transformations in data representation are required. They are enabled using schema mappings.

### Schema mappings

Explicit identifiers can be used to help determining if a service meets the requirements of a client. Still, there may be mismatches between the semantic concept representation and the structural representation of the in/out parameters. To resolve these mismatches, the idea is to map between the representations of the in/out parameters and their related semantic concepts. The mapping should transform the syntactic representation of any in/out parameter into a representation of its related semantic concept. As in/out parameters are represented using XML schemas, semantic concepts should also be represented using the same standard, so as to ease the mapping mechanism. In this way, when in/out parameters and semantic concepts are represented using XML schemas, the mapping can be realized using XML Style Sheet Transformation (XSLT) [W3C, 1999], which is a language for transforming XML documents into other XML documents.

To enable this mapping, SAWSDL provides means to make reference to XSLT documents that transform XML schemas of the operations' parameters into an instance of the semantic concept to which they correspond and *vice versa*, using *schema mappings*. There are two mechanisms for schema mappings, namely, lifting and lowering. *Lifting schema mapping* transforms XML data into instances of a semantic concept, and *lowering schema mapping* does the opposite; it transforms instances of a semantic concept into an instance of an XML schema. To illustrate the schema mapping mechanisms, we employ our train ticket booking scenario.

As explained above, semantic concepts need to be represented using XML schemas in order to enable lifting and lowering mechanisms. Hence, after having introduced the ontology in Section 2.4.1, we describe each class using an XML schema.

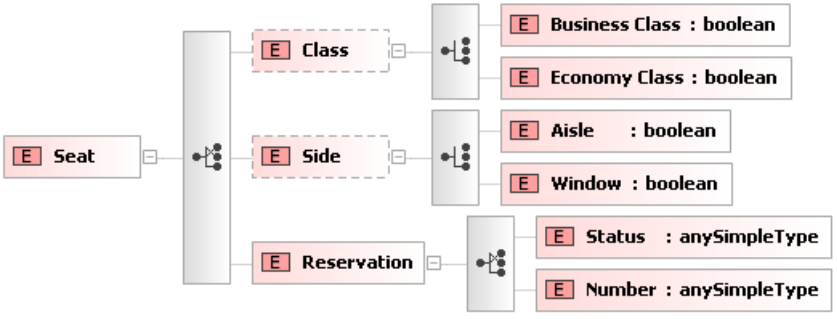



Figure 2.11: XML schema of the concept 'Seat'

For instance, we consider the concept 'Seat' as a complex element composed out of three

elements 'Class', 'Side' and 'Reservation', as represented in Figure 2.11. For each individual of the class 'Seat', these simple elements have to be specified at most once, in a random order. The 'Class' and 'Side' are complex XML elements that include a choice between two elements. The element 'Class' includes either a 'Business Class' simple element, or an 'Economy Class' one. Similarly, the element 'Side' includes either a 'Window' simple element, or an 'Aisle' one. The values type of the simple elements is set to 'boolean', as they can be either the one or the other but not both at the same time. The 'Reservation' complex element is composed of a sequence of two simple elements: 'Status' and 'Number'. The value type of these simple elements can be 'anySimpleType' in order to not restrict the class instantiation.

In this way, using lifting mechanism, we can transform the input parameter 'SeatPreferences' of the operation 'SelectTypeOfSeat' (described in Figure 2.10) into an instance of the XML schema of the 'Seat' concept described (described in Figure 2.11). This mapping is useful, for example, when targeting to replace the invocation of the operation 'SelectTypeOfSeat' (provided by *Service A*) by a semantically equivalent operation provided by another service. By semantically equivalent, we mean that the operations are annotated with the same semantic concept picked from the same ontology. We explain hereafter the details of lifting an lowering mechanisms.

Figure 2.12 depicts the SAWSDL descriptions of two services that enable train ticket booking namely, *Service A* and *Service B*. *Service A* and *Service B* respectively declare the operations 'SelectTypeOfSeat' and 'SelectSeatType', both referencing the same semantic concept '$Preference\_Selection$' from the ontology for the train ticket booking, presented in Figure 2.8. However, the respective operations of *Service A* and *Service B* require different types of input parameters, while both of them make reference the same semantic concept (i.e., 'Seat').

Mapping from the inputs of the operation 'SelectTypeOfSeat' to the ones of 'SelectSeatType' is possible by applying XSLT transformations defined and provided by the developers of (*Service A*) and (*Service B*), respectively through the XSLT scripts: $A2Seat$ and $Seat2B$. As illustrated in Figure 2.12, the schema mapping includes a structure transformation and a type conversion between the input of *Service A*'s operation and the XML schema of the concept '*Seat*', and inversely for the input of *Service B*. The input of *Service A*'s operation is a list of integers, while *Service B*'s one is a string concatenating the two parameters. The transformation is performed as follows.

The lifting technique consists in associating each attribute of the input parameter 'SeatPreferences' of the operation 'SelectTypeOfSeat', with the corresponding XML element in the schema of the semantic concept '*Seat*' (Figure 2.11). The lowering technique consists in concatenating the values of the attributes of 'SeatPreferences' within a same string.

In addition, we notice that *Service A* does not require as input the parameter 'Reservation' of the semantic concept *Seat*', neither does *Service B*. Therefore, the mapping does not introduce any further complexity than the structure transformation and the type conversion. However, considering the case that the operation 'SelectSeatType' of *Service B* requires an extra parameter, lifting and lowering are not sufficient to perform an automated mapping. This can be solved by assigning default values to the attributes of a semantic concept, or by a third party (e.g., the user) intervention.

### 2.4.4 BPEL

The Business Process Execution Language for Web Services [OASIS, 2007] (WS-BPEL 2.0 or BPEL for short) is an OASIS standard that has emerged from the earlier proposed XLANG [Microsoft, 2001] and Web Service Flow Language (WSFL) [Group, 2001]. It enables the construction of complex

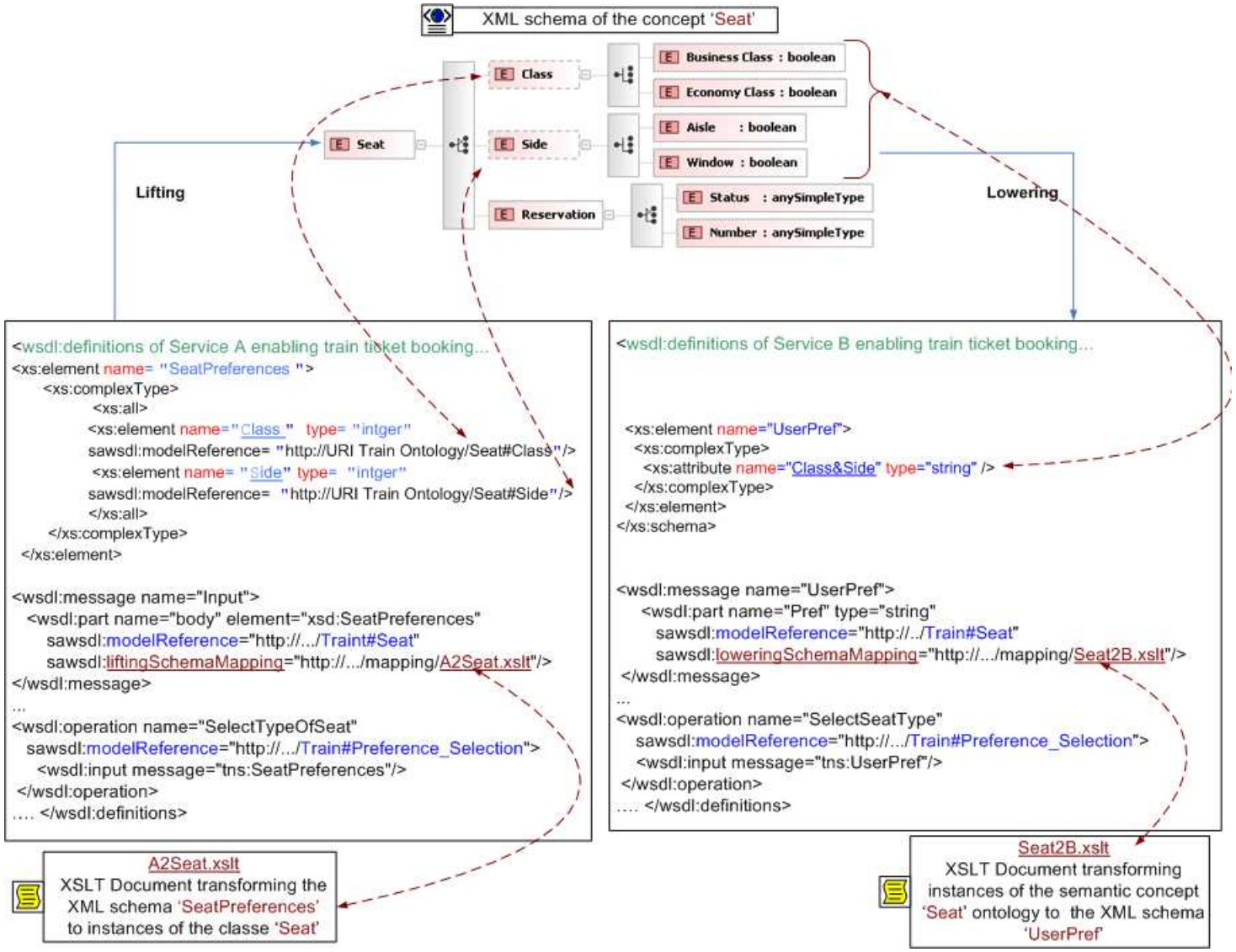



Figure 2.12: Schema mapping

Web services (implementing composite behaviors) by composing the operations of one or more Web services, which act as the basic activities in the process of the newly constructed service.

BPEL is commonly used for describing workflows of composite services, and can be executed by a process execution engine in order to provide the functionality of the composite service to a client. In BPEL, all entities orchestrated in a workflow are seen as "partners". At runtime, partners are mapped to actual service instances by the workflow-enactment engine. Access to the process is exposed by the execution engine through a Web service interface (e.g., SAWSDL description), allowing such processes to be accessed by the clients, or to act as basic activities in other process specifications. BPEL features several basic activities which allow interacting with the services being arranged in the workflow. These activities are: (1) $< invoke >$ activity, which allows the business process to invoke a one-way or request-response operation, (2) $< receive >$ activity, which allows the business process to wait for a matching message to arrive , and (3) $< reply >$ activity, which allows the business process to send a message in reply to a message that was received by a $< receive >$ activity. Furthermore, it is possible to wait for some time ($< wait >$), terminate the execution of the workflow instance ($< terminate >$ activity), copy data ($< assign >$) from one message to another using *variables*, which enables to define the data flow, announce errors ($< throw >$), or just to do noting ($< empty >$ activity).

To allow the composition of operations, a variety of structured activities exists. The $<$ *sequence* $>$ activity offers the ability to define ordered sequences of activities, the $< flow >$ activity executes a collection of activities in parallel. The $< switch >$ activity allows branching, $< pick >$ allows to execute one of several alternative paths, and loops can be defined using the $< while >$ activity. Furthermore, BPEL includes the feature of $<$ *scoping* $>$ activities and specifying *fault handlers* and *compensation handlers* for scopes. Faults handlers get executed when exceptions occur, for instance, through the execution of the mentioned $< throw >$ activity. With a compensation handler, BPEL enable to define a set of activities that have to be executed when there is a problem in the process. The compensation handler can be started from the process itself to undo certain activities (included in a scope) that have already been completed.

In the present thesis, we use BPEL for describing the workflows of composite behaviors as well as the ones of composite services. To illustrate the use of BPEL process, we employ first the the previously-introduced train ticket booking service to describe its composite behavior, then we present how it can be combined with a hotel booking service in order to realize a service composition for travel booking arrangements.

#### Use of BPEL for describing composite behaviors

As aforementioned, service functionalities implement composite behaviors. Using BPEL processes, a service advertises the description of its composite behaviors to foster the correct consumption of its functionalities.

For example, consider the train ticket booking *Service A*, the SAWSDL of which is described in Section 2.4.3. *Service A* enables the clients to book their train tickets according to their preferences. Besides providing the operation 'SelectTypeOfSeat', *Service A* offers a more complex functionality for train ticket booking, which enables a client to select, confirm and then pay for his/her train tickets. This functionality is implemented using a composite behavior that includes three operations, namely 'SelectTypeOfSeat', 'Confirm' and 'Pay'. These operations have to be performed in a specific order. A graphical representation of the 'train ticket booking' behavior is presented in Figure 2.13.

First, a client requesting for a train ticket booking triggers the process execution by providing his/her inputs. Then, after selecting the client preferences using the operation 'SelectTypeOf-

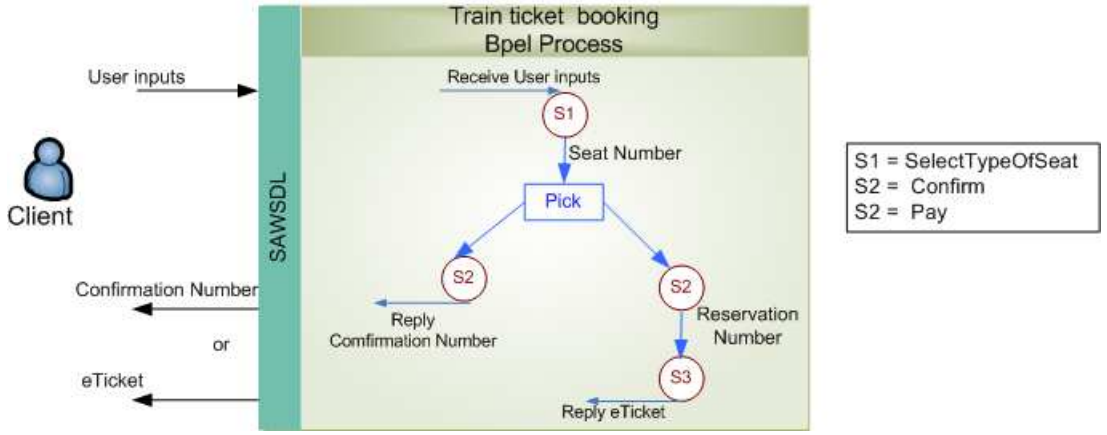



Figure 2.13: Graphical representation of a composite behavior

Seat', the service books a seat on a specific train according to the seats availability. This is enabled using the operation 'Confirm'. The seats status are centralized and stored within a database managing the concurrent client bookings. Finally, the service enables the client (1) to pay for the booking and returns an 'eTicket', or (2) to save the booking and pay later on. In the later case, the client is provided with a confirmation number that makes reference to the seat reservation.

### Use of BPEL for describing composite services

As aforementioned, the train ticket booking functionality can be composed with a hotel booking one in order to provide a travel booking functionality. An overview of the travel booking orchestration is illustrated in Figure 2.14.

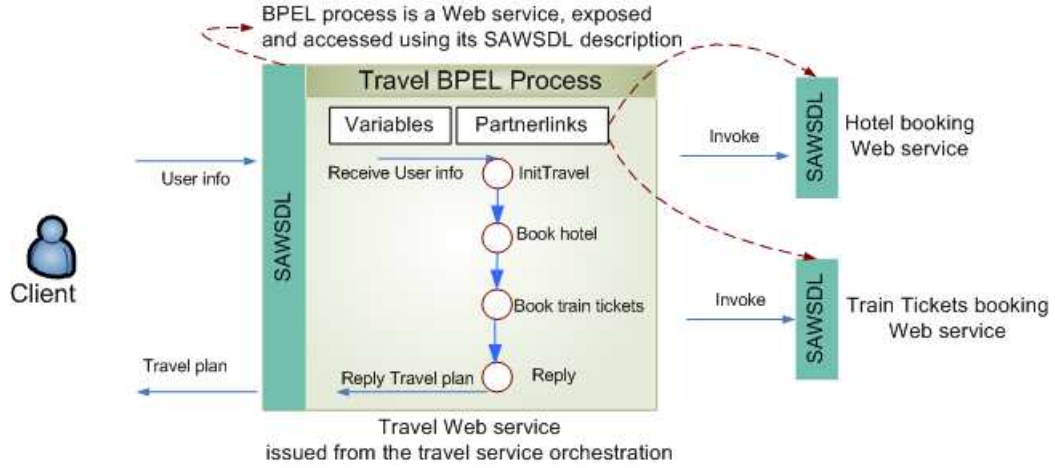



Figure 2.14: BPEL interactions with Web services

As mentioned in the beginning of this section, BPEL process combines the operations of Web services in a workflow structure. The newly created process is exposed to the external environment as a Web service by advertising its SAWSDL description. A client that would like to organize its travel would invoke the process using the operation described in the SAWSDL

description of the process. S/he invokes the 'Travel' BPEL process using an 'InitTravel' operation, and provides the process with the required inputs. The 'Travel' process first checks the available hotel rooms located in the destination city ('Hotel booking' functionality in Figure 2.14). Then, the 'Travel' process books the train tickets according to the user inputs ('Train Ticket booking' functionality). Finally, the BPEL process returns the travel plan to the client as a response to the 'InitTravel' request. The bindings with the 'Hotel booking' , the 'Train Ticket booking' and the 'Travel' service itself are declared using a list of *Partnerlinks* in the 'Travel' process, where each *partnerlink* is related to a specific Web service.

### 2.4.5 WS-Resource Framework

State management has always been an underlying issue for behavior execution, but only recently it has been brought to the forefront with the introduction of the Web Services Resource Framework (WSRF) [OASIS, 2006b]. WSRF is an OASIS standard, which defines a framework for modeling and accessing stateful resources through Web services. This includes mechanisms that describe views of the resource state and support its manipulation.

#### Representing service state

At their beginnings, Web services have been assumed to process incoming messages regardless of earlier invocations, or the time of message arrival. As mentioned earlier, this is commonly called stateless service behavior. However, as they gain in popularity, Web services provide more advanced functionalities, requiring to maintain a state across multiple interactions with their clients, implementing in most cases stateful behaviors.

In such cases, Web services and clients form a joint session during interaction, where the services responses depend not only on the input parameters, but also on the current session state. To differentiate their clients and maintain data related to the client during interaction, existing real world services store a session state at the client side, such as RESTful services [Fielding, 2000]. One example is services with authentication, which requires handling some proprietary user session such as with payment services. Another example is cart management in shopping services.

In general, to deal with this, proprietary implementations of session identification are included, e.g., in the form of the SOAP-body data [W3C, 2007b], or transport-protocol dependent methods such as HTTP cookies [Netscape, 1999, IETF, 2000].

Technically, cookies are arbitrary pieces of data chosen by the Web service and sent to the client. The client returns them unchanged to the server, introducing a state (memory of previous events) into otherwise stateless HTTP interactions. Without cookies, each interaction with the service is an isolated event, mostly unrelated to all other interactions of the client with the same service. By returning a cookie to a Web service, the client provides the service a means of relating the current interactions to prior ones. Other than being set by a Web service, cookies can also be set by a script in a language such as JavaScript, if supported and enabled by the client application. The client is then in charge of performing session-related operations, such as saving and sending back cookies to the service, which are specific for each particular kind of service.

While this solution might be feasible for pre-defined client applications, such as Web browser, dynamic service consumption and composition are not able to perform specialized state operations for each service. Instead, dynamic service consumption and composition facilities demand a unified and interoperable way of interacting with all services, regardless of their technical varieties or stateful behaviors.

Based on the above requirements, the idea behind WSRF is to provide a dynamic service hosting environment with a unified service interface. Instead of putting the state *in* the Web service, WSRF keeps it in a separate entity called a *resource* or stateful resource, which stores all the state information. The description of the resource is integrated in the unified service interface, providing thereby external entities (i.e., clients and other services) with a view of the state that the service manipulates in order to access and manipulate it in a standardized way.

In particular, the WS-Resource specification [OASIS, 2006a] defines a stateful service as a service that acts upon stateful resources. This definition assumes that a service/behavior acting upon stateful resources is described "statelessly", delegating the responsibility for the management of the state to another component such as a database or a file system.

The way that WSRF models services is interesting, but it does not cover all the existing cases of stateful behaviors. WSRF assumes that the service implementation is stateless, and all manipulated data are stored in an external resource, which is not common to all stateful behaviors. As mentioned in our service model, services implementing stateful behaviors may maintain an external resources state, besides the implementation state at runtime. Hence, in order to cover all cases of stateful behaviors, we deviate the use of WSRF from its initial definition of service state. Herein, we maintain our definition that service implementing stateful behaviors hold a state internally at runtime, and we use the WSRF standard to externalize a copy of the data maintained at runtime, in order to provide the external entities with the ability to access and manipulate a part of the service state.

### Overview of WSRF

A WS-Resource [OASIS, 2006a] is an association of a stateful resource with a Web service through which the resource can be accessed.

To understand the concept of WS-Resources, we use the example of *Service A*, enabling train ticket booking. Such a Web service requires manipulating a database (denoted SeatDataBase) that manages the seat bookings per user and per train. As depicted in Figure 2.15, the association between *Service A* and the 'SeatDataBase' is a WS-Resource, where the service is described using its SAWSDL and the resource is defined using a resource properties document.

A *WS-Resource Properties* [OASIS, 2006c] (or resource properties) document is an XML schema, which defines how the data associated with a stateful resource can be queried and changed using Web service technologies. The WS-Resource Properties document also serves to define the structure of the resource. For instance in the 'SeatDataBase' (Figure 2.15, the element 'SeatResource' has two attributes, namely 'SeatNumber' and 'SeatStatus' (which can be booked, paid or empty). 'SeatStatus' element enables to store the status of a specific seat in a train in order to prevent from, e.g., over-booking the train seats.

The association between the Web service and the resource is expressed by annotating the SAWSDL port type [12] (portType) with the type definition [13] of the resource properties document, using the attribute $< wsrp : ResourceProperties... >$.

Furthermore, based on the WSRF standard, a stateful resource supports a set of operations enabling external entities (e.g, Web services or clients) to access and manipulate the state of its elements. As presented in Figure 2.15, these operations are integrated in the SAWSDL description of the service in order to be advertised to, and used by, the external entities. According to the WSRF standard, the service supports the *GetResourceProperty* operation and may support a set of *SetResourceProperty* operations. The *GetResourceProperty* operation provides the

[12] The 'portType' element includes a supported set of operations in the (SA)WSDL description.

[13] The target name space of the XML schema that defines the resource.

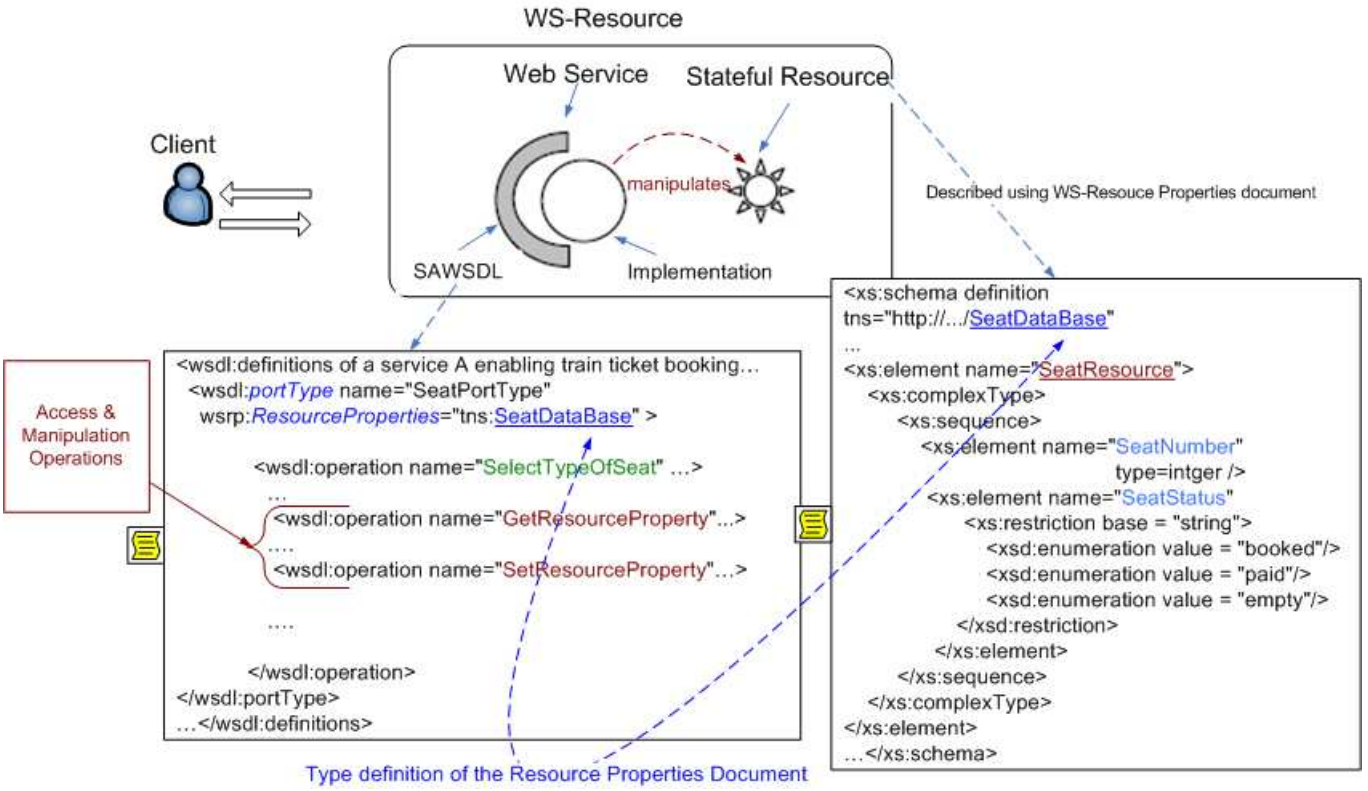



Figure 2.15: WS-Resource description

current state (i.e., XML data) of the XML element of the stateful resource at the request time. *SetResourceProperty* operations allow performing changes on the state of the resource. There are three types of change that may be applied on the state of the resource elements: *insert*, *update* and *delete*.

### Extending the WS-Resource Framework with Semantic Annotations

We extend the WS-Resource Framework with two semantic annotations: the first for the WS-Resource Properties document and the second for the SAWSDL description.

On the one hand, in order to understand the semantic meaning of the data stored in the resource elements, we extend the elements description of the resource properties document with semantic annotations. The semantic annotations are enabled by the standard SAWSDL, as it supports the annotations of both, WSDL and XML schemas.

For example, consider the train ticket booking ontology described in Section 2.4.1. Using the train ticket ontology in the resource properties document of the 'SeatDataBase', the element 'SeatResource' is annotated with its corresponding semantic concept (i.e., $Seat\#Reservation$), as follows:

On the other hand, in order to differentiate WSRF-enabled operations for state access and manipulation from the other service operations, we introduce an ontology that captures the impact of the operations on the service state, which we call 'OperationImpact' ontology.

The 'OperationImpact' ontology (illustrated in Figure 2.16) is retrieved from the different operations enabled by WSRF standard. An operation that queries the resource, such as GetResourceProperty, then is annotated with the 'QueryState' concept from the 'OperationImpact'

```
<xsd:schema tns:"http//.../SeatDataBase ...>
...
<element name="SeatResource">
 sawsdl:modelReference="http://URI_Train_Ontology/Seat#Reservation">
  <complexType>
       <element name="SeatNumber" type="intger"
        sawsdl:modelReference="http://URI_Train_Ontology/Reservation#SeatNumber"/>
       <element name="SeatStatus" type="string"
        sawsdl:modelReference="http://URI_Train_Ontology/Reservation#SeatStatus"/>
  </complexType>
</element>
...
</xsd:schema>
```

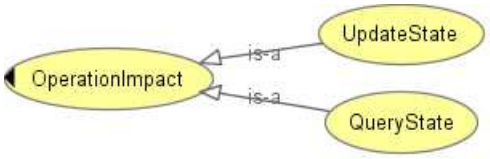



Figure 2.16: OWL ontology for distinguishing the SAWSDL operations with respect to their impact on the resource state

ontology. An operation that updates the resource, such as SetResourceProperty operations (delete, update, insert), is annotated with the 'UpdateState' concept from 'OperationImpact' ontology. The operations annotations are enabled by the SAWSDL standard, using the attribute *modelReference*. In this way, the SAWSDL of *Service A* (partly presented in Figure 2.15) includes, the WSRF-enabled operations and their semantic annotations. For example, GetResourceProperty operation is described in the listing below.

```
<wsdl:portType name="SeatPortType" ... >
...
  <wsdl:operation name="GetResourceProperty">
   sawsdl:modelReference="http://URI_of_the_Ontology/OperationImpact#QueryState">
     <wsdl:input name="GetResourcePropertyRequest" message="GetResourcePropertyRequest">
     <wsdl:output name="GetResourcePropertyResponse" message="GetResourcePropertyResponse">
  </wsdl:operation>
...
</wsdl:portType>
```

Using the semantic annotations, we are able to distinguish the "state access and manipulation" operations (i.e., Get/SetResourceProperty ) in the service interface, from the other operations that the service provides, which we call "functional" operations. We model this distinction in Figure 2.17.

### 2.4.6 SWRL

To express the pre- and post-conditions in WS, there is currently a main candidate: the Semantic Web Rules Language (SWRL) [W3C, 2004c], which is a proposal for standardization at W3C. Besides SWRL, many logic languages allow pre- and post-conditions to be expressed, including

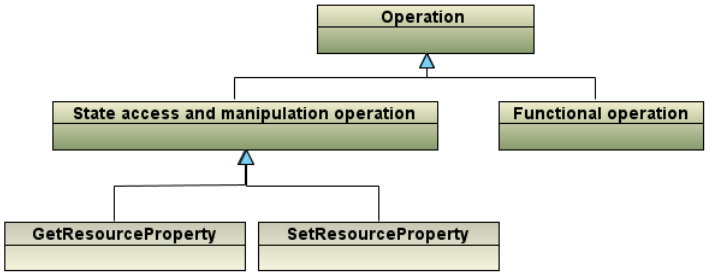



Figure 2.17: Distinguishing between state access and manipulation operations and functional ones

e.g., DRS [14], RDQL, SPARQL, and KIF. In SWRL and DRS the input parameters and local parameters involved in the formulæ can be mentioned by their URI, but this is not the case for SPARQL, RDQL, and KIF: when one of those languages is used, a mapping is needed to explicitly show the correspondence between variables involved in the formulæ and the corresponding variables in the behavior description.

SWRL combines the OWL Web Ontology Language [W3C, 2004b] with the Rule Markup Language (RuleML) [Boley et al., 2001]. SWRL allows users to write Horn-like rules [15] expressed in terms of OWL concepts to reason about OWL individuals, primarily in terms of OWL classes and properties.

In common with many other rule languages, SWRL rules are written as antecedent-consequent pairs. The intended meaning can be read as: whenever the conditions specified in the antecedent hold, then the conditions specified in the consequent must also hold. In SWRL terminology, the antecedent is referred to as the rule *body*, and the consequent is referred to as the *head*. The general form for these rules is expressed as follows [W3C, 2004c]:

$$body \Rightarrow head$$

where *body* and *head* are a set conjunctions of atoms, which are unary or binary predicates, i.e., properties that assigns truth values to combinations of $k$ individuals ($k \in \{1, 2\}$). A general form of the *body* and *head* of a rule can be expressed as follows.

$a(x, y) \wedge b(y, z) \wedge c(x) \wedge \ldots \Rightarrow n(x, z)$ where
$a,\ b,\ n$ : binary predicates(roles).
$c$ : unary predicate (atomic concept).
$x,\ y,\ z$ : variables, instances or literals.

At the current stage of specification, SWRL does not support more complex logical combinations of atoms than implications and conjunctions.

To illustrate the use of SWRL for describing service rules, we employ the train ticket booking scenario. For instance, a SWRL rule expressing that the seat number should not change after confirming the reservation requires capturing the concept of 'SeatNumber' in OWL. Intuitively, the relationships for confirming and changing the value of a seat number can be expressed using OWL properties 'hasValue', 'hasConfirmed' and 'hasChanged', which are attached to 'SeatNumber'. The rule in SWRL would then be:

[14] DRS is described by Drew McDermott in an appendix of the OWL-S 1.0 release [W3C, 2004a].

[15] a Horn clause is a clause (a disjunction of literals) with at most one positive literal, e.g., $\neg p_1 \vee \neg p_2 \vee ... \vee p_n$. A Horn formula is a conjunction of Horn clauses.

$$SeatNumber(?SN) \wedge hasValue(?SN, ?CltId) \wedge hasConfirmed(?SN, ?CltId) \Rightarrow \neg hasChanged(?SN, ?CltId)$$

where $SN$ and $CltId$ are the variables, such that $SN$ represents the the seat number, and $CltId$ represents the identifier of the client requesting for train ticket booking, and for whom the seat number $SN$ has been assigned. A question mark (?) is put before the variable name to denote that the rule is applied on a specific value of the variable.

Applying this rule would verify that if a value of the variable $SN$ is correlated to a client provided with a specific client identifier $CltId$, and the client has confirmed his reservation, then the correlation should be kept till the end of the behavior execution.

## 2.5 Integrating Web Service Concepts in our Basic Service Model

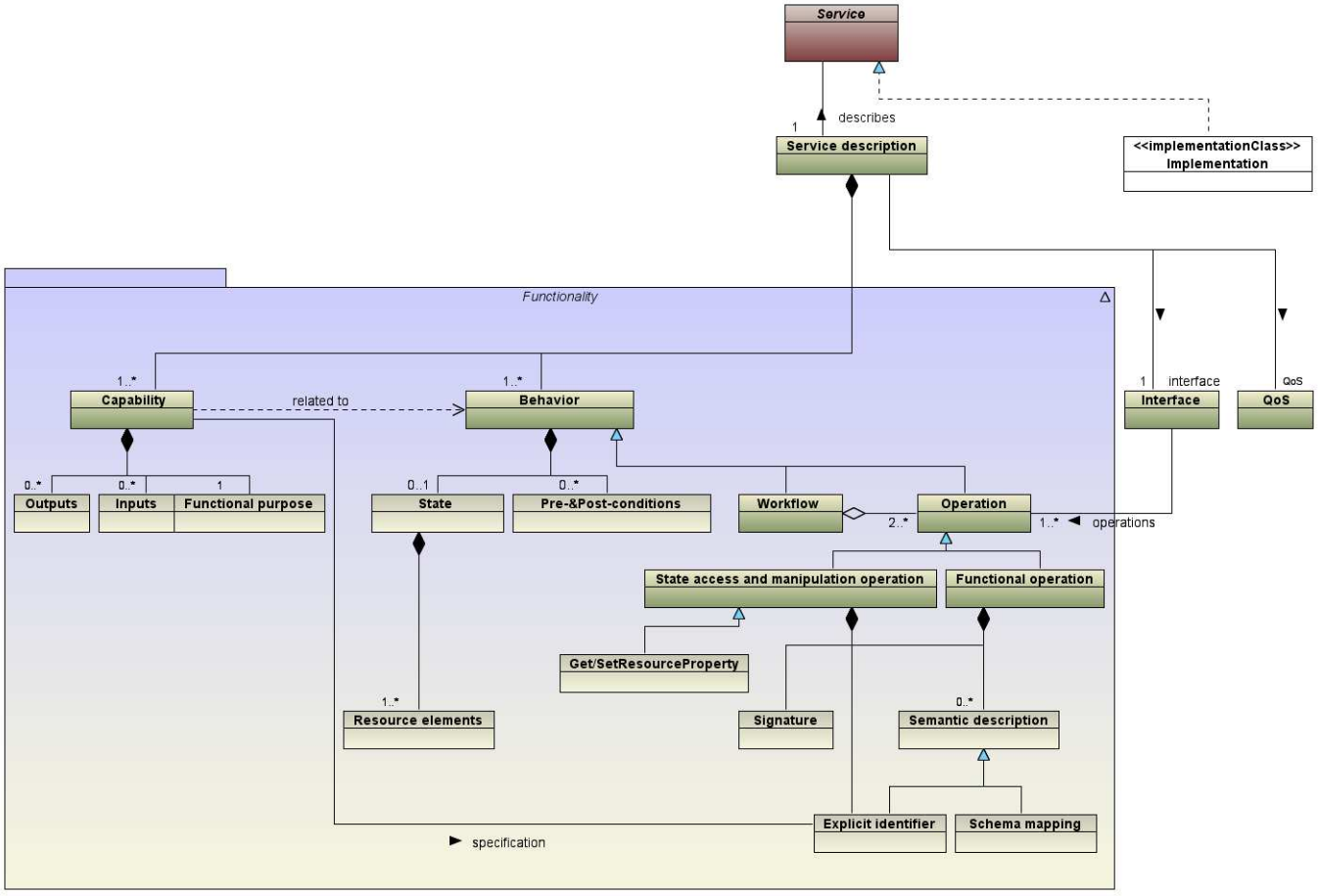



Figure 2.18: Enriching the basic service model with Web services concepts

In the previous section, we presented the most commonly used Web service standards that enable the description of the concepts of our basic service model. However, the presented Web service standards introduce a set of concepts that have to be related to the concepts of our service model. We further integrate them in our service model (presented in Figure 2.7), which becomes

a specialization of SOA paradigm using Web service technologies. The result of enriching the service model is presented in Figure 2.18.

In SAWSDL, we presented the "semantic annotation" concept, which corresponds to the concept "semantic description" of the operations in the service model introduced in Section 2.3. Integrating the semantic annotations model (introduced in Figure 2.9), the semantic description of the operations in the service model (Figure 2.18) becomes of two types: "explicit identifier" and "schema mapping". As the "explicit identifiers" represent the semantic concepts that describe the operations and their in/out parameters, they represent a *specification* of the concept "capability description".

In WSRF, we distinguished two kinds of operations: "functional" operations and "state access and manipulation" ones. Hence, in our service model, we enrich the concept "operation description" with two sub-types, retrieved from the operation model, introduced in Figure 2.17. *Get* and *SetResourceProperty* operations are modeled as a sub-type of the "state access and manipulation" operations. Their semantic description includes the explicit identifiers that take their value from the 'OperationImpact' ontology. Other (functional) operations can be described with a richer semantic description that includes schema mappings.

WSRF models a state as a set of "resource elements". Integrating this concept in our service model, we model a state description as a composition of (1..*) "resource elements".

## 2.6 Concluding Remarks

Service-oriented architectures (SOA) have proliferated in the few last years, due the facilities they provide to both users and software developers. They enable to cope with the computer environments' heterogeneity. In this chapter, we presented the basic concepts of SOA, focusing more specifically on the service model. We established a basic service model that integrates the main concepts that define a service as well as their inter-dependencies.

We further emphasized the importance of the SOA paradigm through one of its major realization: Web services. Web services do not only support the concepts introduced in our service model, but they also provide means to describe them through a set of Web service languages. Among these languages, we used OWL ontologies for describing the service capabilities, SAWSDL for describing the service interfaces, BPEL for describing composite behaviors and services, WSRF for describing the service state, and SWRL for describing pre- and post-conditions. In addition, we extended WSRF with semantic annotations in order to provide the description of the state and the operations that manipulate it with a richer semantic description. Finally, we integrated the WS-introduced concepts in our service model which becomes a specialization of SOA concepts using WS technologies.

Still, the SOA abstraction contributes, but is not sufficient, to cope with all pervasive environments' characteristics, such as openness and dynamics. In particular, SOA systems do not support service continuity in face of runtime variations of services availability. In the following chapter, we present the necessary background on dependability in SOA systems.

*There is a tremendous difference between a computing system that works and one that works well*

[Birman et al., 2004]

# 3

# Dependability in SOA Systems

Since SOA systems are collections of interacting services implemented on multiple interconnected machines, they inherit all the classical challenges associated with building distributed systems, including dependability requirements in order to ensure continuity in service provisioning for the users.

Ensuring service continuity in traditional closed distributed systems has been the focus of several research activities; accurate and relevant solutions exist, each of them enhancing a specific facet of dependability, including system reconfiguration to deal with failure during system execution. However, these solutions are not applicable as they are in SOA systems. Indeed, in traditional distributed systems, components are implemented to work together, where a main authority has knowledge of the changes that should take place, and its main responsibility is to perform them, whilst not jeopardizing the overall system integrity [Kramer and Magee, 1990]. In SOA systems, such authority lacks. Services are autonomous entities that are not implemented to work together. Hence, their data dependencies cannot be predicted at design time. Ensuring dependability in SOA systems should take into account these dynamic data dependencies. Also, the knowledge about services is limited to their description. The service description has thus to advertise the dependability means the service supports, in order to use them when a reconfiguration is required. Thus, ensuring dependability in SOA systems requires to (1) identify the individual dependability means of the SOA system components, i.e., services, and (2) coordinate them according to their data dependencies, while (3) respecting services autonomy and loose coupling.

This chapter points out the need for adapting traditional techniques in order to ensure dependability in SOA systems. Section 3.1 presents the fundamental concepts of dependability, focusing, in particular, on service unavailability as a type of failure that threatens dependability of SOA systems. More specifically, we envision to replace an unavailable service with another one, in order to take over the execution of the unavailable service. To this aim, Section 3.2 presents the existing dependability techniques that are used to face component unavailability in closed distributed systems, as opposed to open dynamic SOA systems. In Section 3.3, we define the concept of service substitution, and present the limits of applicability of these techniques in SOA systems. In particular, we stress the need for adapting the dependability techniques of closed systems to fit SOA systems specifics, as well as, the need for a middleware support for dependability in SOA systems. Some of these techniques have been adapted and widely used in SOA systems. Still, their adaptation does not completely serve our need for runtime service substitution; Section 3.4 reviews a set of existing approaches for service substitution in SOA systems in order to propose an approach that deals with their limits while taking advantage of their strong points. Such an approach is faced with several issues; Section 3.5 points out these

issues. Finally, Section 3.6 presents our concluding remarks.

## 3.1 Basic Concepts of Dependability

Dependability of a computing system is the ability to deliver a service that can justifiably be trusted [Laprie et al., 1992, Avizienis et al., 2001]. The main goal of dependability is to conceive and specify systems in which a fault is natural, predictable and tolerable. It is a global concept that includes various notions that can be grouped into three classes: *threats*, *means* and *attributes*. [Laprie et al., 1992] organizes these classes along with their subclasses as a 'Dependability Tree', which is represented graphically in Figure 3.1.

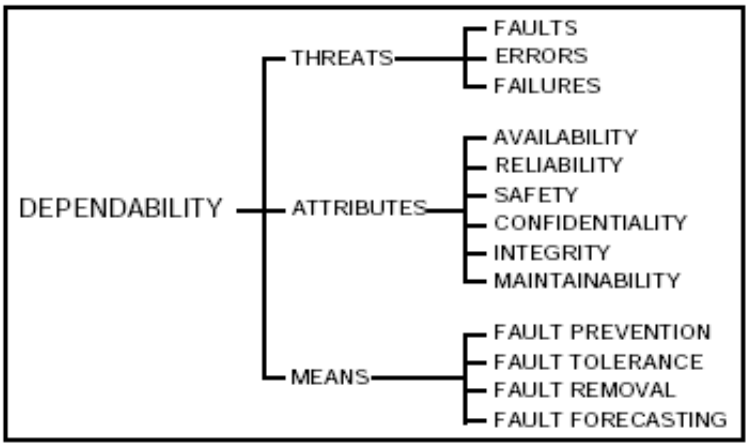



Figure 3.1: Dependability tree

The threats against dependability are:

- *Errors*: are the parts of the system state that may cause a subsequent failure;
- *Failures*: occur when errors reach the service interface and alter the service;
- *Faults*: are the hypothesized causes of an error. A fault is *active* when it produces an error; otherwise it is *dormant*.

They are undesired –in several cases, unexpected– circumstances causing or resulting in undependability. The means to attain dependability are:

- *Fault prevention*: how to prevent the occurrence or introduction of faults;
- *Fault tolerance*: how to deliver correct service in the presence of faults;
- *Fault removal*: how to reduce the number or severity of faults;
- *Fault forecasting (or prediction)*: how to estimate the present number, the future incidence, and the likely consequences of faults.

which are techniques enabling to deliver a service on which reliance can be placed, and to reach confidence in this ability. The attributes of dependability [Laprie et al., 1992] are:

- *Availability*: readiness for correct service;

- *Reliability*: continuity of correct service;
- *Safety*: absence of catastrophic consequences on the users and their environment;
- *Confidentiality*: absence of unauthorized disclosure of information;
- *Integrity*: absence of improper system state alterations;
- *Maintainability*: ability to undergo repairs and modifications.

#### Service failure

In the system dependability domain, the lifetime of a system is perceived by its users as an alternation between two states of the delivered service namely correct and incorrect, which are relative to the accomplishment of the system function:

- *Correct service*, where the delivered service accomplishes the system function;
- *Incorrect service*, where the delivered service does not accomplish the system function.

A service failure is an event that occurs when the delivered service deviates from the correct service. A service failure is thus a transition from a state of correct service to incorrect service. In contrast, the transition from incorrect to correct service is a *restoration* [Laprie et al., 1992].

Due to computing environments dynamics, service *availability* cannot be guaranteed. In this thesis, we aim at dealing with service *unavailability* by ensuring continuity in service provisioning for the user, i.e., *reliability* of SOA systems. In our study, we concentrate on service unavailability as a *type of failure* that threatens reliability. We study more in details the causes that induce service unavailability, and we focus on dependability means that enable the SOA system restoration.

#### Causes of failure

In distributed systems, service availability is based on the presence of a network connection between the client and the service. [CHANDRA et al., 2001] proposes a failure classification based on location, in which we can distinguish three types of failures: 1) "*near-user*", 2) "*in-middle*" and 3) "*near-host*". "Near-user" failures represent failures that disconnect a user device from the rest of the networked environment. Similarly, "near-host" failures make the service provider unreachable from the other networked environment constituents due to, e.g., crashes, overloads. "In-middle" failures refer to the break of the network connection that links the user device and the specific service providers, but the user may still connect to a significant number of the remaining networked services. Most notably, these failures represent an interruption of connectivity to a single device that does not affect any other device to communicate. In our study, we focus specifically on dealing with "in-middle" and "near-host" failures to ensure the continuity of service provisioning for the user. Indeed, "near-user" failures make the user isolated from any other networked device, which makes the realization of service continuity possible only with local services that are deployed on the user device.

#### Dealing with service failure

To deal with service unavailability, the dependability means of Figure 3.1 may be grouped into the two following type of solutions:

- Prevent or forecast service unavailability either by eliminating, or by predicting all cases that induce "in-middle" or "near-host" failures. However, this is not achievable in dynamic environments where independent entities (i.e., clients and services) autonomously join and leave the networked environment without beforehand notification.

- Tolerate service unavailability and remove the failure by reconfiguring the SOA system according to the user environment.

Both fault prevention and forecasting means have the same purpose of fault avoidance. However, fault avoidance can be applicable only at the price of user freedom (i.e., mobility) or environment dynamics. In our study, we mainly concentrate on the fault tolerance (FT) techniques in order to respect the environment dynamics and entities autonomy when restoring SOA systems function [Fredj et al., 2006].

As presented in the previous chapter, SOA systems implement either stateless or stateful behaviors. The restoration of a SOA system is more or less straightforward of stateless behaviors. Indeed, from a client-oriented perspective, the unavailability of a stateless behavior before producing the required results amounts to re-sending the same request to another service instance that provides the required functionality. In addition, since a service that implements a stateless behavior does not maintain any state for the interaction with a specific client, its recovery from a network disconnection or other failure consists in interrupting the execution processing of the received request and getting back to a failure-free situation, i.e., before receiving the client request. Therefore, stateless behaviors are by definition fault tolerant and actually do not require any extra effort other than redirecting the client request to another networked service, and canceling the request processing in order to leave the service in a consistent state. The restoration of SOA systems that involve stateful behaviors may require a significant effort for maintaining and transferring the state of the service that becomes unavailable. *State durability* is an optional attribute of dependability that focuses on maintaining the state availability after service failure. In our study, a part of our focus includes the study of the techniques that preserve state durability in SOA systems.

However, before going more in details in the study of the FT techniques for *open* and *dynamic* SOA systems, we first study existing techniques that deal with system unavailability in *closed* distributed systems in order to discuss the limits of their applicability in SOA systems, and elaborate on their adaptation in order to fit SOA systems specifics.

## 3.2 Tolerating System Unavailability in Closed Distributed Systems

Tolerating systems unavailability have been the focus of tremendous efforts in closed distributed systems, where all the system components that –actually, or possibly– participate in the distributed system are known, and in most of the cases designed and implemented by the same team. The specifics of closed distributed systems are different from the ones of open, dynamic SOA systems, however, the issue remains the same: How to enable continuity of the system execution when a system component becomes unavailable at runtime.

The abundant resources of similar software systems that are available in the networked environment makes fault tolerance by design diversity [Bishop, 1995] a natural choice for distributed systems to ensure reliability. Replacing the unavailable software component with another one available in the networked environment enables to fulfill the distributed system objectives.

Runtime system replacement recalls naturally the well-known replication technique. Section 3.2.1 presents the basic concepts and the different strategies of replication that have been

introduced and used in closed distributed systems. Then, we present in Section 3.2.2 the impact of the replacement of a system component on the other system components that are participating in the distributed system. We also present the techniques that enable restoring the distributed system consistency after component replacement.

### 3.2.1 Basic Concepts of Replication

From the review of the state of the art, replication is the *de facto* solution for systems reliability. Replication consists in distributing replicas of data or a software application over a network, in order to back up the data or the functioning of the system in the case of failure. In the case of software application replication, the functioning of the original application with its replicas are regulated with strategies that indicate, e.g., the right number of replicas, their appropriate locations, and the way they interact with the original software application, and also among them. However, strategies for replicating distributed systems face a trade-off: they should be (1) efficient (low latency), while (2) ensuring consistency of the replicas. The replication strategies can be of three types: active, passive or replay [Helal et al., 1996].

- *Active replication*: replicas run simultaneously with a constraint of *total* order in sending and receiving messages, i.e., clients send requests to all replicas, and replicas have to process the clients' requests in the same order. In this way, all replicas maintain the same data as a state, which are updated in the same way according to the clients requests. The state coherence among the set of replicas is made implicitly: replicas have to provide deterministic code in order to reach the same state after processing each request, i.e., given the same message as input, all the replicas produce the same output message as a result of processing the input message. In this way, at any time of execution, all replicas hold the same data internally, which represent their state.

  To overcome the constraint in messages order, active replication is usually implemented using the *Atomic Broadcast* algorithm [Birman et al., 1991, Kaashoek and Tanenbaum, 1991]. The algorithm is set as follows. Consider a group $G$ of replicas, and a client issuing an atomic broadcast of a message $m$ to $G$. First, the message $m$ is sent to all the replicas in $G$. Then, one of the replicas in $G$, called the sequencer, assigns sequence numbers to the request messages and sends these numbers to $G$, to inform them of the order of executing the clients' requests: when a replica receives a client's request, the replica does not execute the request until it has received the message order from the sequencer. Then, each replica in $G$ delivers the response messages according to the sequence numbers of the related requests. Several efforts have been proposed in the literature in order to make atomic broadcast efficient (i.e, with low latency), while ensuring consistency of the replicas. A comprehensive survey of a set of these efforts is presented in [Défago et al., 2004].

  The main advantage of active replication resides in its simplicity and failure transparency. However, the determinism constraint [1] is the major limit of this strategy. To support non-determinism in active replication, existing approaches rely on *synchronization* or *semi active replication* [Poledna, 1996].

  *Synchronization* consists in applying an agreement protocol (i.e., consensus) that rules out dynamic scheduling decisions and task preemption. The concensus problem is stated as follows [Wolf, 1998]: Given a set of replicas, each replica proposes a value $v_i$. They then have to decide on a one common value $V$ of the values $v_i$.

[1] Starting from the same state, all the replicas have to reach to a same different state after processing the same requests in the same order.

*Semi active replication* [Powell et al., 1991] relies on the selection of a "leader" (a central replica) that is in charge of executing non-deterministic operations. The other replicas (called "followers") have to be informed of the leader's decision. In this case, it is not necessary to keep message order consistent for the replicas group. Rather the leader selects the message to process next, and communicates his selection to the followers. Deterministic operation sequence can be executed by the leader and the followers concurrently. Compared to synchronization method, the advantage of semi active method is that non-deterministic operations are allowed without the need to carry out a consensus prototcol which induces low complexity of the communication protocol. An information dissemination protocol, such as reliable broadcast, is sufficient for semi-active replication to handle non-determinism. Still, the communication overhead for non-deterministic decisions may be unacceptable for many application areas.

- *Passive (primary-backup) replication* [Budhiraja et al., 1993] where a replica is designated as a *primary* and all the others as *backups*. Compared to active replication, for passive replication, only one replica (i.e., the primary) among the group is active . Clients make requests only to the primary. If the primary fails, then one of the backups takes over the interaction with the clients. Two main variants of the passive replication are *cold* and *warm*, where the difference lies mainly in the synchronization between the primary and the backups. In *cold passive replication*, each request processed by the primary is checkpointed (i.e., the state of the replicas is stored after processing the request) into a predefined secondary storage. In case the primary fails, the logged information in the secondary storage is retrieved and used to start a new primary. Hence, the backup replica synchronizes its state only when a failure occurs. The *warm passive replication* requires that all state updates generated by the primary are propagated periodically (i.e., after a pre-fixed period of time) to the backups. As a special case of the warm approach, *hot passive replication* propagates the state updates at the end of each request that updates the state of the replica.

  The difference between active and passive replication lies in the determinism constraint. In passive replication, the requirement is related to the synchronization of the replicas, whatever the state of the primary replica is, the secondary has to synchronize its state accordingly. There is no constraint on the processing of the requests. All the secondary have only to be able to synchronize according to the state of the primary, and resume the execution from the last state stored.

  In comparison with active replication, a main advantage of passive replication is the low consumption in terms of computing and networking resources, as the requests are not processed by the replicas, and the client interacts with a replica using unicasts. However, passive replication suffers from a high recovery time when failures arise, specially, in cold passive replication.

- Another type of replication is *replay* [Strom and Yemini, 1985] or log-based replication, which consists in reconstructing the state of the primary on a new replica. The reconstruction of the state is based on logging all the message exchanged between the client and the primary. The execution is restarted from the beginning with the backup replica, in order to reconstruct the state of the primary.

  As for active replication, the replicas have to be deterministic in order to provide the same processing results as the primary. We consider the replay as a middle strategy between active and passive replications: as for passive replication, it relies on stored data (i.e.,the

logged messages) to put the secondary replica in the same state as the primary, at the difference that the stored data represent the logged messages and not the state of the primary. On the other hand, as for the active replication, the state of the replica is reconstructed as a result of processing the set of requests, and not as a result of synchronization with the state of the primary. In this way, replay reduces network consumption by avoiding requests broadcast. It involves the secondary replica only when a failure occurs as for passive replication. However, unlike active replication, replay saves computational resources since only a single replica processes the requests, and not all the replicas, and only when a failure occurs. However, upon a failure that affects a long-lived conversation, the response time of the replay turns to be high, since the backup service replays the sequence of all messages that have been exchanged from the beginning of the conversation.

### 3.2.2 Reconfiguring Closed Distributed Systems

Using passive replication, the secondary replica synchronizes its state with the last state stored (or checkpointed) of the primary replica. However, the system unavailability may occur between two state storages. In such cases, some computation performed by the now unavailable system is lost, as it is not taken into account in the state transferred to the secondary. In the case that the distributed system involves a set of component systems, besides the unavailable component, these components may be affected by the loss of computation due to data dependencies with the now unavailable system. In this case, it is required to reconfigure these systems in order to restore the distributed system consistency before resuming its execution, and notify the client about the reconfiguration.

This issue has been thoroughly studied in the fault tolerance domain, leading to checkpoint-based rollback recovery technique. Hereafter, we present a brief background on the basic concepts of such a technique, along with the existing protocols that realize it.

#### Basic concepts of checkpoint-based rollback recovery

Rollback recovery has been one of the most widely used means for system recovery in the case of failure. The basic idea behind it is to consider the system execution as a succession of valid system states, and when a failure occurs between two valid states, to roll the system back to a previously reached valid state and resume the execution from that state. Commonly used techniques for rollback recovery are based on *checkpoints*: the system saves in a stable storage some of the valid states it reaches during its execution. The saved state of a single process is called *local checkpoint*, the one of a system that includes multiple processes is called *global checkpoint*, which is a set of local checkpoints of the processes participating in the system. The action of saving the state is called *checkpointing* or *taking a checkpoint*. In rollback recovery, the dependencies between processes may force some of the processes that did not fail to roll back, creating thus a rollback propagation. For example, consider the situation where a sender of a message $m$ rolls back to a state that precedes the sending of $m$. The receiver of $m$ must also roll back to a state that precedes $m$'s receipt; otherwise, the states of the two processes would be inconsistent because they would show that message $m$ was received without being sent, which is impossible in any correct failure-free execution [Elnozahy et al., 2002]. Rollback propagation helps the system to resume its execution from a *consistent global checkpoint* [Chandy and Lamport, 1985], also called *recovery line* [Randell, 1975]. Under some scenarios, rollback propagation may extend back to the initial state of the computation, causing the loss of all the work performed before a failure. This situation is known as *domino effect*.

The checkpoint-based techniques for system recovery which restore the system state from the checkpoints previously taken, are qualified as *checkpoint-based rollback recovery*. Different protocols of checkpoint-based rollback recovery are presented in the following.

### Protocols for checkpoint-based rollback recovery

In checkpoint-based rollback recovery [Elnozahy et al., 2002], there are three types of checkpointing protocols that distinguish themselves according to the tradeoffs achieved between the overhead of checkpointing and the effectiveness in reducing the extent of the domino effect: coordinated, communication-induced and uncoordinated.

*Coordinated* or *synchronized checkpointing* protocols require processes to synchronize their checkpoints in order to form a consistent global state of the system at each checkpointing decision. During the global synchronization phases, synchronization messages are exchanged between processes. There exist both blocking and non-blocking checkpointing methods. In the blocking approach, processes block their normal execution during checkpointing and exchange only synchronization messages. While in the non-blocking one, processes overlap their normal execution with the synchronization phase. Generally, the non-blocking approach is preferred due to its lower latency. Coordinated checkpointing ensures recovery without domino effect, since, after a failure, each process will restart from its most recent checkpoint.

Alternatively, *Communication-Induced Checkpointing* (CIC) [Alvisi et al., 1999] protocols enable processes autonomy in deciding when to take checkpoints while avoiding domino effect. Processes take two kinds of checkpoints, *local* and *forced*. A process can take a *local* checkpoint at runtime, when the checkpointing is most convenient [Li and Fuchs, 1990]. In addition to local checkpoints, CIC protocols forces processes to take checkpoints, i.e., *forced* checkpoints, in order to guarantee the eventual progress of the recovery line. As opposed to coordinated checkpointing, CIC protocols do not exchange any special coordination messages to determine when forced checkpoints should be taken. To guard against the domino effect, the CIC protocols piggyback protocol-specific information to application messages that the processes exchange. The receiver then examines the information and occasionally is forced to take a checkpoint. Thus, processes have to pay the overhead of piggybacking information on top of the application messages.

Similarly to CIC, *uncoordinated checkpointing* allows each process to decide autonomously when to take checkpoints. The distinctive feature of this kind of protocol compared to the previous ones is that it neither synchronizes nor enforces checkpoints, maximizing thereby processes autonomy. The main advantage is that each process may take a checkpoint whenever it is most convenient and locally appropriate. However, uncoordinated checkpointing may suffer from the domino effect. In this context, many approaches have been proposed to remedy to the domino effect in independent checkpointing. Among them, *log-based uncoordinated rollback recovery* uses both checkpointing and message logging techniques, such as in [Sens, 1995]: instead of relying solely on checkpoints for recovering from a failure, each process logs the messages that have been received from and sent to other processes between successive checkpoints. After a failure, the system uses checkpoints to recover to a recent error-free state and replays the logged messages to move its execution to a point as close as possible to the occurrence of the failure. The system determinism ensures the state consistency after recovery.

## 3.3 Discussing the Limits of Applicability of FT Techniques for Closed Distributed Systems in SOA Systems

As described in the previous chapter, SOA systems present an evolution of the traditional closed distributed systems. In closed distributed systems, fault tolerance is designed and implemented in accordance with the system components. However in SOA, application components (i.e., services) are independent entities that are not necessarily designed to work together. Thus, even though each service may implement its own FT mechanism, services are not aware of possible data dependencies with other services involved in the SOA system. In case of service unavailability, a mechanism that ensures the whole SOA system consistency is required [Zarras et al., 2006]. Consequently, an automated integration of FT mechanisms to SOA systems is highly desirable, as it allows making a system –composed of independent services– fault tolerant with regard to the individual FT mechanisms of the services. Furthermore, the FT mechanism should be reusable for different SOA systems, as SOA systems can be composed dynamically and their composition is mutable over time.

In this section, we point out the limits of applicability of the above presented FT techniques in SOA systems. We first define the notion of service replacement or substitution in SOA systems. Then, we consider the applicability of replication techniques on services in order to substitute a service with another one. Finally, we consider the case that the service substitution is performed on a running orchestration in order to point out the limits of applicability of checkpoint-based rollback recovery protocols. We finally stress the need of a middleware support for handling the adaptation of replication and checkpoint-based rollback recovery in order to ensure SOA systems reliability.

### 3.3.1 Service Substitution

When considering the issue of service substitution, it is first necessary to clearly define what is the exact meaning of a *substitute service* in open, dynamic SOA systems.

**Definition 4. Service Substitution**
A service $A$ is a *substitute* for a service $B$ if the service $A$ offers a functionality $F_A$ that is able to replace at runtime a functionality $F_B$ provided by the service $B$, i.e., the two following requirements have to be met.

1. *Respect of the requested capability*: if $F_B$ complies with a user-requested capability, then $F_A$ should also.
2. *Functionalities conformance*: $F_A$ has to conform to $F_B$ in order to ensure the runtime substitution. Functionality conformance comprises both capability and behavioral conformance. The former ensures that the functionality $F_A$ preserves the meaning of $F_B$, which is guaranteed if the first requirement is satisfied. The latter guarantees that the runtime replacement of the $F_B$'s behavior by $F_A$'s one can be correctly performed and does not lead to incoherent results.

Note that the service substitution is defined essentially with respect to the functionalities provided by the services. Indeed, it is more accurate to call it "functionalities substitution", instead of "service substitution", as services may provided multiple independent functionalities. Also, in the above definition, we do not impose any constraint on services implementation. The unavailable service and its substitute can be independently designed and implemented, which gives to service substitution a flexibility that fits SOA systems autonomy and loose coupling.

### 3.3.2 Applicability of Traditional Replication Techniques

Active replication strategies provide short recovery time. However, they require determinism for the replicas, while in general services in SOA systems are not necessarily deterministic.

Passive replication reduces network utilization by activating redundant replicas only in case of failures. It requires replica synchronization after a failure, which requires that the secondary replica is able to *understand* and correctly *use* the state transferred from the primary. However, services are not assumed to be identically realized.

Similarly to passive replication, replay is activated only upon a failure, with the difference that it does not synchronize the state of the replica but reconstructs the state of the primary in the secondary replica. Thus, the replay may induce a higher recovery time than passive replication in the case that the failure occurs at an advanced stage of the execution. Also replay, requires determinism in order to enable the secondary to use the exchanged messages with the primary. However, SOA systems are not deterministic.

Hence, the above techniques need to be adapted to meet SOA systems specifics.

1. In active replication, we may broadcast each client request to all service candidates and tolerate that the state of the candidates may be different. At substitution time, the client has to be updated according to the current state of the substitute service. Still, this is applicable only when the service responses are not governed by the client interactions, but only by the state they maintain. This heavily restricts the service internal logic. Hence, in our study, we essentially focus on passive and replay techniques.

2. In passive replication, the synchronization requires a state mapping between the state provided by the now unavailable service and the one required by the substitute service.

3. The replay should adapt the sequence of messages as well as their content according to the substitute service requirements.

In Section 3.4, we survey and discuss how some existing replication strategies (particularly passive and replay ones) need to be adjusted because of the intrinsic features of services, and how substitution cannot fit into other strategies (particularly active ones).

### 3.3.3 Applicability of Traditional Checkpoint-based Rollback Recovery

Using passive replication, the substitute service should be able to synchronize with the last state stored of the unavailable service.

However, as the synchronization is based on a state mapping between the unavailable service state and the substitute one, in some cases, the mapping is not possible with last stored state of the unavailable service, but only with a previous one. In other cases, the last state stored of the unavailable service does not correspond to the most recent state stored before the unavailability occurs. In both cases, some computation performed by the now unavailable service is not taken into account by the state transferred to the substitute service, and thus is lost. In the case that the service participates in a composition, and more particularly in a service orchestration, not only the client but also the other (still-available) services participating in the orchestration may be affected by the loss of computation due to data dependencies with the now unavailable service. The service orchestration has then to be reconfigured in order to restore its consistency before its execution resumption, and notify the client of the reconfiguration. This recalls all naturally the issue tackled in checkpoint-based rollback recovery. Hereafter, we consider the applicability of the previously presented checkpoint-based rollback protocols.

- In coordinated checkpointing, the replacement of the unavailable is heavily constrained, i.e., the substitute component should have the same implementation as the old one and should be able to synchronize it state with any given checkpoint. While, we target open, dynamic SOA-based distributed systems without restricting the implementation of the service substitute. Thus, we cannot expect from the substitute service to be able to synchronize its state according to any checkpoint, which is provided as a black box from the unavailable service. Hence, coordinated checkpointing is not appropriate for services populating todays' open SOA environments.

- The CIC protocols are restrictive for SOA systems, as they expect from services to be able to interpret protocol-dependent information. Also, services have to allow forced checkpoints at any time of their execution regardless of the logic of their implementation. Similarly to coordinated checkpointing, CIC protocols expect that substitute services are able to synchronize with any data included in a checkpoint at any time of the execution of the behavior. These constraints make CIC protocols hardly applicable in autonomous SOA systems.

- Regarding uncoordinated checkpoint-based rollback protocol, the main advantage lies in allowing each component system to take a checkpoint whenever it is most convenient and locally appropriate, which respects the loose coupling and autonomy of SOA systems.

Among the discussed protocols, uncoordinated checkpointing best meets the specific characteristics and requirements of open, dynamic SOA systems. However, it is not directly applicable in SOA systems. First of all, the notion of checkpoint is not explicitly defined in SOA systems. Thus, we should extend the execution of SOA systems with checkpoints. Also, services involved in service orchestrations are not aware of each other, and thus, they are not aware of their data dependencies. Maintaining orchestrations consistency after service unavailability requires managing the service checkpoints by a party that is aware of the data dependencies between services, i.e., the client. Hence, we should make the client fault tolerant in order to manage the reconfiguration of the orchestrated services.

For reusability purposes, we stress in the next section the need for a middleware support to embed the approach that adapts FT techniques to specifics of SOA systems.

### 3.3.4 Need for Middleware Support for Fault Tolerant SOA systems

In the current study, we focus on middleware architectural support for fault tolerance, not because application-level or network-level approaches are uninteresting or less promising, but because the middleware-level seems to provide the required level of abstraction, genericity and reusability to deal with service unavailability [Bernard, 2006]. Indeed, applications solutions are very often not reusable for multiple SOA systems, and network ones require the support of network infrastructures to achieve seamless mobility. Hence, FT mechanisms should be supported by appropriate middleware. They need to have access to relevant information from the application layer (e.g., current state of service), but also from the lower layers (e.g., availability of network connections) in order to allow the middleware (possibly with some user intervention) to decide when to handle of, and how to change, the configuration of the system in order to maintain service continuity. To avoid dependence on network infrastructures, a promising solution should try to substitute the service that becomes unavailable with another service that can resume the execution previously started by the now unavailable service [Rong et al., 2007a]. The following section presents relevant efforts that have been proposed to deal with service substitution in SOA systems.

## 3.4 Existing Approaches to Support Service Substitution in SOA Systems

Aiming at enhancing dependability through replication [Helal et al., 1996], there exist a plethora of replication middleware for distributed systems. However, few solutions exist in the specific case of service-oriented systems [Osrael et al., 2006]. This section reviews the research approaches undertaken in the field of fault tolerant middleware for SOA systems. In particular, it focuses on some pragmatic issues in dealing with service substitution and service state restoration.

WS-DREAM [Zheng and Lyu, 2008] proposes an approach to evaluate the performance of different replication strategies and assist developers to select the most suitable strategy amongst nine recovery strategies, including active replication, passive replication, replay, and a combination of replay and active replication strategies. Other recovery strategies, such as retrying the same service instance, are not applicable in case of service unavailability. With WS-DREAM, users in different locations evolve in a collaborative environment, helping each other to carry out their requests, and share some results under the coordination of a centralized server. WS-DREAM server serves as a coordinator for the users. It is in charge of receiving requests, scheduling the users' tasks, and analyzing the results. Similar to service orchestrations, WS-DREAM has a coordinator server that generates, runs and coordinates a composite behavior (test plan) over Web services. The tests enable assessing the reliability of Web services in order to make service-oriented applications reliable. They also enable to select the optimal replication strategy for applications according to the reliability of individual services [Salatge and Fabre, 2007]. This complies with our purpose of adapting the replication strategy to the runtime system configuration.

Considering the issue of service unavailability from the service side, [Zhang et al., 2006] proposes a service-transparent approach to handle service replication and state reconstruction techniques. Zhang *et al.* propose an architecture that allows service developers to protect service state flexibly and transparently using durability attributes. The state is stored in one or more state objects, where the durability –that is, the likelihood that the state can survive failures– is an explicit attribute associated with each object. Services may persent different types of state, each of them with specific durability requirements. For example, an e-commerce service maintains inventory information, information about the regular customers (e.g., address, credit card number, preferences), and a state about on-going customer interactions ("shopping carts"). The inventory information is most valuable for the service because its loss would prevent the service from operating, and thus it is provided with a high durability attribute, while the loss of the information about ongoing customer interactions would be a nuisance for the users, but would not stop the service. Thus, it is provided with a lower durability attribute. The authors focus on optimizing the tradeoff between cost and the risk of losing the state. For example, some part of the service state can be stored in a database, while another part is replicated in-memory on two or more computers. The idea that governs this approach is to transform the service code in order to create a new Web service with the associated state object. Obviously, contrary to our main focus, this cannot be applied on services in general as the approach requires to have the service code in order to make services fault tolerant. However, what can be learned from this approach is that the state objects composing the state of a service can be provided with priority attributes, e.g., mandatory or optional, in order to be able to transfer the state of the service with missing (optional) data to the substitute service, while the substitution remains correct.

Similarly to [Zhang et al., 2006], the main issue considered by Tempest [Marian et al., 2008] is to store the soft state of a service (i.e., data that do not have to be stored durably and can be reconstructed at some cost, such as short-lived user sessions). It replicates the service's soft state

in order to make the recovery more efficient in time and resources rather than reconstructing it through user interaction or third-tier reaccess (i.e., database, file). Tempest provides the developer with a Java runtime library that enables to store in memory the soft state of a service as *TempestCollections* (i.e., data structures similar to Java collections) in order to decrease resource consumption when accessing the service state at recovery time. The data stored is automatically and transparently replicated across multiple machines, providing fail-over and load balancing of soft state. Based on active replication, Tempest multi-casts all the requests to the networked replicas (which are multiple instances of the same service that execute across different servers). To maintain replicas consistency, it relies on the possibility that all replicas converge to the same set of objects [DeCandia et al., 2007] using gossip-based reconciliation . The process of reconfiguration is set as follows. Each request is uniquely identified using a hash function over the front-end's IP address and port pair (called *wsiid*). This identifier serves essentially for selecting the replicas that first answer to a request, and for the reconciliation process, in order to identify the objects that are added and/or removed. For an update request, a request is sent to all replicas and a hashing mechanism is employed to select which instance is responsible for replaying the update on the state. For a read request, the front end entity selects the $k$ instances that respond first to the request; this ensures load balancing over replicas. *TempestCollections* are automatically and periodically checkpointed on a disk. A process fail is detected by time out. When a node crashes and reboots, upon starting the Tempest server, the services are brought up-to-date with the state that was last written to disk before the crash. When a server is newly spawned, or when a server that has been unavailable for a period of time missed many updates, Tempest employs a bulk transfer mechanism to bring the server up-to-date. In such cases, a source server is selected and the contents of the relevant *TempestCollections* are transmitted over a TCP connection.An implication of this model is that the programmer is not provided with ACID [2] transactions, which may lead to inconsistencies. To cope with the lack of atomic execution, the authors rely on the assumptions that (1) the soft state is not critical for the system functioning, and (2) data stored within soft state structures are naturally immutable (e.g., a list of items). Thus, manipulating it with arbitrary operations may not have impact on the soft state. For instance, a user is always asked to verify the contents of a shopping cart or the final itinerary of a travel plan before committing to it. Nevertheless, when considering a solution for service unavailability in SOA system, the above issue can be avoided using passive replication. Hence, this approach remains interesting as it differentiates between read and update requests, not only on the entire service state, but more specifically on "pieces" of the service state, i.e., data stored in *Tempestcollections*. Indeed, even though in the environment that we consider, we may not have multiple replicas, this differentiation enables to ease drastically the reconfiguration, meaning that only update requests have to be synchronized with the substitute service state. However, as services are not identical, and neither are the results of their read request, the client information has to be synchronized according to the service results. For instance, in case of a hotel booking service, a read request consists in providing the client with the room cost; thus substituting a hotel with another requires providing the client with the cost of the room in the substitute hotel.

In [Maamar et al., 2008], the authors consider the substitution of a service with another one that provides semantically similar functionalities, both belonging to the same community [3]. Communities are dynamic, where a master is elected and monitors the activities of the members of the community in terms of, e.g., joining and leaving the group. The master is further responsible for attracting and retaining services in a community using rewards [Bentahar et al., 2007]. When

[2] ACID: Atomicity, Consistency, Isolation, Durability.

[3] Group of similar services.

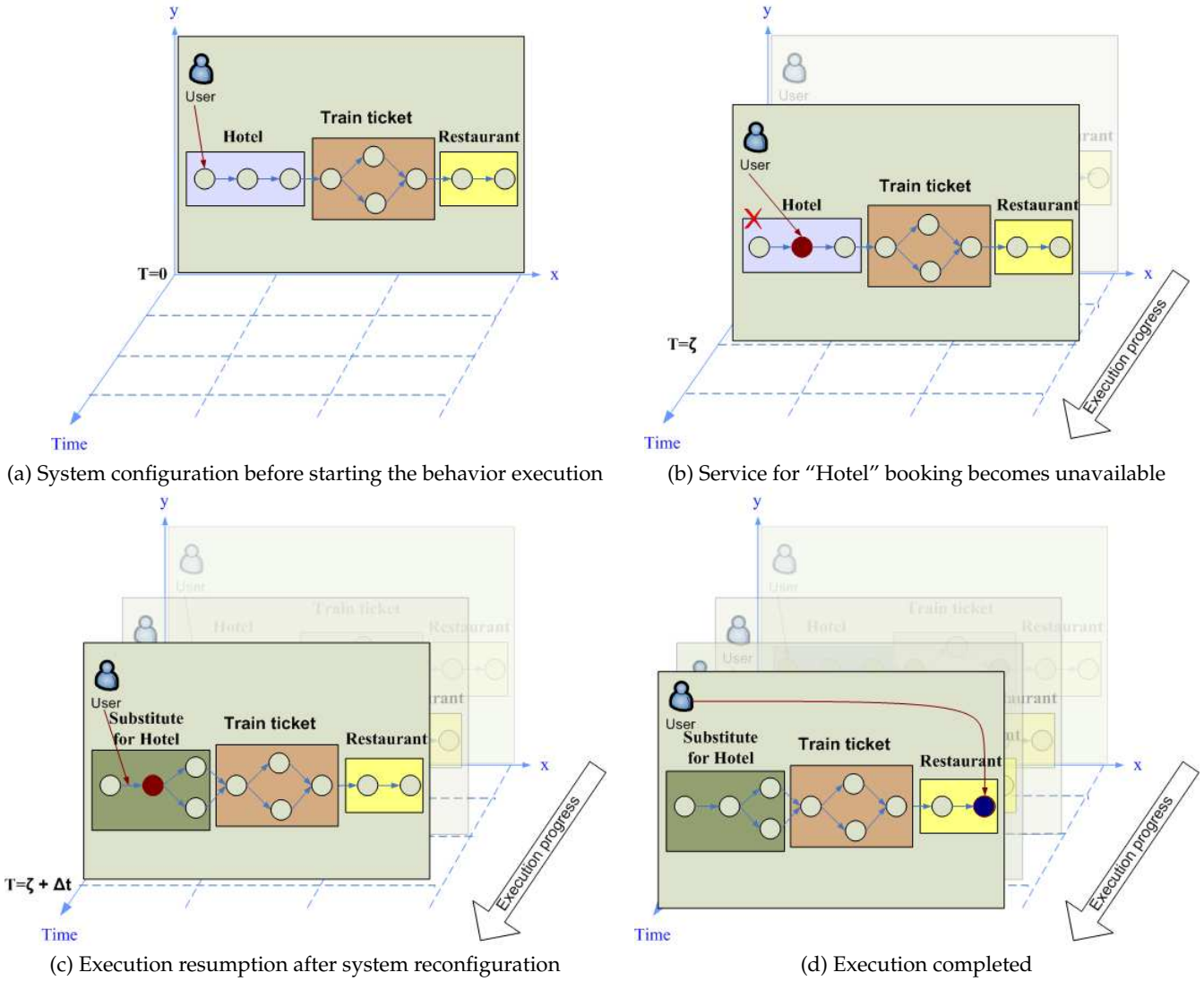
a candidate Web service provides a functionality that characterizes a community, the master Web service engages its provider in interactions. Some arguments are used during interactions, including high rate of participation of the existing Web services in compositions, and efficiency of these Web services in handling users' requests. Furthermore, retaining Web services in a community for a long period of time is a good indicator for the community: although the Web services in a community are in competition, they expose a cooperative attitude as they aim at participating as much as possible to service compositions. The service selection is directed with respect to the contract Net (CN) protocol, which is set as follows. The master sends to all slaves (i.e., the other Web services) of a community a call for bids, the slaves assess their status (i.e., whether they are busy in responding others users' requests), only services that are interested in the bidding contact the master. The latter considers the bids, identifies the best one, and notifies the interested services that are not selected that they play a role in backing-up the functionality of the selected slave. The back-up is defined as follows. At runtime, services maintain an operational flow, besides their control flow, which provides status information, i.e., whether they are activated, suspended, and alike. For a given functionality, services belonging to the related community support the same operational flow, but may differ in terms of control flow. The proposed approach synchronizes only the operational flow, and not the control flow. It follows an active replication relaxing the determinism constraint by accepting that backups can return similar results, i.e., partially different (overlapping), or totally different results. This approach is interesting in terms of managing and selecting the service substitutes according to their functionalities, which serves our need to efficiently find a substitute service for the one that becomes unavailable. However, as synchronization does not consider the control flow of the services, this approach is limited to services which state is not updated by the client requests. This approach assumes that any active operational state of the primary can be synchronized with a backup replica state even though they do not present the same control flow. However, services that change their state according to the control flow execution have to consider the synchronization of the control flows, besides the operational flows.

From the review of the existing approaches that deal with service unavailability, we see that traditional solutions that rely on replication could be used to a certain extent, and each solution has its pros and cons. Because of the existence of services offering similar functionalities in SOA-based environments, reliability of services-based applications could be achieved in a non-traditional way, using semantic-based service substitution. In Section 3.5, we present the requirements that have to be satisfied to order to achieve reliability in SOA systems.

## 3.5 Requirements for Runtime Reconfiguration of SOA Systems

Based on the review of the state of the art, we identify (1) the basic steps that have to be performed, and (2) the requirements that have to be satisfied, during the execution life cycle of a SOA system to realize runtime service substitution. Figure 3.2 illustrates a generic schema of runtime service substitution.

First, the SOA system –whether it is a simple client/service interaction or a service orchestration– starts its execution (Figure 3.2-a). At runtime, a service involved in the running SOA system may become unavailable (e.g., the hotel booking service) while executing the behavior that implements a specific functionality (Figure 3.2-b). The runtime reconfiguration then consists in finding a substitute service that provides a functionality semantically similar to the one of the substituted service. The behavior of the substitute service may be different from the one of the now unavailable service (Figure 3.2-c). The difficulty lies in :




(a) System configuration before starting the behavior execution

(b) Service for "Hotel" booking becomes unavailable

(c) Execution resumption after system reconfiguration

(d) Execution completed

Figure 3.2: System execution life cycle

1. Checkpointing the state of the now unavailable service before becoming unavailable. The checkpoints should be managed by the client's middleware to ensure state durability after service unavailability.

2. Clearly reasoning about whether a functionality may serve as a substitute for the one of the unavailable service.

3. Finding the point of execution of the substitute behavior where the execution resumption is possible and correct.

4. Translating and transferring the checkpointed state of the unavailable service to the substitute service in order to synchronize accordingly; if this fails then, a replay may be alternatively performed on the substitute service, which should be adapted with respect to the substitute service behavior.

5. In the case of service orchestration, identifying the impact of the substitution on the services

that are involved in the service orchestration, and restoring the orchestration consistency according to their data dependencies. Note that in the illustrative scenario in Figure 3.2, the workflows of the services "Hotel", "Train ticket" and "Restaurant" are integrated in a sequential structure to form the orchestration workflow. However, in the general case, services' workflows may interleave in the orchestration workflow.

Once the above steps are carried out, the SOA system execution can be resumed in order to terminate normally (Figure 3.2-d). Part II of the present thesis details how our approach overcomes the above listed difficulties.

## 3.6 Concluding Remarks

Our objective is to ensure continuity in service provisioning in SOA systems when a service becomes unavailable at runtime. To this aim, our main focus consists in substituting the unavailable service with another service that provides semantically similar functionality to the unavailable one, instead of aborting the execution of the SOA system.

In this chapter, we presented the basic concepts of dependability, focusing more specifically on service unavailability as a type of failure. We studied the existing techniques to tolerate such a failure in closed distributed systems, in order to emphasize their limits of applicability in SOA systems. The study of the existing techniques for closed distributed systems lead us to point out a need for a middleware-based approach that realizes runtime service substitution, with respect to services autonomy and environment dynamics and heterogeneity. We then reviewed a part of the existing efforts on service substitution in SOA systems, each of them focusing on specific facets of the issue. We pointed out the advantages of the existing efforts for dealing with service substitution as well as their limits of serving completely our needs. Finally, we established a set of requirements that have to be satisfied in order to realize semantic-based service substitution in SOA system, while restricting the least service autonomy and loose coupling.

In Part II, we define the constraints/rules under which the substitution between services is correct.

# Part II

# Formalization

*Reliability can be purchased only at the cost of simplicity*

Hoare [Hoare, 1969]

# 4

# Revisiting the Service Model

As stated in the previous chapters, service unavailability occurs more frequently as the computing environments gain in openness and dynamics. In this thesis, we target runtime service substitution in order to face service unavailability. However, in heterogeneous SOA-based environments, it hardly happens that the unavailable and substitute services are identically implemented or described. Runtime service substitution becomes then a complex task, specifically, when services implement composite behaviors and/or maintain a state at runtime. The substitution requires matching between the respective behaviors and states of the unavailable and substitute services.

To make this matching possible, we propose in this chapter a formal representation for composite behaviors, and service states. We then integrate this formalization into our advanced service model that enriches the basic one, introduced in Chapter 2.

The rest of this chapter is organized as follows. In Section 4.1, we provide a formal modeling for composite behaviors in order to enable reasoning and matching between the service behaviors. In Section 4.2, we detail the definition and the description of the service state. We also specify the set of requirements that have to be satisfied in order to enable state translation and transfer to the substitute service. In Section 4.3, we integrate the concepts introduced in Sections 4.1 and 4.2 into the basic service model presented in Chapter 2. The advanced service model serves the formalization of the service substitution in the next chapter.

## 4.1 Modeling Service Behavior

As discussed in Chapter 2, a composite behavior can be modeled as a workflow. The authors in [Van Der Aalst et al., 2003] provide thorough description and classification of the twenty most used workflow patterns, classified in six groups, namely, basic control flow, advanced branching and synchronization, structural, multiple instances, state-based, and cancellation patterns. Several efforts have been proposed to formally describe these workflow patterns using, e.g., $\pi$-calculus [Cicekli and Yildirim, 2000], Petri-Nets, flowcharts or statecharts. In this thesis, based on the efforts of Ben Mokhtar [Ben Mokhtar, 2007] and Wombacher *et al.* [Wombacher et al., 2005], we use the annotated finite state automata (aFSA) formalism to model a service behavior, as it serves our need for formalization and it is less complex than the above mentioned formalisms. Nevertheless, we can be easily apply any another formalism enabling workflows description. As specified in the first part of this thesis, we assume that services describe their composite behavior using a BPEL process. [Wombacher et al., 2005] presents how to transform a BPEL process into an aFSA to enable reasoning on and, potentially, transforming their workflows. It also presents the inverse transformation, in order to get back to an executable BPEL process. The definition of an aFSA [Wombacher et al., 2004] is described below.

### Annotated Finite State Automata

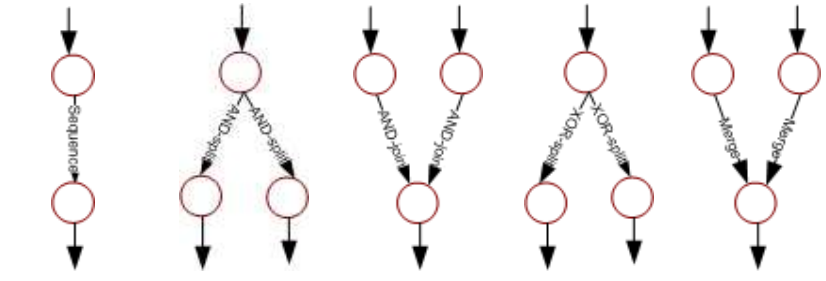
An aFSA is a six-tuple $(Q, \sigma, \delta, s_0, F, QA)$, where

- $Q = \{s_i, i \in [0..n]\}$ is a finite set of nodes (or states),
- $s_0 \in Q$ is the initial node,
- $F \subseteq Q$ is a set of final nodes,
- $\sigma$ is the alphabet annotating the transitions,
- $\delta : Q \times \sigma \times Q$ represent the labeled transitions, and
- $QA$ is a finite set of relations between nodes of $Q$.

A behavior is modeled as an aFSA, where the nodes (graphically presented using circles) denote the operations to be performed, and the transitions (arrows) denote the order of their execution, i.e., the control flow. Each transition is labeled with the related workflow pattern that links the two nodes. For instance, the transition that links two operations that have to be executed sequentially is labeled using the keyword *Sequence*. Hereafter, we describe the basic workflow patterns that we support in this thesis. Furthermore, final nodes are represented by double circles (with different diameters) within the automaton. The initial node is pointed out with an input arc. This initial node is connected using a *Sequence*-labeled transition to the first operation that triggers the behavior execution. This representation enables only to model the control flow. To include the data flow, the nodes of the aFSA are annotated with data dependencies (detailed below).

### Workflow patterns

Figure 4.1 presents graphically the five basic control flow patterns [Van Der Aalst et al., 2003] among the 20 most used workflow patterns, and Table 4.1 provides a brief description for each of them.




(a) Sequence (b) AND-split (c) AND-join (d) XOR-split (e) Merge

Figure 4.1: Basic workflow patterns

In the present thesis, we support only these basic control flow patterns when matching between the behavior of the unavailable service and the substitute service's one. Extending the matching to the others advanced workflow patterns can be performed by expressing them through the basic ones. For instance, the multi-choice pattern [1] can be constructed by combining the exclusive choice and parallel split patterns. Arbitrary cycles (i.e., loops) can be constructed as a

[1] A set of transitions where, based on a decision, a number of branches are chosen.

| Basic Workflow patterns | Descriptions |
|---|---|
| **Sequence** | It is represented by a transition that links two nodes: a predecessor and a successor. At runtime, the transition activates the execution of its successor node after the completion of the execution of its predecessor in the same workflow (Figure 4.1 - (a)). |
| **Parallel split** (also called AND-split) | It is represented by a set of transitions where a single sequence of control splits into multiple sequences of control (branches) which can be executed in parallel, thus allowing nodes (on the different branches) to be executed simultaneously or in any order. (Figure 4.1 - (b)). |
| **Synchronization** (also called AND-join) | It is represented by a set of transitions in the workflow where multiple parallel sequences of control synchronize and converge into one single sequence of control (Figure 4.1 - (c)). |
| **Exclusive choice** (also called XOR-split) | It is represented by a set of transitions where, based on a decision or data, one of several sequences of control (branches) is chosen (Figure 4.1 - (d)). |
| **Simple merge** | It is represented by a set of transitions where two or more alternative sequences of control (branches) come together without synchronization (Figure 4.1 - (e)). |

Table 4.1: Description of the five basic control flow patterns

repeated sequence of the sub-workflow pattern included in the cycle as long as the condition that direct the loop is satisfied. The verification of the condition can be realized using an XOR-split structure.

### Representing data flow

A data dependency appears when an output of an operation is an input of another. Assuming that workflow includes $N$ operation invocations, and each operation $Op_i$, $i \in [1..N]$, has a finite set of input parameters $I_i$, and a finite set of output parameters $O_i$. Both sets are included in $P$, which is a finite set of the overall parameters of the operations in a workflow. We have:

$$P = \cup_{i \in [1..N]} (I_i \cup O_i)$$

Thus, each node $s_i \in Q$ representing the invocation of the operation $Op_i$ is modeled using a three-tuple included in $P \times OpId \times P$, where $OpId$ is the finite set of the $N$ operations' names. A data dependency between two nodes $s_i = (I_i, Op_i, O_i)$ and $s_j = (I_j, Op_j, O_j)$ is expressed using the following rule:

$$Op_i \text{ depends upon } Op_j \Rightarrow \exists o \in O_j, \exists k \in I_i \text{ such that } o = k$$

The data dependency is denoted $(s_j, \{o\}) \rightarrow (s_i, \{k\})$. In the case that the node $s_j$ presents multiple data dependencies with the same or other nodes, we model them using a conjunction and disjunction of data dependencies. A conjunction (denoted $\wedge$) is used when the forward path (in the control flow) of a node $s_j$ includes, e.g., an "AND-split" pattern, where one or multiple nodes in the parallel branches of the "AND-split" consume one of the outputs of $s_j$. A disjunction

(denoted $\vee$) is used in the case of multiple alternatives in the forward path of the node $s_j$, for example, in the case of an "XOR-split" pattern, when multiple nodes in the parallel branches of the "XOR-split" consume one of the outputs of $s_j$. We model the data dependencies in $QA$ using a generative grammar, formalized by Noam Chomsky [Chomsky, 1956]. Let $G$ be the grammar that generates correct representation of data dependencies, $G$ is modeled as 4-tuple $(N, \Sigma, P, S)$ where

- $N$ is the non-terminal symbols, such that $N = \{D, R\}$, where $D \in (Q \times P) \rightarrow (Q \times P)$ represents a data dependency between two nodes in $Q$, and $R \in \{\vee, \wedge\}$ represents the relation of disjunctions or conjunction between data dependencies.
- $\Sigma$ is the set of terminal symbols, which is reduced to the empty string, denoted $\varepsilon$.
- A symbol $S \in N$ that is the start symbol.
- $P$ represents the generation rules:

$$S \rightarrow D$$
$$D \rightarrow D\ R\ D$$
$$D \rightarrow \epsilon$$

Hence, $QA$ is a sub-set of all the expressions generated by the grammar $G$. Graphically, the data annotations are represented using dashed transitions between the dependent nodes. The transitions are labeled with the set of parameters that are involved in the data dependency.

**Illustrating example**

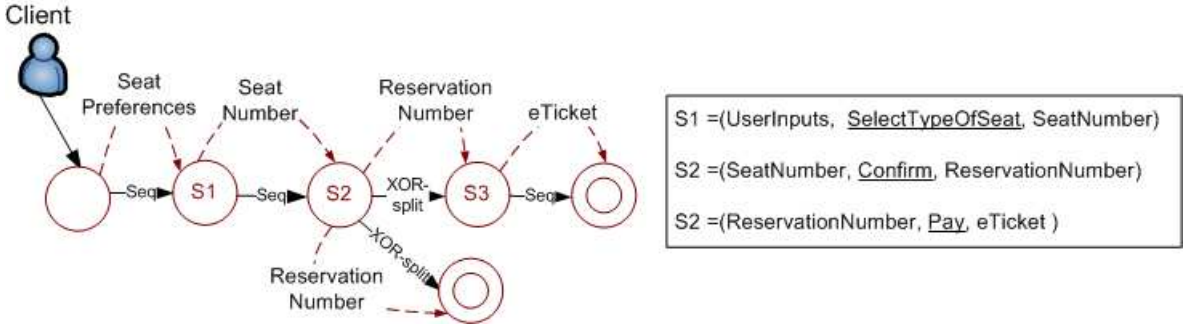



Figure 4.2: Graphical representation the aFSA for the train ticket booking behavior

Considering the example of the train ticket booking described in Section 2.4.4, we model its behavior as an aFSA (graphically represented in Figure 4.2). The aFSA includes besides the initial and final nodes, three other nodes: $S_1$, $S_2$ and $S_3$, corresponding respectively to the operations: 'SelectTypeOfSeat', 'Confirm' and 'Pay' (underlined terms in Figure 4.2). As presented in Chapter 2, the operation 'SelectTypeOfSeat' takes as input the seat preferences and returns a seat number, we denote $S_1 = (SeatPreferences, SelectTypeOfSeat, SeatNumber)$. This seat number is taken as input by the operation 'Confirm' which returns a reservation number. The user can then decide whether to pay for the reservation using the operation 'Pay' and be provided with an 'eTicket', or to pay his/her reservation later. In the latter case, s/he will be provided with her/his reservation number.

## 4.2 Service State

Respecting service autonomy in implementing its functionalities, we assume, in the present thesis, that the functionality of the substitute service may implement a different behavior from the unavailable service's one. In the previous section, we formalized the modeling of composite behaviors in order to enable reasoning and matching between service behaviors. Still, behavior matching is not sufficient to ensure reliability of SOA systems. To enable state transfer, a state translation may be required. In this section, we formalize the notion of service state. In Section 4.2.1, we provide an overview of the service state. In Section 4.2.2, we present how we generate the service state description. The generation of the service state description requires the definition of checkpoints in SOA. In Section 4.2.3, we present the specifics and different types of checkpoints that a service may define. As the state of the service requires to be translated and transferred to the substitute service, in Section 4.2.4, we present the required operations that enable the state access and manipulation.

### 4.2.1 Overview

In this thesis, we share the same point of view as the WS-Resource standard regarding advertising the service state to the external environment. In Section 2.3, we defined the service state when executing a given behavior using 3 parts:

1. The workflow or observable state: represents the data observable by the client, which are included in the data flow of the behavior.

2. The implementation state: represents the data that are manipulated by the service implementation, at runtime.

3. The resources state: represents the external data, which can potentially be shared among multiple running instances.

We gather the two last parts of the state (i.e., implementation and resources) into a single part, which we call "internal state", as opposed to the workflow/observable state that can be accessed and manipulated by external entities.

More generally, looking beyond the Web services domain, several efforts have been investigating the concept of a state for computer systems. A comprehensive literature review [Peti, 2002] has compared the notion of state from a diverse range of engineering disciplines. Among them, the conceptual model of Dependable System of Systems (DSoS) [Gaudel et al., 2003] proposes two styles for the 'state definition' namely, *backward-looking* and *forward-looking* styles.

In the *backward-looking* style, the state of a system at a given time $t$ is the *total data* explicitly stored by the system in the time interval [2] $[0..t]$. This definition corresponds to the view of a system as an entity that stores information about its interaction with the environment, and uses the stored information to influence its future results. This concept of state is therefore often called "internal state" [3]. This style can be used to define the state of the service when a rollback can be perfomed. Indeed, the state of the service then includes all data generated and manipulated at specific steps of its execution, which are required to synchronize the state of the service with one of its previously reached states.

[2] Assuming that the system execution has started at $t = 0$.

[3] In DSoS, the internal state of a system consists of the values explicitly stored in state variables (i.e., variables related to the environment or to the computer system, which value may change as the execution progresses).

In the *forward-looking* style, the state of a system at a given time $t$ includes the *sufficient data* that enable determining the system possible behavior after the given instant, including the responses to possible future invocations of the service operations. The forward-looking defined state is thus a sub-part of the backward-looking defined state, from which it removes the consumed data that will not be used after the time $t$. This style can be used to define the state of the service when a state transfer has to performed. Using this style enables to transfer the part of the state that is sufficient for the substitute to resume the execution. This is relevant only when the substitute and unavailable services are identical. However, as services may not identical, the substitute service may require data that are not included in the forward-looking defined state. Hence, to avoid removing data that may be useful for subtitute services, we use in all cases the backward-looking style in order to define the state that has to be transferred to the substitute service.

### 4.2.2 Service State Description

Following the WS-Resource representation of the service state (Section 2.4.5), we model the service state as a set of tuples $state =< Value, Variable >$, where each variable has a predefined name, type and semantic concept.

In this section, we target to provide the means to generate the description of the service state. We envision to associate (before runtime) a state description with each specific step of execution. In the aFSA that models a composite behavior, an execution step corresponds to a specific node $n$ of the aFSA. We denote $SD(n)$ the state description at the node $n$.

In the following, we use the term executing node $n$ to denote the execution of the operation that is related to the node $n$, which is by definition an atomic unit of execution. Thus, the state of the service does not change, but after completing the execution of the node $n$. While processing the node $n$, the state description at $n$ corresponds to the state of the service before performing the node $n$, i.e., the state of the service from the initial node, till the node $n$ ($n$ excluded).

#### Generating the description of the workflow state

To describe the workflow state (denoted $SD_{wkf}$) at a node $n$, we consider all the required inputs of the workflow operations that are included in the path from the node $n$ and onwards (including $n$'s inputs). In case of multiple paths leading from $n$ to a final node in the workflow, we consider all the paths that may be executed, since we cannot predict, before runtime, which path will be actually executed.

Using the notation presented in Section 4.1, let $b = (Q, \sigma, \delta, s_0, F, QA)$ be the behavior of a service $s$. Let $N = \{n_k = (I_k, Op_k, O_k)\}_k$ be the set of nodes located on all the path linking the node $n \in Q$ ($n$ included) to the final nodes in $F$. In case of multiple paths leading from $n$ to a final node in the workflow, we consider all the paths that may be executed, since we cannot predict, before runtime, which path will be actually executed. $I_k$ represents the set of required inputs of the operation $Op_k$, and $O_k$, the set of provided outputs. Let $N^{-1}$ be the set of nodes located on the path that links the initial node $s_0$ of $b$ to the node $n$ ($n$ excluded).

We use the backward-looking style for describing the workflow state. We define the description of the workflow state $SD_{wkf}$ at $n$ includes all the in/out parameters of the operations $Op_k$ available at $n$ (i.e., $Op_k \in N^{-1}$). By available parameters, we mean the parameters that are resulting from the outputs of the previously performed operations, the user inputs and the environment context. The forward-looking style requires to remove all the parameters that are consumed by the nodes of $N^{-1}$, and which will not be used by the nodes included in $N$. This

is not performed when defining $SD_{wkf}(n)$ in order to avoid removing data that may be required by the substitute service.

We propose to generate the description of the workflow state automatically at each node. Algorithm 1 presents the high level instructions that generate this description.

**Algorithm 1**: Generating the description of the workflow state at a node n

```
Data: b and n.
Result: SD_wkf(n).
begin
    /*  Initialization phase                                              */
    SD_wkf(n) =empty
    s = s_0
    /* j is a variable that counts the j^th successor of s               */
    j = 1
    while True do
        forall s_i ∈ j^th successor of s such that s_i = (I_i, Op_i, O_i) ∈ N^-1 do
            /*  Verifying that n is not reached                           */
            if s_i ≠ n then
                SD_wkf(n) = SD_wkf(n) ∪ {s_i.O_i}
                foreach k ∈ I_i do
                    /* Taking only the inputs that are not included in SD_wkf(n) */
                    if (k ∉ SD_wkf(n)) then
                        SD_wkf(n) = SD_wkf(n) ∪ {k}
            else
                Return SD_wkf(n)
        j++;
end
```

Algorithm 1 takes as inputs: the aFSA description of the behavior, and the node at which the description of the workflow state is required. It initializes the description of the workflow state $SD_{wkf}(n)$ to *empty*, and the temporary variable $s$ to $s_0$, in order to start the aFSA traversal with the initial node of the workflow. Then, for all the nodes $s_i$ which are successors of $s$ and which are included in $N^{-1}$, Algorithm 1 adds the output parameters of $s_i$ to the $SD_{wkf}(n)$. To avoid redundancy in $SD_{wkf}(n)$, Algorithm 1 verifies first whether the input parameters of $s_i$ have been previously added to $SD_{wkf}(n)$, or not. In the latter case, it adds them to $SD_{wkf}(n)$. Note that we use a variable $j$ (initialized to 1) to count the $j^{th}$ successor of $s$, which denotes that there are $(j-1)$ nodes that separate the node $s$ from its $j^{th}$ successor. The set of instructions of Algorithm 1 are put in a loop with a guarding condition that enables to reach the end of the algorithm only when all the nodes in $N^{-1}$ are processed, i.e., when $s_i$ reaches the node $n$.

### Internal state description

The internal state is tightly coupled with the service implementation. Thus, it is hardly re-usable by the substitute service. Indeed, finding an exact replica (i.e., identical implementation) of the unavailable service is too optimistic, and poses a real constraint on substitute discovery. It is then useless to transfer such a state to the substitute service. However, as mentioned in Chapter 3, a service may be forced to rollback to a previous state to restore the whole SOA system consistency (i.e., the client and the service(s)). Therefore, storing the internal state locally at the service side enables to get back to a previous state of the service and invalidate a set of results of the behavior execution.

Based on the above, in our work, we only need to externalize the description of the workflow state at a node $n$, which corresponds to the observable part of the state. This description of the workflow state should be sufficient to a substitute service to "understand" the transferred state, and the data transferred should be sufficient to synchronize its internal and observable states in order to resume the interaction with the client from the point it was interrupted. These constraints recall naturally the notion of checkpoint, presented in the previous chapter, and lead us to define the notion of checkpoint for services.

### 4.2.3 Checkpoint Definition

To define the notion of checkpoint in the domain of services, we require to structure the composite behaviors in terms of atomic actions [Campbell and Randell, 1986]. An atomic action is an indivisible part of the workflow (i.e., one or a structured set of operations) that performs an integral piece of computation atomically, meaning that, the effects of an atomic action execution are "committed" only when it terminates normally, reflecting an "all-or-nothing" execution.

The atomicity property is necessary in order to make the substitution consistent. Indeed, as explained in Chapter 3, we cannot expect that services can resume their execution at any point of their workflow, regardless the workflow structure and the execution progress. Structuring the workflow into atomic actions argues the fact that the state of the service has to be checkpointed at the end of the execution of atomic actions. The semantic meaning of a *checkpoint* represents then the state of a service at which the middle results are committed, and which can be reused by other services, notably substitute services. The execution of an atomic action is then delimited by two checkpoints (at its beginning and its end).

#### Illustrating example

The use of atomic actions can be illustrated in the case of the train ticket booking functionality, the behavior of which includes three operations, namely, 'SelectTypeOfSeat', 'Confirm' and 'Pay'. Consider the case that the user has entered his/her seat preferences and confirmed his/her reservation, but the 'eTicket' has not been edited yet. If a problem occurs (e.g., a user disconnection due to the train departure) at this stage of execution, then the ticket booking cannot be transferred to another service, because the substitute service may not access the same data base (SeatDataBase) as the service that becomes unavailable. However, the seat preferences can be transferred to a substitute service in order to spare the user of entering again his/her preferences. Hence, the workflow of the train ticket booking functionality can be decomposed into two atomic actions: (1) one atomic action that includes the selection of the user preferences, and (2) another atomic action that includes the operations for confirming and paying the reservation.

However, considering the case that *Service A* [4] is involved in a service composition, and that another service participating in the service composition becomes unavailable, *Service A* may be forced to roll a set of operations back to a previous checkpoint due to data dependencies with the unavailable service. In this case, the checkpoint does not serve to synchronize a substitute service, but it used by the same service instance that generated the checkpoint. Hence, in such a case, the checkpoint must include, besides the workflow state, the internal state of the service, to enable the service getting back to one of its previous states.

Based on the above, we distinguish two different types of checkpoints: *state transfer* and *rollback* checkpoints.

[4] The service that has been introduced in Chapter 2, and which provides the train ticket booking functionality.

**Types of checkpoints**

- **State transfer checkpoint**: This type of checkpoints serves to synchronize the state of the substitute service (i.e., internal and observable) according to the data included in the checkpoint. These checkpoints should be provided with a description of the workflow state. At runtime, these checkpoints store a copy of the data that are included in the workflow state in order to be used when a state transfer is required.
- **Rollback checkpoint**: This type of checkpoints is applied on, and used by, the same service instance that generates the checkpoints, in order to invalidate a set of middle results and get back to a previously reached state of the execution. Thus, they do not require a state description to be externalized. At runtime, these checkpoints store a copy of all the data maintained and manipulated by the service till the time of performing the given checkpoint; they include both the workflow and the internal states of the service. Regarding the resources that are shared among multiple instances, the rollback checkpoint does not include the whole resources data, but only the data manipulated from the beginning of the execution till the time of taking the *rollback* checkpoint.

In general, we find more *rollback* checkpoints in a behavior definition than *state transfer* ones. *State transfer* checkpoints can be performed at the end of atomic actions. *Rollback* checkpoints can be performed within atomic actions, e.g., using split-transactions [Kaiser and Pu, 1992]. More specifically, we notice that *state transfer* checkpoints represent a sub-set of *rollback* checkpoints where the service externalizes a copy of the workflow state to the external environment, in addition to storing its complete state internally.

### 4.2.4 Service State Access and Manipulation

To enable state transfer, we require that composite stateful behaviors enrich their description with special nodes that correspond to *state transfer* and *rollback* checkpoints. We then generate automatically, for each *state transfer* checkpoint, the corresponding description of the workflow, using a wsrp document (presented in Section 2.4.5). Also, we require that services implementing stateful behaviors support and advertise a set of operations that enable to manipulate the service state. In particular, we require three *recovery* operations: *GetState*, *SetState* and *Rollback*.

- *GetState* recovery operation enables to get a copy of the workflow state and store it on a persistent storage. Based on the WSRF standard, when reaching a *state transfer* checkpoint, the client can send a *GetResourceProperty* query to the service, with the description of the workflow state, and the client identifier as input parameters. The service response provides a copy of the workflow state at the invocation time.
- *SetState* recovery operation can be performed at *state transfer* checkpoints. *SetState* recovery operation synchronizes the state of the service with a given *workflow* state.

  After invoking a *SetState* operation, the substitute service replies to the requester (in our approach, the client middleware) with an acknowledgment denoting a successful synchronization of the service state according the transferred state. Otherwise, it replies with an error message.
- *Rollback* recovery operation can be performed at *rollback* checkpoints. *Rollback* checkpoints do not require a client request in order to be performed. They are performed autonomously by the service. The client is able to invoke *Rollback* recovery operation in order to get back to a pervious *rollback* checkpoint.

### Modeling checkpoints and recovery operations

*GetState* and *SetState* operations have the same semantic meaning as the concepts 'QueryState' and 'UpdateState' defined in the 'OperationImpact' ontology (Figure 2.16). We further enrich the 'OperationImpact' ontology with a 'Rollback' concept, that serves for semantic annotations of *Rollback* recovery operations, as presented in Figure 4.3. Thus, to avoid naming restrictions, services providing recovery operations that enable state access and manipulation, may annotate them with 'QueryState', 'UpdateState' and 'Rollback' concepts.

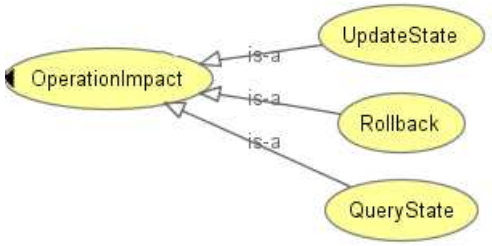



Figure 4.3: OWL ontology for distinguishing the recovery operations

Furthermore, we enrich the aFSA definition with a set of *state transfer* checkpoints and *rollback* ones. The behavior is the modeled as an eight-tuple aFSA = $(Q, \sigma, \delta, s_0, F, QA, STCkpt, RCkpt)$ which maintains the same definition for $Q$, $\sigma$, $s_0$, $F$, and $QA$ as in Section 4.1, and enriches or redefines the others as follows:

- $STCkpt$ includes the set of *state transfer* checkpoints that are modeled as nodes that enable to invoke either *GetState* or *SetState* recovery operation, or both of them.
- $RCkpt$ includes the set of *rollback* checkpoints that are modeled as nodes that enables to invoke a *Rollback* operation.
- The nodes representing checkpoints, which are included in $STCkpt$ and $RCkpt$, are connected to $Q$'s nodes using the transitions defined in $\delta$. In this way, $\delta$ is redefined as follows.

$$\delta : (Q \cup STCkpt \cup RCkpt) \times \sigma \times (Q \cup STCkpt \cup RCkpt)$$

Graphically, the recovery nodes are integrated in the aFSA using symbols different from the ones used for $Q$'s nodes:

- A square is placed on the transition that links two circles representing two $Q$'s nodes, to denote a *rollback* checkpoint.
- A tick (checkmark) is placed on the transition that links two $Q$'s nodes, to denote a *state transfer* checkpoint.
- A square with tick is placed on the transition that links two $Q$'s nodes, to denote a *rollback* and a *state transfer* checkpoints.

In order to leave the choice to the client whether handling service reliability or not, the set of *state transfer*-supported checkpoints are connected to the other nodes of the aFSA using "XOR-split" and "Merge" workflow patterns. In this way, the client may choose to perform a sub-set of the possible *state transfer* checkpoints enabled by the service, all of them, or none of them. Also, the set of supported-*rollback* checkpoints are connected to all their successors using an "XOR-split" transition in order to allow getting back from any node to a previously performed *rollback* checkpoint.

**Illustrating example**

To illustrate the integration of the recovery operations within the behavior modeling, we employ the train ticket booking functionality, the workflow of which is described in Figure 4.2.

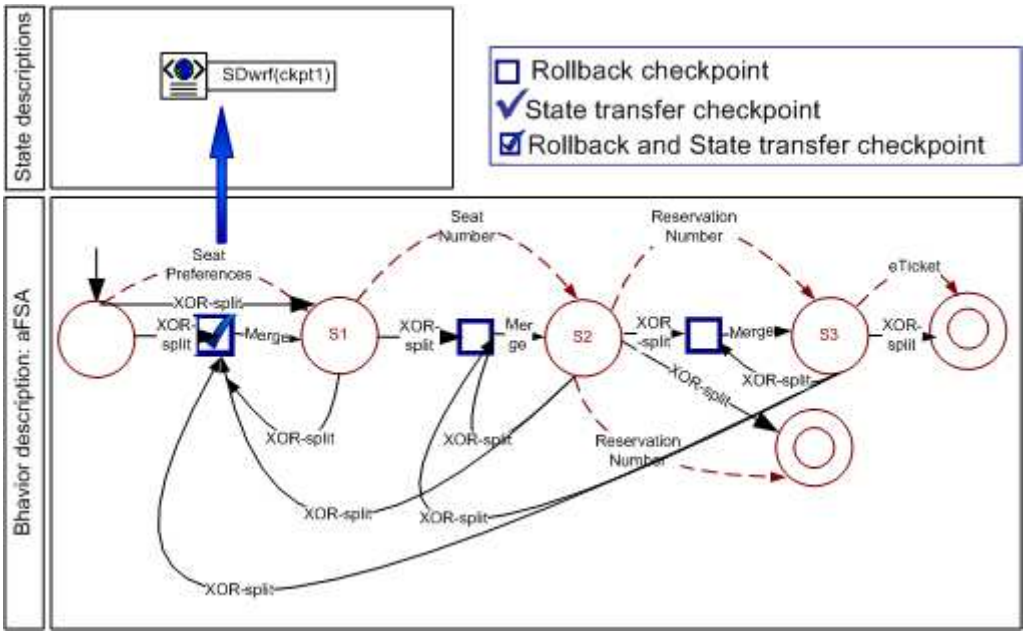



Figure 4.4: Modeling checkpoints in the train ticket booking behavior

As presented earlier, the workflow of the train ticket booking is composed of two atomic actions. As depicted in Figure 4.4, the first atomic action is delimited with a state transfer checkpoint, which is provided with a description of the workflow state $SD_{wkf}(ckpt1)$ that includes the user preferences. The state transfer checkpoint represents also a rollback checkpoint. Other rollback checkpoints within the second atomic action are defined in order to store the seat selection and the reservation number.

As the behaviors may differ from one service to another, the recovery operations of the unavailable and substitute service may not coincide. In our approach, we tolerate such heterogeneity in terms of recovery operations, allowing services to define their checkpoints at their convenience. This requires to match between recovery operations when matching between service behaviors.

## 4.3 Advanced Service Model

In this section, we propose to enrich the basic service model introduced in Chapter 2 with new concepts necessary for handling runtime service substitution. These concepts are retrieved from the behavior and state modeling, presented in the previous sections.

In the class diagram of the advanced service model (Figure 4.5), a workflow is represented as a composition of (1..*) atomic action(s), which can be atomic or composite behaviors. In this way, atomic actions is a sub-type of the concept behavior. Furthermore, based on the behavior modeling presented in Section 4.1, the workflow description is modeled as an aFSA that is composed of (2..*) nodes, including the initial and final nodes. As presented in Section 4.2.4, these nodes can be associated with functional operations or checkpoints. The checkpoints can be of two types: *state transfer* or *rollback*, respectively invoking *Get/SetState* recovery operations

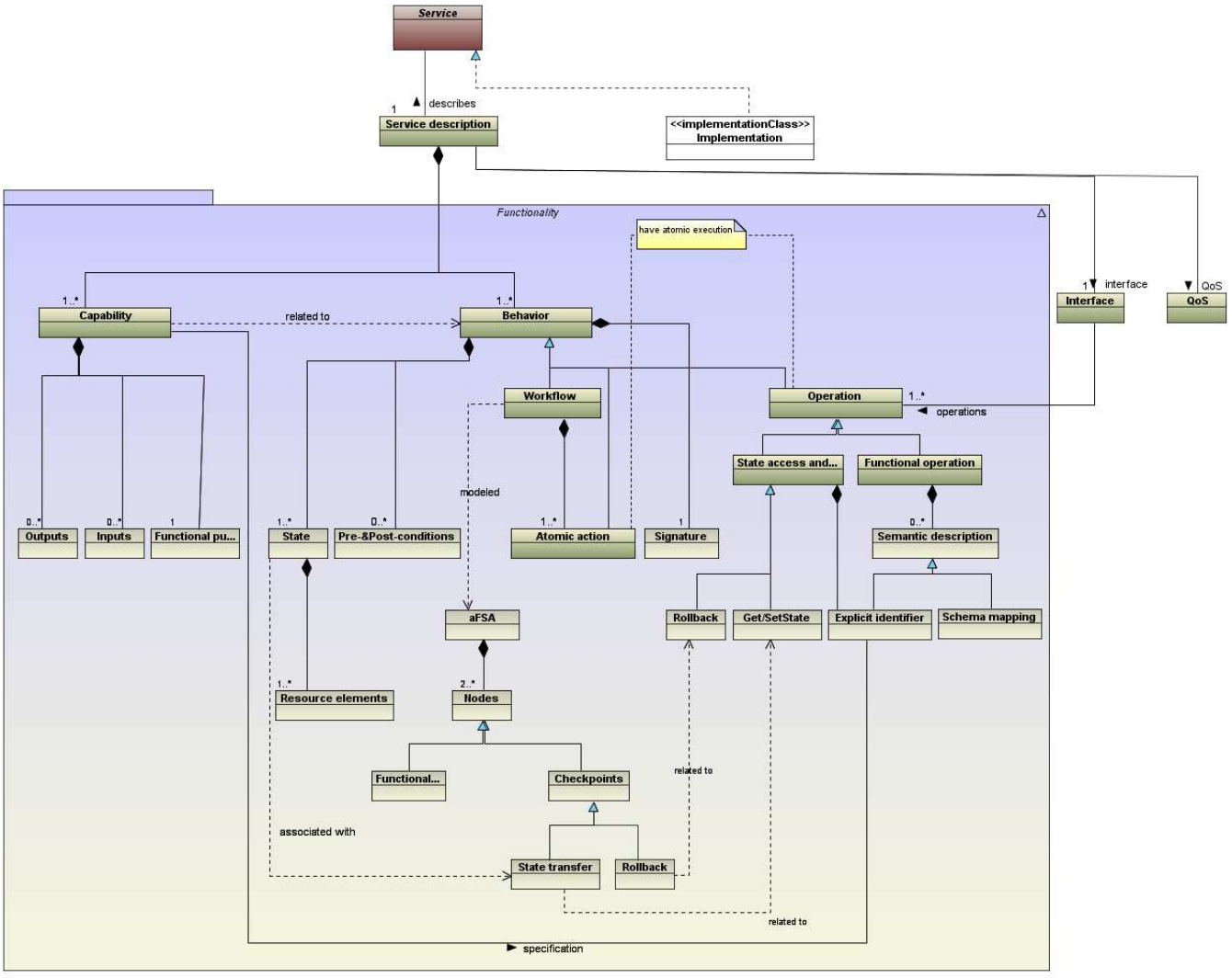



Figure 4.5: Service Class diagram of the advanced service model

or a *Rollback* one. We further associate a description of the workflow state with *state transfer* checkpoints. Consequently, a stateful behavior does not have a single description of the workflow state, but as many descriptions as *state transfer* checkpoints.

In Chapter 2, we presented the concept of signature for atomic behaviors (i.e., operations), herein we extend it to composite behaviors. A behavior signature comprises the behavior name along with the name and data types of the in/out and error parameters of the behavior.

Also, a capability is associated with a behavior description through the relation 'related to'. In particular, this relation associates the behavior signature's concepts with the semantic concepts that are included in the capability (as presented in Figure 4.6).

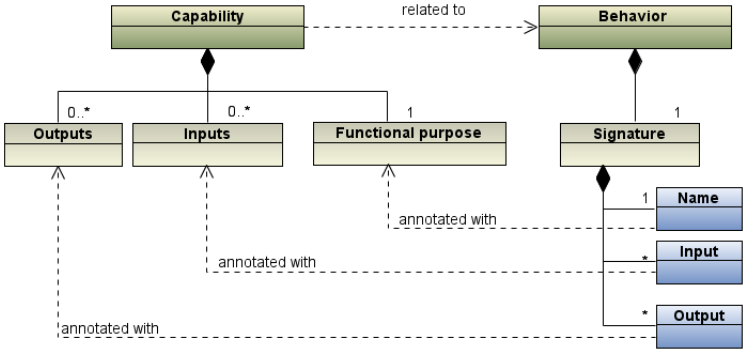



Figure 4.6: Relating capabilities and behaviors descriptions

## 4.4 Concluding Remarks

As the substitution of stateful composite behaviors poses a real challenge for runtime service substitution, in this chapter, we essentially focused on modeling the basic concepts that the stateful composite behaviors include, namely, the behavior and the state. The behavior is modeled using aFSA in order to provide means to reason on, and potentially, transform the workflow structure at runtime. We also concentrate on the definition of the service state. We defined the service state as a two-part state: workflow and internal states. The workflow state represents all the data that are observable and accessible by external entities to the service. The internal state is related to the internal logic of the service implementation and resources, and thus it is not accessible by the external entities. More important, it cannot be reused by the substitute service in the case of state transfer. To enable state transfer, we use the notion of atomic actions that split the workflow into parts that have to be executed integrally and atomically. Between two atomic actions, the observable state of the behavior can be used to synchronize a substitute service, which we call *state transfer* checkpoint. In addition, services may provide the opportunity to rollback a set of results, by defining a set of *rollback* checkpoints. To take benefit from the checkpoints definition in performing the runtime service substitution, we introduce a set of recovery operations that enable to access and manipulate the state maintained by a service, synchronize the substitute service, and in case of service composition, restore the consistency of still-connected services. All the newly introduced concepts are integrated in our service model, to form an advanced service model, in order to provide a complete view of the service concepts and their relation with each other.

In Chapter 5, we use the advanced service model to formally define the relation of service substitution between services.

*While FORMAL schooling is an important advantage, it is not a guarantee of success, nor is its absence a fatal handicap.*

Ray, Kroc

# 5 Formalizing Service Substitution

In this chapter we target to provide the set of rules that have to be satisfied in order to perform correct service substitution at runtime. To this aim, we split the issue into three parts:

1. The first one includes the rules that ensure the ability of a service to service as a substitute for another.
2. The second ensures the ability of a service to *take over* the computation performed by the unavailable service at runtime.
3. The third includes finding a matching between the behaviors of the substitute service and the unavailable service's one, that complies with the above rules. The matching enables to find a *state transfer* checkpoint at which the substitute service can synchronize its state and resume the execution initially started by the unavailable service.

Using the concepts included in the advanced service model of the previous chapter, we deal in Section 5.1 with the two first parts of the issue. We provide the rules that have to be satisfied in order to perform a correct runtime service substitution. Then, in Section 5.2, we deal with the third part of the issue. We present the strategies that identify at runtime the point at which the service state can be transferred to the substitute service. Finally, Section 5.3 summarizes the results established in this chapter, and organizes them into an algorithm for runtime service substitution.

## 5.1 Formal Definition of Service Substitution

The use of hierarchy is important when considering the issue of substitution. It allows the use of service groups as type families, in which a group of services (subtype) may serve as substitute candidate for a service included in another group (supertype) that is higher in the hierarchy. In this section, we first present the basic principles of subtyping that have been formalized for object-oriented systems (Section 5.1.1), emphasizing the need for adapting the formalization to SOA systems in order to serve the substitution purpose. In our adaptation (Section 5.1.2), we map the specification of subtyping issued from object-oriented design (OOD) to SOA systems specifics using the concepts introduced in our advanced service model. The adaptation of subtyping enables to classify services according to the functionalities they offer. Nevertheless, this is not sufficient to ensure correct *runtime* substitution of services. Indeed, as behaviors of the substitute and unavailable services are not identical, a need for matching between these behaviors is additionally required: the issue consists in identifying the rules under which a matching can be defined. Hence, in the third part of this section (Section 5.1.3), we

use Hoare's logic [Hoare, 1969] in order to introduce additional constraints that ensure correct runtime substitution.

### 5.1.1 Principles of Subtyping in Object-oriented Design

One feature is almost universally present in object-oriented systems [Cardelli and Wegner, 1985]: subtyping. Subtyping captures the intuitive notion of inclusion between types. Subtyping is independently specified, and a single rule, called *subsumption* [Cardelli, 1997], is added to connect the type to the subtype. The subsumption rule states that if an object has a type $\sigma$, such that $\sigma$ is a subtype of $\tau$ (denoted $\sigma < \tau$), then the object also has a type $\tau$. This allows an object to be used flexibly in many different typed contexts. Subtyping thus behaves as set inclusion, where type membership is seen as set membership.

The subtype relation between functions, where a function is a program that consumes a set of inputs $A$ in order to produce a set of outputs $B$, says that $F : A \rightarrow B$ is a subtype of $F' : A' \rightarrow B'$ if $A'$ is a subtype of $A$ ($A' < A$), and $B$ is a subtype of $B'$ ($B < B'$). Note that the inclusion is inverted (contravariant) for function arguments, while it goes in the same direction (covariant) for function results. More generally, within the type system of a programming language, an operator from types to types is *covariant* if it preserves the ordering ($<$) of types, which orders types from more specific ones to more generic ones; it is *contravariant* if it reverses this ordering. If neither of these apply, the operator is *invariant*. This distinction is important when considering argument and return types of methods in class hierarchies. The definition of covariance and contravariance in the case of classes is set as follows. In OOD, if class $B$ is a subtype of class $A$, then all the methods of $B$ must return the same or narrower set of types as $A$. The return type is said to be *covariant*. On the other hand, the methods of $B$ must take the same or broader set of arguments compared with the methods of A; the argument type is said to be *contravariant*. The problem for instances of $B$ is how to be substitutable for instances of $A$. The only way to guarantee type safety and substitutability is to be equally or more liberal than $A$ on inputs, and to be equally or more strict than $A$ on outputs. In this context, Liskov and Wing in [Liskov and Wing, 1994] have introduced a behavioral notion of subtyping. Hereafter, we present the important definitions and theorems that have been proved for subtyping in order to apply them in SOA systems.

#### Subtyping formalization in OOD

Liskov and Wing [Liskov and Wing, 1994] define a type as follows. Let $Obj$ a set of all potentially existing objects, partitioned into disjoint typed sets. Each object has a unique identity. A type defines a set of values for an object and a set of methods that provide the only means to manipulate that object. $Obj$ represents a set of unique identifiers for all objects that can contain values. A state defines a value for each existing object. It is a pair of mappings, an environment and a store. An environment maps program variables to objects; a store maps objects to values.

- State = Env x Store
- Env = Var $\rightarrow$ Obj
- Store = Obj $\rightarrow$ Val

Given a variable $x$ and a state $\rho$ with an environment ($\rho.e$) and store ($\rho.s$), the notation $x_\rho$ denotes the value of $x$ in the state $\rho$.

A type is modeled as a triple $< O, V, M >$, where $O \subseteq Obj$ is a set of objects, $V \subseteq Val$ is a set of values, and $M$ is a set of methods. The type specifications includes the following information:

- The type's name;
- A description of the type's value space from which the variables take their value;
- For each of the type's methods:
  - Its name;
  - Its signature (including signaled exceptions);
  - Its behavior in terms of pre-conditions and post-conditions,

The relation of subtyping requires that any property proved about the supertype service holds for its subtype service, which is formalized as follows. Let $\varphi(O_\tau)$ be a property provable about objects $O_\tau$ of type $\tau$. Then, $\varphi(O_\sigma)$ should hold for objects $O_\sigma$ of type $\sigma$, where $\sigma$ is a subtype of $\tau$, denoted $\sigma < \tau$.

Liskov and Wing are interested only in safety properties (i.e., "nothing bad happens"). First, the authors define properties that preserve object's behavior in specific program: these properties ensure that a program continues to work as expected, i.e., calls of methods made in the program that assume the object belongs to a supertype must have the same behavior when the object actually belongs to a subtype. These properties are called *behavioral properties*. In addition, the authors define *independent properties* which have to be preserved when independent programs share the same objects. The authors focus on two kinds of such properties: *invariants*, which are properties true over all states, and *history* properties, which are properties true for all sequences of states. Invariants are formulated as predicates over single states and history properties, over pairs of states.

The definition of the subtyping rule $<$ is then set as follows. $\sigma =< O_\sigma, V_\sigma, M_\sigma >$ is a subtype of $\tau =< O_\tau, V_\tau, M_\tau >$ if there exists an abstraction function, $A : V_\sigma \rightarrow V_\tau$, and a renaming map, $R : M_\sigma \rightarrow M_\tau$, such that:

1. The abstraction function complies with the invariants.
   - Invariant Rule:
     Let $I_\sigma$ and $I_\tau$ be respectively, the invariants of the types $\sigma$ and $\tau$, $\forall s : V_\sigma$, the rule $I_\sigma(s) \Rightarrow I_\tau(A(s))$ must hold, where $A$ may be partial, need not be onto, but can be many-to-one.
2. Subtype methods preserve the behavior of the supertype ones. If $m_\tau$ of $\tau$ is the corresponding renamed method $m_\sigma$ of $\sigma$, the following rules must hold:
   - Signature's rules:
     - Contravariance of arguments.
       $m_\tau$ and $m_\sigma$ have the same number of arguments. If the list of argument types of $m_\tau$ is $\alpha_i$, and the list of argument types of $m_\sigma$ is $\beta_i$, then $\forall i,\ \alpha_i < \beta_i$.
     - Covariance of results.
       Either both $m_\tau$ and $m_\sigma$ have a result or neither has. If there is a result, let $m_\tau$'s result type be $\alpha$, and $m_\sigma$'s be $\beta$. Then $\beta < \alpha$.
     - Exception rule.
       The exceptions signaled by $m_\sigma$ are contained in the set of exceptions signaled by $m_\tau$.

- Methods' rules:

  For all $x$ of type $\sigma$:
  - Pre-condition rule.
    $$m_\tau.pre[A(x_{pre})/x_{pre}] \Rightarrow m_\sigma.pre$$
    where $x_{pre}$ is the initial state of $x$, and $m_\tau.pre$ (resp., $m_\sigma.pre$) are the pre-conditions of the method $m_\tau$ (resp., $m_\sigma$).
  - Post-condition rule.
    $$m_\sigma.post \Rightarrow m_\tau.post[A(x_{pre})/x_{pre}, A(x_{post})/x_{post}]$$
    where $x_{post}$ is the final state of $x$, and $m_\tau.post$ (resp., $m_\sigma.post$) are the post-conditions of the method $m_\tau$ (resp., $m_\sigma$).

3. Subtype constraints ensure supertype constraints.

   - Constraint Rule.
     For all $x$ of type $\sigma$, for all computation performed by $x$, and all states $\rho$ and $\psi$ of $x$ such that $\rho$ precedes $\psi$, the following rule must hold:
     $$C_\sigma \Rightarrow C_\tau[A(x_\rho)/x_\rho, A(x_\psi)/x_\psi]$$

where the notation $P[a/a']$ stands for the predicate $P$ with every occurrence of $a'$ replaced by $a$ [Thompson, 1991].

**Discussion**

In SOA domain, most of the difficulty in service substitution resides in saving the computation preformed by the unavailable service. Thus, the subtype relation should ensure that the state of the unavailable service (of supertype) is correctly used by the substitute service (of subtype) in order to ensure continuity in service provisioning without introducing inconsistencies. Hereafter, we map the subtyping relation to services in order to serve the substitution purpose. Also, the above formalization supports only service substitution, and not *runtime* service substitution. The execution progress of the unavailable service is not considered, neither is the execution resumption by the substitute service. Hence, the mapping would only enable to formalize the rules for correct service substitution. After adapting the subtyping rules, we have to consider the case of runtime replacement by defining rules that ensure the correct *runtime* substitution of services.

### 5.1.2 Mapping Subtyping Definition to Services

In SOA, we use type with respect to a service functionality. A service that belongs to a specific type has to provide the same functionality as the one the type specifies. In SOA, services may belong to multiple types as they are able to provide various functionalities. For instance, a travel agency service may provide two functionalities that instantiate different types, where one type specifies a train ticket booking functionality, and the other type, a functionality for organized sport activities. Hereafter, we map the type specification to SOA domain. We then map the requirements a subtype has to satisfy. Based on these requirements, we finally provide a mapping for subtype specification in SOA systems.

### Type specification

Let $Ser$ be all potentially existing services partitioned into typed sets (not necessarily disjoint). Each service in $Ser$ instantiates the advanced service model (presented in Section 4.3) and has a unique identity. A type defines a set of states for a service, and a single functionality ($F$) that provides the only means to manipulate the state of the service. As defined in the previous chapter, a state associates a set of variables with their related values (i.e., $< Value, Variable >$) for each existing service. $Value \in Val$, where $Val$ is the set of all possible values that a variable may have. $Variable \in Var$, where $Var$ is the set that defines all the variables the service may use.

Based on Liskov and Wing's specification of a type, we model a type for SOA systems as a triplet $T =< S, V, F >$ such that $S \subseteq Ser$, $V \subseteq Val$ and $F \in Func$, where $Func$ is the set of all functionalities that services in $Ser$ may provide. Each service defines the types to which it belongs according to the functionalities it offers. As presented in the advanced service model (Figure 4.5), a functionality is an abstract concept which is concretely described using a capability (*capa*) and implements a behavior ($b$). We formalize the concepts of capability and behavior as follows:

- A capability (*capa*) is modeled as a triplet $capa = (C_F, C_{Inputs}, C_{Outputs})$, where $C_F$ is the conceptual purpose of the functionality, $C_{Inputs}$ and $C_{Outputs}$ are respectively the semantic concepts of its required parameters, and provided results.
- A behavior ($b$) that implements the functionality is modeled using its signature, its pre- and post-conditions. In the case of composite behavior, the behavior is modeled as an aFSA and split into (1..*) atomic actions $b_i$. A behavior execution is modeled as a *sequence* of alternating states $\rho_i$ and transitions $Tr_i$ starting from an initial state $\rho_0$ such that $\rho_0\ Tr_1\ ...\ \rho_{n-1}\ Tr_n\ \rho_n$, where each transition $Tr_i$ corresponds to the execution of an atomic action $b_i$.

Figure 5.1 represents the class diagram of a type, including a functionality description that is composed of:

1. The capability description of the functionality, denoted $capa = (C_F, C_{Inputs}, C_{Outputs})$.
2. The description of the behavior of the functionality (denoted $b$), which includes:
   a) The behavior signature.
   b) The description of the pre- and post-conditions of the behavior, denoted respectively, $b.pre$ and $b.post$.
   c) In case that the behavior $b$ is composite, $b$ is split into (1..*) atomic actions $b_i$ that have atomic execution, and items (2.a) and (2.b) are retrieved for each atomic action $b_i$ composing the behavior of the functionality $F$.

Note that the concepts that define a type are retrieved from the advanced service model presented in Section 4.3.

### Type specifications need explicit invariant

The invariant defines the legal data values of the type [Liskov and Wing, 1994] (denoted as a predicate $\Phi$ over a single state) such that for any behavior execution, for any service $s$ of type $\tau$, the invariant of $\tau$ holds. For instance, an invariant property of a hotel booking functionality

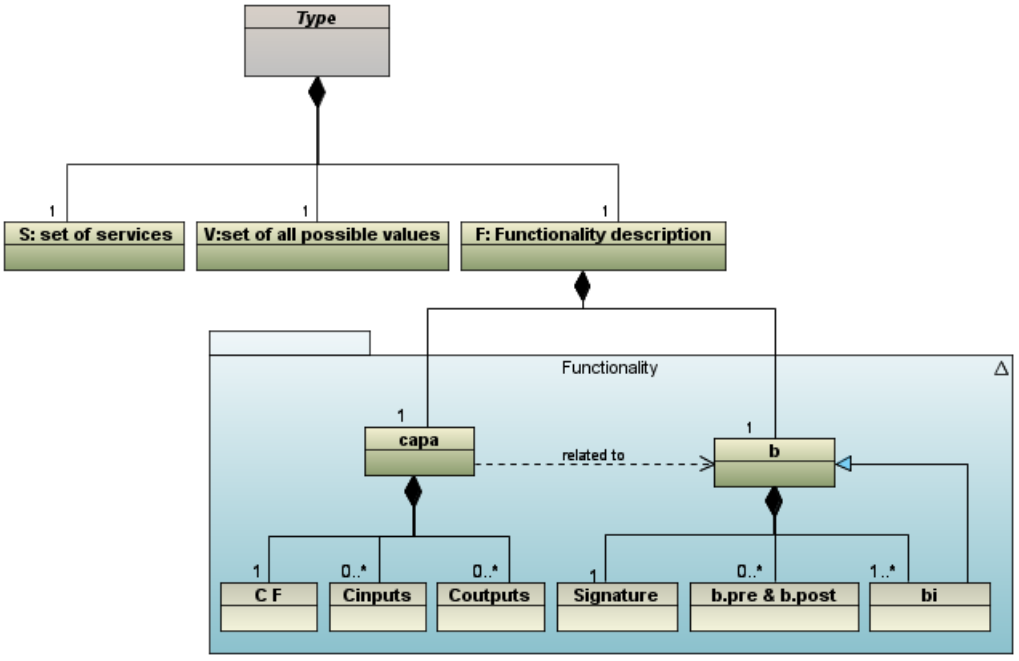



Figure 5.1: Type specification

ensures that the size of a bounded number of booked rooms never exceeds its bound, i.e., the total number of rooms of the hotel. The invariant related to the number of rooms can be expressed as follows:

$$\textbf{invariant} \quad Nb.ReservedRooms < Nb.TotalRooms$$

In this way, using the invariants, the property that preserves the behavior post-conditions can be formalized as follows. Let $I_s$ be the invariant in the type $\tau$ that preserves the post-conditions of the service behavior. For each service $s$ of a type $\tau$, having $b$ as a behavior, and $b.post$ as post-conditions of the behavior $b$, the following property must hold.

$$I_s[b.post/s_\rho]$$

where $s_\rho$ is the state of the service after $b$'s execution.

Recursively, in the case that $b$ is a composite behavior, all its atomic actions $b_i$ must preserve the invariant. For each atomic action $b_i$, the execution of which leads to the state $s_{\rho,i}$ of the service $s$ (of type $\tau$), the following property must hold.

$$I_s[b_i.post/s_{\rho,i}]$$

where $b_i.post$ are the post-conditions of the atomic action $b_i$.

### Subtype requirement

As presented in Section 5.1.1, Liskov and Wing introduced two kinds of properties for the objects: behavioral and independent.

- The *behavioral* properties ensure that the methods of the subtype preserve the behavior of the supertype ones. Mapped to services, the behavioral properties require that the substitute service should support the behavior of the unavailable service, in order to perform

seamless service substitution. However, as aforementioned, it is unlikely to find an exact replica of the service that becomes unavailable in the user environment. Hence, in our case, we do not constrain the behaviors implementing the functionalities of the unavailable and substitute services. Instead, we extend and reason on the other kind of properties, i.e., *independent* properties, as detailed in the next item.

- The *independent* properties were introduced to preserve objects consistency when more than one program share the same objects. Independent properties are of two kinds: *invariants* and *history* properties. As mentioned in the previous section, invariants must hold for all states of execution of the behavior. *History* properties are formulated as predicates over a pair of states, they must hold for a sequence of states between two specific states. The subtyping relation is then defined with respect to the history properties; this is called *constraint definition of subtyping* [Liskov and Wing, 1994]. This constraint captures exactly those history properties of a type that must be preserved by all of its subtypes. Showing that a type $\sigma$ is a subtype of $\tau$ requires showing that $\sigma$'s constraint implies $\tau$'s ones. In addition, the type $\sigma$ has to satisfy the invariants of $\tau$. Hereafter, we present the way to define the constraints for a type.

#### History constraint definition

The history properties are formulated as predicates over pairs of states [Liskov and Wing, 1994]. For instance, in the behavior of a hotel booking functionality, the history rule related to the room charges should be:

$$\textbf{constraint } \forall i : integer, \text{ such that } \; i \neq 0, \quad \text{if } s_{\rho}.cost = i \text{ then } s_{\psi}.cost \leq i$$

where $s_k.cost$ ($k \in \{\rho, \psi\}$) denotes the value of the variable *cost* denoting the room charges in the state $k$ of the service $s$, and $\rho$ is state that precedes $\psi$. The above rule means that if the hotel room is booked to a specific user at a specific cost $i$ ($i \neq 0$), in a state $\rho$ of the service $s$, then for any state $\psi$ succeeding $\rho$, the room charges should be less or equal to the first price the user has selected. This enables, in case of substitution, to comply with the price that has been proposed to the user by the service that becomes unavailable. More generally, a history constraint is a predicate that defines a constraint for a type $\tau$ over a pair of states $\rho$ and $\psi$ of $s$ (of type $\tau$), such that $\rho$ precedes $\psi$. The specification of the constraint is made over the nodes that produce the states $\rho$ and $\psi$.

#### Subtype rules

Let $\sigma =< S_\sigma, V_\sigma, F_\sigma >$ and $\tau =< S_\tau, V_\tau, F_\tau >$ be two types.

$\sigma$ is a subtype of $\tau$, denoted $\sigma < \tau$, if:

1. There exists a mapping between the capability of the supertype $capa_\tau$ and the one of the subtype $capa_\sigma$, and the subtype behavior $b_\sigma$ preserves the supertype behavior $b_\tau$ in terms of signature (i.e., covariance, contravariance).

2. The transferred state of the supertype has to preserve the invariants and constraints of the subtype.

3. The subtype behavior $b_\sigma$ preserves the rules of the supertype behavior $b_\tau$ in terms of pre- and post-conditions.

Hereafter, we detail each of the rules enumerated above, which define the subtype relation between services.

**1. Complying with supertype signature** requires a mapping between the capabilities of the supertype and the subtype, respectively, $capa_\tau$, $capa_\sigma$. In our approach, we introduce flexibility in capabilities matching, by allowing semantic inclusion between the semantic concepts of their functional purpose. In this way, the subtype functionality realizes the functional purpose of the supertype, and potentially, a richer purpose that includes the one of the supertype.

Let $capa_\tau = (C_{\tau,F}, C_{\tau,Inputs}, C_{\tau,Outputs})$ and $capa_\sigma = (C_{\sigma,F}, C_{\sigma,Inputs}, C_{\sigma,Outputs})$ be the respective capabilities of the supertype and subtype services. Mapping between the capabilities implies that:

$$C_{\tau,F} \subseteq^s C_{\sigma,F} \tag{5.1}$$

In addition, we have to map between the arguments and the results, respectively, required and provided by the services of supertype and subtype. To this aim, we have to preserve the *contravariance of arguments* and *covariance of results*, which are detailed in the following.

1. *Contravariance of arguments* preserves the number of arguments, meaning that $\sigma$'s behavior has to require the same number of arguments as $\tau$'s behavior. In our approach, we relax the contravariance condition to the relation $\leq$ (less or equal) instead of strict equality when comparing the number of arguments of $\sigma$'s and $\tau$'s behaviors, meaning that $\sigma$'s behavior has to require at most the same number of arguments as $\tau$'s behavior. Furthermore, the contravariance requires that the list of the argument types $\beta_i$ of $\sigma$'s behavior are supertypes of $\alpha_i$, of $\tau$'s ones, i.e., $\alpha_i < \beta_i$. The mapping is performed using the semantic concepts of the behavior inputs.

   Let *card* be the function that returns the cardinality of the set provided in argument. We preserve the contravariance of arguments using the two following rules:

$$\begin{array}{l} card(C_{\sigma,Inputs}) \leq card(C_{\tau,Inputs}) \\ \text{and} \\ \text{for each } i \in C_{\sigma,Inputs},\ \exists j \in C_{\tau,Inputs} \text{ such that } C_{\sigma,Inputs}(i) \subseteq^s C_{\tau,Inputs}(j) \end{array} \tag{5.2}$$

2. *Covariance of results* ensures that the number of outputs of $b_\sigma$ is equal to the number of outputs of $b_\tau$. Here also, we relax the covariance rule to the relation $\geq$ (greater or equal instead of strict equality), as providing extra results does not prevent the substitution from being correct. If the behaviors of the supertype and subtype define a set of outputs, respectively, typed $\beta_i$ and $\alpha_i$ . The covariance of results is mapped to services using semantic inclusion between the semantic concepts of the outputs. In this way, we preserve the covariance of results using the following rules:

$$\begin{array}{l} card(C_{\sigma,Outputs}) \geq card(C_{\tau,Outputs}) \\ \text{and} \\ \text{for each } i \in C_{\tau,Outputs},\ \exists j \in C_{\sigma,Outputs} \text{ such that } C_{\tau,Outputs}(i) \subseteq^s C_{\sigma,Outputs}(j) \end{array} \tag{5.3}$$

**2. Complying with subtype invariants and constraints** enables to correct synchronize the substitute service.

Let $A : V_\tau \rightarrow V_\sigma$ be the abstraction function that maps the state variables of $\tau$ (to which belong the unavailable service) to the state variables of $\sigma$ (to which belongs the substitute service). $A$ is partial, need not be onto, but can be many-to-one.

Note that $A$ in SOA is inversely defined compared to the abstraction function defined by Liskov and Wing, as in our issue, we need to map from the unavailable service variables to the substitute service's ones.

1. Subtype invariant has to be preserved by the transferred state.
   Let $\omega$ be the state of the the unavailable service, that has to be transferred to the substitute service. Let $\omega'$ be the state required by the substitute service, such that $A(\omega) = \omega'$. $s$ has to preserve the invariant of the substitute service (denoted $I_\sigma$).

$$I_\sigma[A(\omega)/\omega] \text{ must hold.} \tag{5.4}$$

2. Subtype constraints have to be preserved by the transferred state.
   Let $\phi$ be a history constraint over two states $\rho$ and $\psi$ of the substitute service, respectively produced by the execution of the nodes $n_\rho$ and $n_\psi$ that are included in the behavior of the substitute service. If the checkpoint at which the substitute service has to be synchronized is included between $n_\rho$ and $n_\psi$ then the transferred state has to preserve the constraint $\phi$. If the state that has to be transferred is $\omega$, and $A(\omega) = \omega'$, then:

$$C_\sigma[A(\omega)/\omega] \text{ must hold.} \tag{5.5}$$

**3. Complying with supertype pre- and post-conditions** amounts to satisfy the following rules, which are related to the pre-conditions and the post-conditions.

1. Pre-conditions of the supertype's behavior imply the subtype's ones. Let $s_{pre}$ be the initial state of the unavailable service, before starting $b_\tau$ execution. The behavior pre-conditions of the unavailable service (of type $\tau$) must imply those of the substitute service's behavior (of type $\sigma$).

$$b_\tau.pre[A(s_{pre})/s_{pre}] \Rightarrow b_\sigma.pre \tag{5.6}$$

2. Post-conditions of the subtype's behavior imply those of the supertype. Let $s_{post}$ be the final state of the unavailable service, after $b_\tau$ execution. The behavior post-conditions of the substitute service (of type $\sigma$) must imply those of the unavailable service's behavior (of type $\tau$).

$$b_\sigma.post \Rightarrow b_\tau.post[A(s_{pre})/s_{pre}, A(s_{post})/s_{post}] \tag{5.7}$$

### Discussion

Behaviors of the supertype and subtype may be composed of different atomic actions. Also, their respective workflows can be structured differently. This does not prevent the substitution from

being correct, but makes the service substitution more complex. The substitution mechanism has to find a mapping between the behavior of the service being substituted and the substitute service's one, in order to detect from which point the substitute service has to resume the execution.

At this stage of formalization, we have essentially focused on the semantic meaning of the substitution and the constraints under which a service functionality may serve as a substitute for another. This is necessary but not sufficient to achieve the continuity in service provisioning. Rules from 5.1 to 5.7 ensure that the substitute service is capable to provide the client with the required results, but it does ensure the ability of the substitute service to resume the execution that has been interrupted. Going one step further, we aim at saving the computation performed by the unavailable service, sparing thereby the client from restarting the interaction with the substitute service from the beginning. In the following section, we use the work introduced by C.A.R Hoare for runtime program replacement, in order to set the rules that ensure correct execution resumption by the substitute service.

### 5.1.3 Enhancing Subtyping with Dynamics: Runtime Service Substitution

Rules for formal substitution of programs have been deeply studied and proved by C.A.R Hoare's work on providing an axiomatic semantics for computer programming. "An axiomatic Basis for Computer Programming" [Hoare, 1969] introduced the now well known notation for partial correctness $P\{Q\}R$, where $P$ and $R$ are predicates specifying respectively the pre-conditions and the desired result, for the program $Q$. That is, if the assertion $P$ is true before initiation of the program $Q$, then the assertion $R$ will be true when $Q$ completes execution. This paper is based on an earlier work of Floyd [Floyd, 1967], where the technique was applied to flowcharts rather than to programs. In his theory, Hoare presented the necessary axioms and inference rules for reasoning about programs written in a simple language. This language includes assignment, sequential composition, consequence and iteration. In particular, the inference rule associated with sequential composition states that if the proven result of the first part of a program is identical with the pre-condition under which the second part of the program produces its intended result, then the whole program will produce the intended result, provided that the pre-condition of the first part is satisfied. In more formal terms:
**Rule of composition** [Hoare, 1971]

$$\text{If } P\{Q_1\}R_1 \text{ and } R1\{Q_2\}R \text{ then } P\{Q_1; Q2\}R \tag{5.8}$$

where $(Q_1; Q2)$ denotes that the programs $Q_1$ and $Q2$ are executed sequentially, starting with $Q_1$ execution. Mapping this rule to our type specification means that:

- $Q_1$ corresponds to the part of the behavior of the supertype that has been performed before service unavailability, and
- $Q_2$ corresponds to the part of the behavior of the subtype that has to be performed to achieve the required functionality.

Let $s_\tau$ be a service instance of type $\tau$, $s_\tau$ provides a functionality $F^s_\tau$, instance of the type functionality $F_\tau$. Let $b^s_\tau$ be the behavior of the functionality $F^s_\tau$, which is decomposed into $b^{s,1}_\tau$ and $b^{s,2}_\tau$ in sequence, respectively corresponding to the part of the behavior that has been executed before the service unavailability, and the non-executed part that remains to be executed in order to achieve the full execution of the behavior. We denote $b^s_\tau.pre$ and $b^s_\tau.post$, respectively the pre- and post-conditions of the behavior $b^s_\tau$. Thus, we have:

$$b^s_\tau.pre\{b^{s,1}_\tau; b^{s,2}_\tau\}b^s_\tau.post \tag{5.9}$$

Let $b_\tau^{s,1}.post$ be the post-conditions of the execution of $b_\tau^{s,1}$ such that:

$$b_\tau^s.pre\{b_\tau^{s,1}\}b_\tau^{s,1}.post \tag{5.10}$$

Let $s_\sigma$ be a service instance of type $\sigma$ subtype of $\tau$ ($\sigma < \tau$); $s_\sigma$ provides a functionality $F_\sigma^s$. In order to be able to substitute $b_\tau$ at runtime, the behavior $b_\sigma^s$ that implements the functionality $F_\sigma^s$, has to be provide a sequential decomposition into $b_\sigma^{s,1}$ and $b_\sigma^{s,2}$, denoted $b_\sigma^s = \{b_\sigma^{s,1}; b_\sigma^{s,2}\}$ such that:

$$b_\sigma^s.pre\{b_\sigma^{s,1}; b_\sigma^{s,2}\}b_\sigma^s.post \tag{5.11}$$

where $b_\sigma^s.pre$ and $b_\sigma^s.post$ are respectively the pre- and post-conditions of the behavior $b_\sigma^s$. The decomposition of $b_\sigma^s$ must be such that $b_\sigma^{s,2}$ performs the remaining part of execution to achieve $F_\tau$. We denote $b_\sigma^{s,2}.pre$ the pre-conditions of $b_\sigma^{s,2}$ such that:

$$b_\sigma^{s,2}.pre\{b_\sigma^{s,2}\}b_\sigma^s.post \tag{5.12}$$

Using Rule 5.8, the rule of substitution that has to be proved is stated as follows:

**Rule for service substitution**

If $b_\tau^s.pre\{b_\tau^{s,1}\}b_\tau^{s,1}.post$ and $b_\sigma^{s,2}.pre\{b_\sigma^{s,2}\}b_\sigma^s.post$ then, the following rule must hold

$$b_\tau^s.pre\{b_\tau^{s,1}; b_\sigma^{s,2}\}b_\sigma^s.post \tag{5.13}$$

Rule 5.13 is relaxed using Hoare's rules of consequences, which state that if the execution of a program $Q$ ensures the truth of the assertion $R$, then it also ensures the truth of every assertion logically implied by $R$. Also, if $P$ is known to be a pre-condition for a program $Q$ to produce the result $R$, then any other assertion which logically implies $P$ can be a pre-condition for the program $Q$. These rules are expressed formally as follows:
**Rules of consequence** [Hoare, 1971]

$$\text{If } P\{Q\}R \text{ and } (R \Rightarrow S) \text{ then } P\{Q\}S \tag{5.14}$$

$$\text{If } P\{Q\}R \text{ and } (S \Rightarrow P) \text{ then } S\{Q\}R \tag{5.15}$$

Furthermore, as $\sigma$ is a subtype of $\tau$, Rules 5.6 and 5.7 imply

$$b_\tau^s.pre \Rightarrow b_\sigma^s.pre \tag{5.16}$$

$$b_\sigma^s.post \Rightarrow b_\tau^s.post \tag{5.17}$$

Rule 5.9 defines a correct execution of the behavior when no service unavailability occurs. Starting from the Rule 5.9, we follow a deductive reasoning to prove that Rule 5.13 holds under specific constraints. Using Rules 5.11, 5.14 and 5.16, we have:

$$b_\sigma^s.pre\{b_\sigma^{s,1}; b_\sigma^{s,2}\}b_\tau^s.post \tag{5.18}$$

Using Rules 5.15, 5.17 and 5.18, we have:

$$b_\tau^s.pre\{b_\sigma^{s,1}; b_\sigma^{s,2}\}b_\tau^s.post \tag{5.19}$$

Hence, the issue is reduced to replace $b_\sigma^{s,1}$ by $b_\tau^{s,1}$ in Rule 5.19.

**Assumption**

$$b_{\tau}^{s,1}.post \Rightarrow b_{\sigma}^{s,2}.pre \tag{5.20}$$

Assume that the constraint 5.20 holds. Thus, using the rule of consequence 5.14 and Rule 5.10, the constraint 5.20 implies:

$$b_{\tau}^{s}.pre\{b_{\tau}^{s,1}\}b_{\sigma}^{s,2}.pre \tag{5.21}$$

Integrating Rule 5.21 in the rule of composition 5.8, we have

$$b_{\tau}^{s}.pre\{b_{\tau}^{s,1}\}b_{\sigma}^{s,2}.pre \text{ and } b_{\sigma}^{s,2}.pre\{b_{\sigma}^{s,2}\}b_{\sigma}^{s}.post \text{ then } b_{\tau}^{s}.pre\{b_{\tau}^{s,1}; b_{\sigma}^{s,2}\}b_{\sigma}^{s}.post \tag{5.22}$$

Hence, Rule 5.22 makes the replacement of $b_{\sigma}^{s,1}$ by $b_{\tau}^{s,1}$ in Rule 5.19 correct. Consequently, the combination of Rules 5.10 and 5.22 makes the rule for service substitution 5.13 satisfied. Therefore, a new constraint (Rule 5.20) is added in the subtyping relation between two services, which takes into account the runtime progress of the execution. The constraint consists in finding a decomposition in the behavior of the subtype $b_{\sigma}^{s}$, the pre-conditions of which are deduced from the post-conditions of the last atomic action of $b_{\tau}^{s,1}$ of the supertype's behavior $b_{\tau}^{s}$, which has been executed before $s_{\tau}$ unavailability.

In the following section, we detail the matching between the behaviors of the unavailable service and the substitute one, which complies with the rules for runtime service substitution.

## 5.2 Execution Resumption by the Substitute Service

The main constraint for substitution (Rule 5.20) is to find an atomic action $b_{\sigma}^{s,j}$ in the subtype's behavior for which the pre-conditions are implied by the post-conditions $b_{\tau}^{s,i}.post$, where $b_{\tau}^{s,i}$ is the last atomic action executed of the type's behavior $b_{\tau}^{s}$.

Based on the previous section, three main requirements regarding the workflow structure emerge. First is to use the behavior composition in terms of atomic actions in order to decompose the behavior of the unavailable service into a sequence of two parts: one part of the behavior that has been executed, and another part that has to be executed. The point of split should coincide with the end of the last completed atomic action (Section 5.2.1). Second is to find an atomic action in the workflow of the substitute service, the pre-conditions of which are implied by the post-conditions of the last atomic action performed by the unavailable service (Section 5.2.2). Third is to sequentially decompose the substitute service behavior, and select the point at which the execution of the behavior of the substitute service can be resumed (Section 5.2.3).

### 5.2.1 Sequential Decomposition of the Unavailable Service Behavior

As presented in Chapter 4, we model a composite behavior using an aFSA. Let $Node$ be the function that tracks at each time $t$, the executing node $n$ in the aFSA with respect to the progress of the behavior execution. The function $Node$ may return more than one node, e.g., when executing an AND-split structure of the workflow. If the service unavailability occurs at the time $t$, then the simplest way to decompose the workflow into two parts is to consider all the nodes that have been executed in the time interval $[0..t]$ ($n$ excluded) as the first part of the workflow, and all the other nodes that remain to be executed, as the second part of the workflow. However, the problem is far more complex. Indeed, many issues have to be considered, including:

- $n$ may not be the initial node of an atomic action. The last *state transfer* checkpoint may have been performed several nodes before the node $n$. Therefore, at the client side, a rollback to the last *state transfer* checkpoint has to be considered.

- The rollback may be extended in the case that the function *Node* returns more than one node. Hence, the rollback may be performed on multiple *state transfer* checkpoints.
- These *state transfer* checkpoints may be positioned on different branches of the AND-split structure, which requires reasoning on a consistent split with respect to the workflow structure.

To deal with the above issues, we consider the workflow patterns presented in Section 4.1 in order to identify the ones that may induce a conflict when decomposing the workflow. In particular, the issue of decomposition turns to be complex when managing multiple branches in the workflow. This is detailed hereafter, considering each pattern independently.

### Conflicts in workflow decomposition and their related strategies

The case of the sequence pattern is the simplest case. As represented in Figure 5.2, considering the case that the execution interruption occurs when executing a sequence of nodes, a first step consists in rolling back to the last *state transfer* checkpoint that has been performed before the execution interruption, i.e., which corresponds to the end of an atomic action execution. Then, the decomposition is easy: the first part of the behavior (i.e., $b_\tau^{s,1}$) is delimited by the the

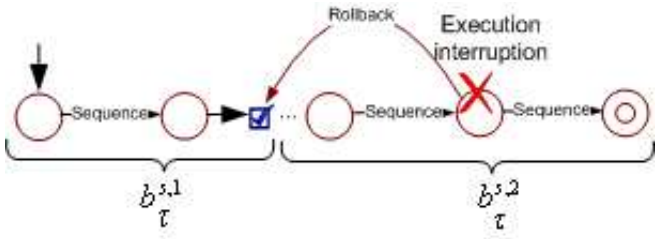



Figure 5.2: Sequence conflict

initial node of the aFSA and the last *state transfer* checkpoint performed before the execution interruption, the second part of the behavior (i.e., $b_\tau^{s,2}$) includes the nodes succeeding that *state transfer* checkpoint till the final node.

The case of the AND-split pattern is more complex. As represented in Figure 5.3, considering the case that the execution interruption occurs when executing multiple branches of an AND-split structure, the main issue concerns the rollback to the last *state transfer* checkpoint: there may be multiple, independently performed *state transfer* checkpoints. In such case:

- Shall we rollback each branch independently of the others? And then, what would be the post-conditions of the execution performed by the unavailable service? This case is illustrated in Figure 5.3 by two rollbacks annotated with ❶.
- Or, shall we rollback to the last *state transfer* checkpoint preceding the AND-split pattern? This case is illustrated in Figure 5.3 by one rollback annotated with ❷.

The first case saves more computation than the second one. However, the consistency of the first decomposition is not guaranteed. Indeed, the set of checkpoints on different branches has to form a consistent and complete state of the service. In the absence of any data dependency between the nodes on different branches, this can be considered. Nevertheless, in a more general

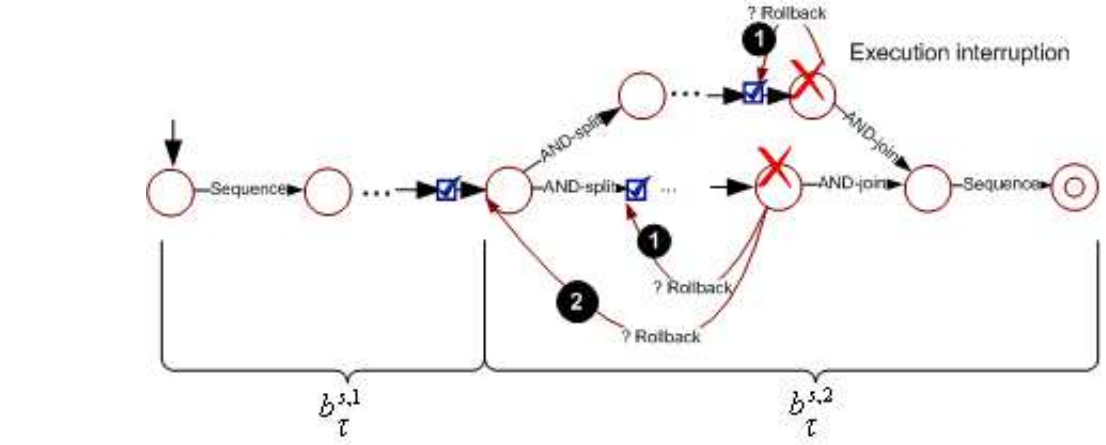



Figure 5.3: AND-split conflict

case, we cannot restrict dependency between parallel branches. Hence, even though the second case wastes more computation than the first one, it serves better our need for sequential split of the behavior. In this way, the behavior of the unavailable service is split into two parts: One part starting from the initial node of the workflow, till reaching the last *state transfer* checkpoint that precedes the AND-split pattern (corresponding to $b_\tau^{s,1}$), and the other part starts from the beginning of the AND-split, till reaching the final node (corresponding to $b_\tau^{s,2}$).

The case of the AND-join pattern is considered when the execution is interrupted at the node where the AND-join is performed. We consider AND-join as a specific case of AND-split, as the last checkpoint performed is distributed on serveral branches, while the union of the last checkpoints performed may not form a consistent and complete state of the service.

As for the AND-split structure, we restart from the beginning of the last *state transfer* checkpoint performed before the AND-split structure, ensuring thereby a sequential decomposition of the workflow.

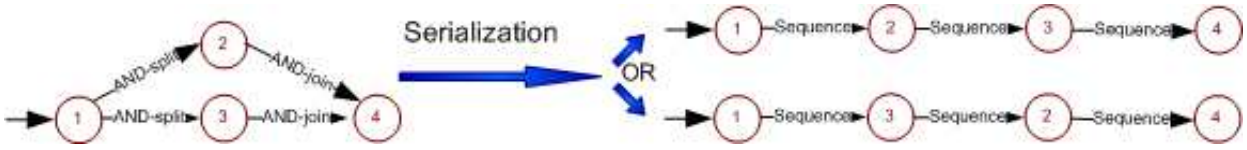



Figure 5.4: Flow serialization

Nevertheless, an optimization of the extent of the rollback can be realized by restructuring the workflow of the unavailable service. Research efforts [Flé and Roucairol, 1985] have been proposed to sequenctially serialize concurrent computation. More recently, in [König et al., 2008], the authors focus on reasonning on BPEL processes compatibility. They propose to serialize the nodes included between an AND-split and an AND-join structures (which they call a parallel flow), applying a set of rules on activities that have to be concurrently executed in a parallel flow. These activities can be executed in any sequential order without having impact on the process results. Obviously this can be performed only in case of absence of any data dependencies between the activities. Mapping the serialization to our need for sequential decomposition, a parallel flow execution can be restructured into the execution of sequences of nodes, such as in the example

of Figure 5.4. Considering the example of Figure 5.4, the nodes 2 and 3 can be modeled, after serialization, in sequence $(1-2-3-4)$ or $(1-3-2-4)$ according to their execution. However, this serialization does not comply with our modeling of composite behaviors as a workflow of atomic actions. Indeed, restructuring the workflow may mix the nodes of multiple atomic actions together, which makes the matching between pre- and post-conditions hardly achievable.

Regarding the other workflow patterns (i.e., XOR-split and Merge), they can be considered as a specific case of the sequence structure as one single branch is chosen and executed.

As a result of this decomposition, we find the *state transfer* checkpoint, at which the previously stored state of the unavailable service has to be transferred to the substitute service.

### 5.2.2 Matching between the Behaviors of the Unavailable and Substitute Services

The issue consists in finding a *state transfer* checkpoint in the workflow of the substitute service, at which the substitute service can synchronize its state with the transferred state of the unavailable service. We also aim to ensure that the execution of substitute service's behavior would satisfy the user required results. However, the behavior of the substitute service may include a set of final nodes, where only a subset of these nodes actually satisfies the user requested results. Hence, we consider only this subset of final nodes, then we use backward chaining, originated from artificial intelligence (AI) planning techniques [Hendler et al., 1990, Yang, 1997], in order to find the checkpoint at which of the substitute service can be performed.

#### Formalization

In Chapter 2, we proposed to represent the user requested capability using the same schema as service capabilities. Let *RequiredCapa* be the user requested capability such that

$$RequiredCapa = (C_{RequiredFunc}, C_{ProvidedInputs}, C_{RequiredResults})$$

where $C_{RequiredFunc}$ denotes the semantic concept of the required functionality, $C_{ProvidedInputs}$ denotes the set of the semantic concepts of the user inputs, and $C_{RequiredResults}$ denotes the set of the semantic concepts of the expected results.

Let $b^s_\tau = (Q_\tau, \sigma, \delta, s_0, F_\tau, QA, STCkpt_\tau, RCkpt_\tau)$ be the behavior that the unavailable service has been executing before the execution interruption. Let $ckpt_\tau \in STCkpt_\tau$ be the *state transfer* checkpoint resulting from $b_\tau$'s sequential decomposition. Let $s_\sigma$ be a candidate service for substitution that complies with the supertype signature and pre- and post-conditions rules. $s_\sigma$ provides a functionality $F^s_\sigma$ implemented through the behavior $b^s_\sigma = (Q_\sigma, \sigma', \delta', s'_0, F_\sigma, QA', STCkpt_\sigma, RCkpt_\sigma)$. Matching between $b^s_\tau$ and $b^s_\sigma$ is twofold.

1. First, we have to find in $b^s_\sigma$ a final node (denoted $N_{F,\sigma}$) that provides as outputs the user requested results. The existence of such a node is guaranteed when complying with the covariance of results between the substitute and unavailable service. In the case of the existence of multiple final nodes that satisfy the user needs, we iterate our matching algorithm for each of them, till finding a *state transfer* checkpoint ($ckpt_\sigma$) in the workflow of the substitute service ($b^s_\sigma$) that is on the backward path of $N_{F,\sigma}$.

2. Second, for each *state transfer* checkpoint $ckpt_\sigma$ in $b^s_\sigma$, in the backward path of final node $N_{F,\sigma}$, such that:

   - $ckpt_\sigma$ should be compatible with $ckpt_\tau$, i.e, $ckpt_\sigma$ should be provided with (1) a description of the workflow state $SD_{wkf}(ckpt_\sigma)$ such that $SD_{wkf}(ckpt_\sigma) \subseteq^s SD_{wkf}(ckpt_\tau)$ in order to comply with the abstraction function $A$ required in the subtyping rule.

- The transferred state should comply with $b_\sigma^s$'s invariants (in the case that $b_\sigma^s$ defines invariants).
- The transferred state should comply with the constraints of $b_\sigma^s$, in the case that there exists a constraint on two nodes of $b_\sigma^s$, such that $ckpt_\sigma$ is included between these two nodes.
- The post-condition of the atomic action that $ckpt_\tau$ completes should imply the pre-conditions of the atomic action initiated by $ckpt_\sigma$.

In the case that one of the above conditions are not satisfied, we iterate the verification of the rules with the *state transfer* checkpoint that precedes $ckpt_\sigma$. In the case that we reach the initial node of $b_\sigma^s$, we process another final node in the list of final nodes that satisfy the user request.

In the case that the structure of the behavior $b_\sigma^s$ of the substitute service $s_\sigma$ is not sequential, we detail hereafter the strategies that enable to select the point in $b_\sigma^s$ at which the execution should be resumed.

### 5.2.3 Sequential Decomposition of the Substitute Service Behavior

Where the execution should be resumed in the substitute service behavior? In this section, we mainly focus on this issue. Indeed, in the previous section, we presented how to find in the behavior $b_\sigma^s$ an atomic action $b_\sigma^{s,j}$ (initiated by $ckpt_\sigma$), from which the execution can be resumed. An issue emerges in the case that $b_\sigma^{s,j}$ is included, e.g., in one of the branches of an AND-split structure. Hereafter, we consider each workflow structure independently in order to distinguish the cases that may cause a conflict.

In the case that $b_\sigma^{s,j}$ is included in a sequence structure, the problem is greatly simplified. After synchronization of the state of $b_\sigma^s$ with the *state transfer* checkpoint issued from the $b_\tau^s$ decomposition, we start executing $b_\sigma^{s,j}$. After its completion the following atomic action (i.e., the successor of $b_\sigma^{s,j}$) is executed, and so forth.

In the case that $b_\sigma^{s,j}$ is included in a branch of a AND-split structure, the issue becomes more complex. In such cases, for each other branch included in the same parallel split pattern, we have to find the *state transfer* checkpoint at which the execution should be resumed, i.e., we have to find the atomic actions for which the pre-conditions are implied by $b_\tau^{s,1}.post$. Still, the set of resuting checkpoints does not guarantee the consistency of the service state, as nodes may present data dependencies. Hence , to avoid inconsistency risks, we find out the *state transfer* checkpoint that initiates the atomic action preceding the parallel split workflow pattern. If the resulting *state transfer* checkpoint is included between two nodes on which the substitute service defines a constraint, then the transferred state should comply with the constraint rule of the substitute service. Otherwise, we should find a *state transfer* checkpoint that precedes the one that has been selected and retry to decompose the workflow.

In the case that $b_\sigma^{s,j}$ starts at a synchronization node (AND-join), there is no conflict that may emerge, as long as the pre-conditions of $b_\sigma^{s,j}$ are satisfied. Indeed, the pre-conditions include implicitly that all the converging branches of the AND-join structure are executed and their results are relevant to be used in the rest of the workflow. The cases of the exclusive choice (XOR-split) and simple Merge are considered as a particular case of the sequence structure. Indeed, we only consider the branch that includes $b_\sigma^{s,j}$.

## 5.3 Algorithm for Runtime Service Substitution

In this chapter, we mapped the definition of the subtyping relation from OOD to SOA systems. In Section 5.1, we established seven rules that are issued from the mapping of subtyping of OOD to SOA systems (Rules 5.1 to 5.7), and another rule that is deduced from Hoare's logic (Rule 5.13). We organize these rules into four groups:

1. Rules for complying with supertype signature (Rules from 5.1 to 5.3).
2. Rules for complying with subtype invariants and constraints (Rules 5.4 and 5.5).
3. Rules for complying with supertype pre-conditions and post-conditions (Rules 5.6 and 5.7).
4. Rule for runtime execution resumption (Rule 5.13).

Then, in Section 5.2, we presented the strategies that enable to:

1. sequentially decompose the workflow of the unavailable service in order to find the checkpoint at which the state of the unavailable service should be transferred.
2. match between the workflow of the unavailable service and the substitute service's one, in order to find the *state transfer* checkpoint that complies with the rule for runtime substitution.
3. sequentially decompose the workflow of the substitute service, according to the checkpoint resulting from the previous step and the workflow structure in which it is included. As a result of this step, we have a *state transfer* checkpoint, at which the substitute service should synchronize its state according to the transferred state.

Before proceeding to state transfer, the state that has to be transferred should comply with the invariant, and potentially, the constraints of the substitute service. The compliance with the constraints is checked only when the state transfer checkpoint of the substitute service is included between two nodes on which a constraint is defined. The high level steps of runtime service substitution are summarized in Algorithm 2.

In Chapter 6, we propose the methods for checking the compliance of Web services with the rules for runtime service substitution. We also present a way for classifying services in the presence of multiple candidates for substitution.

**Algorithm 2**: High level instructions for runtime service substitution

**Data**: Service descriptions of the unavailable service and a candidate for substitution.
**Result**: A *state transfer* checkpoint ($ckpt_\sigma$) at which the candidate service should resume its execution.
**begin**
  Check the compliance with the rules for supertype signature.
  **if** *not* **then**
    Return $ckpt_\sigma = \oslash$.
  Check the compliance with supertype pre-conditions and post-conditions.
  **if** *not* **then**
    Return $ckpt_\sigma = \oslash$.
  Starting from the node at which the service unavailability occurs:
  Find in the backward path the preceding *state transfer* checkpoint using sequential decomposition of the unavailable service's workflow.
  **if** *such checkpoint exists* **then**
    Find a final node $N_{F,\sigma}$ in the behavior of the substitute service that satisfies the user requested results.
13  Find in the backward path the *state transfer* checkpoint in the workflow of the candidate service that complies with the rule for runtime substitution.
15  From previous step-resulting checkpoint, sequentially decompose the candidate service's workflow, in order to find the checkpoint ($ckpt_\sigma$) from which the candidate service should resume its execution.
    Check whether the checkpoint $ckpt_\sigma$ is located between two nodes on which the candidate service defines a constraint.
    **if** *it is the case* **then**
      Check whether the state that has to be transferred complies with the constraint.
      **if** *not* **then**
        /* We repeat the processing with the predecessor of $ckpt_\sigma$ */
        $ckpt_\sigma$ = predessor of $ckpt_\sigma$
        Repeat from Step 13.
    Check whether the state that has to be transferred complies with the invariant of the candidate service.
    **if** *not* **then**
      /* We repeat the processing with the predecessor of $ckpt_\sigma$ */
      $ckpt_\sigma$ = predessor of $ckpt_\sigma$
      Repeat from Step 13.
    **else**
      Return the checkpoint ($ckpt_\sigma$) resulting from Step 15.
  **else**
    Return $empty\ set$.
**end**

*Science is a collection of successful recipes.*

Paul, Valery

# 6

# Compatibility Check and Semantic-based Service Classification

The focus of this chapter concerns checking the compatibility between the functionality of unavailable service and the substitute service's one, that is the ability of a service to be a substitute for the now unavailable service. The compatibility with the unavailable service is determined and measured on the basis of the advanced service model and the runtime service substitution rules presented respectively in Chapters 4 and 5.

To evaluate the compatibility, we define a compatibility degree between the unavailable service functionality and the one of the candidate service for substitution. In the case of multiple candidate services for substitution, comparing the compatibility degree of the candidates enables to classify them into catalogs, and select the substitute service that matches the best with the unavailable service. The more the substitute service saves the computation performed by the unavailable service, the better is the compatibility. The key idea behind our approach for service substitution is to find all the candidates that may substitute the unavailable service, and then to refine the selection till finding the substitute service that matches the best with the unavailable service.

The rest of this chapter is organized as follows. In Section 6.1, we define the compatibility degree over service functionalities, according to the compliance with the four groups of rules for runtime substitution, established in the previous chapter. In Section 6.2, we check the compatibility with respect to the rules related to the supertype signature (i.e., unavailable service). In Section 6.3, we check the compatibility with respect to the supertype pre- and post-conditions. In Section 6.4, we check the compatibility with respect to the rule for runtime execution resumption. In Section 6.5, we check the compatibility with respect to the invariants and constraints of the candidate service. Once the compatibility degree is evaluated, we present in Section 6.6 the decision graph that enables to select the service that will act as an actual substitute service for the unavailable service. Finally, Section 6.7 wraps up this chapter by presenting our concluding remarks.

## 6.1 Compatibility Degree

In this section, we define the compatibility degree between the functionalities provided by the unavailable service and its substitute. Based on the four groups of rules established in the previous chapter, the compatibility over services has to be satisfied in four aspects:

- *Signature compatibility* includes matching between the capability of the unavailable service and the one of the substitute service, and if required, the syntactic mapping of the in/out

parameters according to the signature of the substitute service behavior.

- *Supertype pre- and post-conditions compatibility* includes matching between the predicates used in defining pre- and post-conditions of the unavailable service and the ones of the substitute service. It also includes ensuring that the pre-conditions of the unavailable service imply those of the substitute service, and inversely for the post-conditions.
- *Compatibility with respect to the runtime execution resumption* includes (1) the structural compatibility between the behavior of the unavailable service and the substitute service's one, in order to find a checkpoint in the behavior of the substitute service from which the execution can be resumed. Also, (2) the state of the unavailable service has to be compatible with the one required by the substitute service, i.e, the transferred state should provide all the data required by the substitute service in order to synchronize its state accordingly.
- *Invariants and constraints compatibility* includes checking whether the state issued from the state compatibility complies with the invariants and constraints of the substitute service.

Hereafter, we define the compatibility degree between two functionalities, taking into account the compatibility over the above aspects, in order to determine whether a candidates service is able to substitute the unavailable service, or not.

**Definition 5. Compatibility degree ($CD$)** between the unavailable service's functionality $F^s_\tau$ and a candidate service's functionality $F^s_\sigma$ includes four aspects namely: (1) the compatibility degree over their signatures ($CD_{Signature}$), (2) the compatibility degree over their pre- and post-conditions ($CD_{Pre-Post}$), (3) their structural/state compatibility for runtime substitution ($CD_{RuntimeExeRes}$), and (4) the compatibility with respect to the invariants and constraints of the substitute service ($CD_{Inv-Const}$). The greater the compatibility degree is, the greater are the chances to achieve a successful substitution.

Based on the above definition, we establish an equation that determines the compatibility degree between an unavailable functionality $F^s_\tau$ and a candidate functionality for substitution $F^s_\sigma$ provided respectively by the service $s_\tau$ (of type $\tau$) and $s_\sigma$ (of type $\sigma$).

$$CD(F^s_\tau, F^s_\sigma) = \begin{cases} \sum_i \omega_i \cdot CD_i & \text{if } CD_{Signature} \cdot CD_{Pre-Post} \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

where $i \in \{Signature, Pre-Post, RuntimeExeRes, Inv-Const\}$ and, $\omega_i$ is the weight related to $CD_i$.

The compatibility degree is the product of the weighted compatibility degrees related to each of the aspects that have impact of the compatibility between functionalities. If the product $CD_{Signature} \cdot CD_{Pre-Post}$ is null, it invalidates the compatibility between two functionalities, as it means that the signature or the pre- and post-conditions of the unavailable service are not followed by the substitute service; the compatibility degree is then null. However, in the case that the state transfer is not possible due to state or workflow structure incompatibilities, or due to the non compliance with the invariants and constraints, the service can still serve the substitution. In the absence of any other alternative to save the computation previously performed, we select a substitute candidate with which the state transfer is not possible, and restart the interaction from the beginning with the selected substitute service.

The weights enable to emphasize the importance that we give to a specific aspect with respect to the others. In case that no preference is provided, the weights can be removed, making all the aspects equally important.

## 6.2 Complying with Supertype Signature

In this section, we focus on mapping between the supertype and subtype signatures. Section 6.2.1 checks the compatibility over the semantic descriptions of the capabilities, and Section 6.2.2 checks the compatibility with respect to the required syntactic mappings between the signature of the unavailable service and the substitute service's one.

### 6.2.1 Signatures Semantic Matching

As presented in Chapter 2, we support 4 semantic relationships when matching between capabilities, namely instance-concept, hierarchical, compositional and equivalence relationships. To establish an order for classifying the services, several efforts [Zhong et al., 2002, Hau et al., 2005, Ben Mokhtar, 2007] have proposed to compute the semantic distance between two semantic concepts to which they make reference. In our approach, we do not restrict the algorithm for computing the semantic distance between semantic concepts. We only need that the semantic distance between two semantic concepts $C_\tau$ and $C_\sigma$ (denoted $SemanticDist$) should be normalized, according to their semantic relationship, as follows:

$$SemanticDist(C_\tau, C_\sigma) = \begin{cases} \alpha \mid 1 \leq \alpha & \text{if } C_\tau \xrightarrow{Comp} C_\sigma \\ \beta \mid \alpha < \beta & \text{if } C_\tau \xrightarrow{SuperClass} C_\sigma \\ \chi \mid \beta < \chi & \text{if } C_\tau = C_\sigma \text{ or } C_\tau \simeq C_\sigma \text{ or } \exists C | C_\tau, C_\sigma \xrightarrow{I-C} C \\ 0 & \text{if none of the above} \end{cases}$$

To compute the compatibility degree $CD_{Signature}$ over signatures, we first compute the compatibility degree over the capabilities of the unavailable and substitute services, denoted $CapabilityMatching$.

Let $F^s_\tau$ be the functionality provided by the unavailable service, which is described using the capability $capa^s_\tau = (C_{\tau,F}, C_{\tau,Inputs}, C_{\tau,Outputs})$, and $F^s_\sigma$ the functionality provided by a candidate service for substitution, which is described using the capability and $Capa^s_\sigma = (C_{\sigma,F}, C_{\sigma,Inputs}, C_{\sigma,Outputs})$. The compatibility degree has to ensure the compliance with the contravariance and covariance between the in/out parameters of the two functionalities, in addition the semantic inclusion between the concepts that represent their functional purposes. Meaning that

1. $C_{\sigma,F} \subseteq^s C_{\tau,F}$
2. the cardinality of $C_{\sigma,Inputs}$ (denoted $n$) has to be less than, or equal to, the cardinality of $C_{\tau,Inputs}$,
3. the cardinality of $C_{\sigma,Outputs}$ (denoted $m$) has to be greater than, or equal to, the cardinality of $C_{\tau,Outputs}$,
4. for each $i \in C_{\sigma,Inputs}$, $\exists j \in C_{\tau,Inputs}$ such that $C_{\sigma,Inputs}(i) \subseteq^s C_{\tau,Inputs}(j)$, in order to satisfy the covariance rule, and
5. for each $i \in C_{\tau,Outputs}$, $\exists j \in C_{\sigma,Outputs}$ such that $C_{\tau,Outputs}(i) \subseteq^s C_{\sigma,Outputs}(j)$, in order to satisfy the contravariance rule.

If one of the above conditions is not satisfied the $CapabilityMatching(F^s_\tau, F^s_\sigma)$ is then null, otherwise, the compatibility degree $CapabilityMatching$ is computed as follows:

$$CapabilityMatching(capa^s_\tau, capa^s_\sigma) = \begin{cases} SemanticDist(C_{\tau,F}, C_{\sigma,F}) + \\ \sum_{i,j=0}^{n} SemanticDist(C_{\tau,Inputs}(i), C_{\sigma,Inputs}(j)) + \\ \sum_{i,j=0}^{m} SemanticDist(C_{\sigma,Outputs}(j), C_{\tau,Outputs}(i)) \end{cases}$$

This enables only to match between the semantic descriptions of the unavailable service and a candidate service for substitution. A syntactic mapping is also required to enable the actual use of the substitute service in place of the unavailable one.

### 6.2.2 Signatures Syntactic Mapping

Once the semantic matching between capabilities is established, we concentrate on syntactic mapping of the input parameters in order to correctly perform the mapping between the unavailable service signature and the one of the substitute service. In Chapter 2, we presented how SAWSDL enables syntactic data mapping using lifting and lowering mechanisms, when two XML elements are annotated with the *same* semantic concept. An issue emerges when the XML elements are annotated with different semantic concepts. For example, they are annotated respectively with $C_\sigma$ and $C_\tau$, where $C_\sigma$ semantically includes $C_\tau$, or the inverse. Hereafter, we follow an iterative reasoning to deal with this issue. The idea is to find a set of semantic concepts that are in

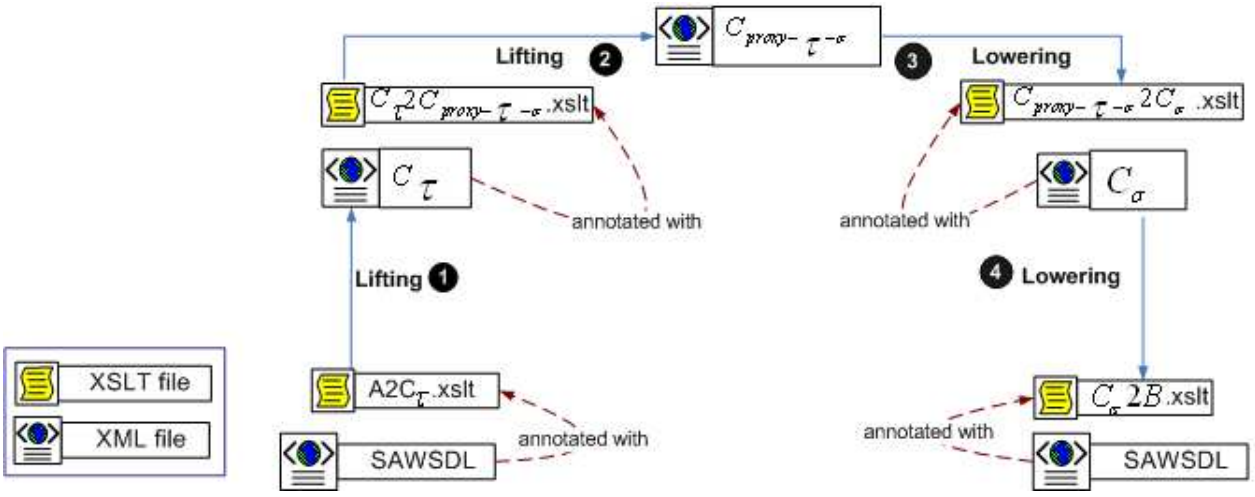



Figure 6.1: Recursive mapping between semantic concepts

relation with $C_\tau$ and $C_\sigma$, and that serve as a "bridge" in order to link $C_\tau$ to $C_\sigma$. These semantic concepts needs to be provided with XSL transformations that enable transforming an instance of XML schema representing $C_\tau$ into an instance of XML schema that represents $C_\sigma$. Hereafter, we present an example of how the mapping should be performed in a single semantic concept that puts $C_\tau$ and $C_\sigma$ in relationship, called "proxy".

#### Illustrating example

As presented in Section 2.4.3, the representation of OWL semantic concepts can be mapped to XML schemas. Hence, lifting and lowering mechanisms can be also applied to semantic concepts using their XML representation. In this case, we introduce the notion of "*proxy*" semantic concept

that enables to make the mapping between the XML representation of semantic concepts. The mapping can be fulfilled in four steps:

- First, we consider that the XML element $A$ makes reference to the semantic concept $C_\tau$, and $B$ to the semantic concept $C_\sigma$. The XML representation of $A$ can be transformed into an instance of $C_\tau$'s representation, using lifting mechanism. The lifting is enabled through the XSL transformation, denoted $A2C_\tau$ in Figure 6.1. This step is annotated with ❶ in Figure 6.1.
- In the second step (Step ❷ in Figure 6.1), the result of the first step is transformed into an instance of the XML representation of the concept $C_{proxy-\tau-\sigma}$, using lifting mechanism. Here, also, the lifting is enabled through the XSL transformation, denoted $C_\tau 2C_{proxy-\tau-\sigma}$ in Figure 6.1.
- In the third step (Step ❸ in Figure 6.1), the result of the previous step is then transformed into an instance of the XML representation of $C_\sigma$ using lowering mechanism. The lowering is enabled through an XSL transformation, denoted $C_{proxy-\tau-\sigma}2C_\sigma$ in Figure 6.1.
- The fourth and final step (Step ❹ in Figure 6.1) transforms the instance of $C_\sigma$ to the XML element $B$, using a lowering technique. As in previous steps, the lowering is enabled using an XSL transformation, denoted $C_\sigma 2B$ in Figure 6.1.

The syntactic mapping cannot be performed without the definition of the "proxy" semantic concepts (e.g., $C_\tau 2C_{proxy-\tau-\sigma}$), as well as the provision of XSL transformations. In the case that $C_\tau 2C_{proxy-\tau-\sigma}$ or the XSL transformations are not available, the degree of compatibility $CD_{Signature}$ is null.

To check the feasibility of syntactic mapping, we introduce a new factor in computing the compatibility degree over signatures, which we denote $SyntacticMapping$. $SyntacticMapping$ ensures that the syntactic mapping between the *in/out* parameters can be performed, and all the required XSL transformations are beforehand available.

Let $b^s_\tau$ and $b^s_\sigma$ be respectively the behaviors of the unavailable and the substitute service. Let $I^{b^s_\tau}$ and $O^{b^s_\tau}$ be respectively the required inputs and provided outputs of $b^s_\tau$, and $I^{b^s_\sigma}$ and $O^{b^s_\sigma}$, $b^s_\sigma$'s ones.

For each $i_\sigma \in I^{b^s_\sigma}$, the syntactic mapping has to transform the corresponding input $i_\tau \in I^{b^s_\tau}$ into an instance of $i_\sigma$. The existence of $i_\tau$ is ensured through the semantic matching of the capabilities, which we evaluated in the previous section. Also, for each $o_\tau \in 0^{b^s_\tau}$, the syntactic mapping has to transform output $o_\tau \in O^{b^s_\tau}$ into an instance of $o_\sigma$, which matches semantically with $o_\tau$. The syntactic mapping is evaluated as follows

$$SyntacticMapping(b^s_\tau, b^s_\sigma) = \begin{cases} 1 & \text{if } \forall i_\sigma \in I^{b^s_\sigma}, \exists i_\tau \in I^{b^s_\tau}, \text{such that} \\ & \text{the mapping from } i_\tau \text{ to } i_\sigma \text{ is possible, and} \\ & \forall o_\tau \in I^{b^s_\tau}, \exists o_\sigma \in O^{b^s_\sigma}, \text{such that} \\ & \text{the mapping from } o_\tau \text{ to } o_\sigma \text{ is possible.} \\ 0 & \text{otherwise.} \end{cases}$$

Note that the syntactic mapping is defined only with respect to the behaviors inputs. Indeed, the purpose behind the syntactic mapping is to ensure that the data transferred from the unavailable service to the substitute one can be transformed in the form required by the substitute service. Thus, there is no need to transform the outputs of the behavior of the unavailable service.

The compatibility degree $CD_{Signature}$ between the functionality of the unavailable service and the one of the candidate service is defined as the product of the compatibility degree over their semantic descriptions (i.e., their respective capabilities) and the one over their syntactic descriptions (i.e., their behaviors' signatures).

$$CD_{Signature}(F^s_\tau, F^s_\sigma) = CapabilityMatching(capa^s_\tau, capa^s_\sigma) \cdot SyntacticMapping(b^s_\tau, b^s_\sigma)$$

In the case that the compatibility degree is the same for two candidate services, then we take into account the required number of syntactic mappings for each candidate. As the number of mappings have impact on the recovery time when substituting a service with another one, we target to reduce it as possible.

### Number of syntactic mappings

In the above mapping, we considered the case of semantic inclusion between $C_\tau$ and $C_\sigma$, instead of strict identity. Then, we present that the syntactic mapping is performed in 4 steps. However, as the semantic inclusion is transitive, the relationship between $C_\tau$ and $C_\sigma$ may not be directly defined. Instead, it may be inferred by a number of relationships with intermediary semantic concepts. In such cases, the number of mappings is proportional to the number of intermediary concepts. More specifically, if $N$ is the number of intermediary concepts that are necessary to infer the semantic inclusion between the concepts $C_\tau$ and $C_\sigma$, the number of required mappings, denoted $NbofMappings(C_\tau, C_\sigma)$, is equal to $(2 \cdot N) + 4$. In the case that the XML schemas of $A$ and $B$ are identical, and there is no need for syntactic mappings the number of required mappings is set to 1.

In order to replace the service $s_\tau$ with a service $s_\sigma$, the required number of XML transformations is computed as the sum of all required mappings over the semantic concepts of their respective inputs. Hence, we have:

$$NbofMappings(b^s_\tau, b^s_\sigma) = \sum_{i,j=0}^{n} NbofMappings(C_{\tau,Inputs}(j), C_{\sigma,Inputs}(i))$$

such that $n$ is the number of inputs required by $b^s_\sigma$, and the input $i$ corresponds to the semantic matching of the input $j$.

Hence, we compute $NbofMappings$ when two candidate services have the same compatibility degree with the unavailable service. We then integrate $NbofMappings$ in the compatibility degree with respect of the supertype signature ($CD_{Signature}$): the less $NbofMappings$ is, the higher is the $CD_{Signature}$. Hence, $CD_{Signature}$ is inversely proportional to $NbofMappings$. The equation that evaluates $CD_{Signature}$ is computed following the equation below:

$$CD_{Signature}(F^s_\tau, F^s_\sigma) = \frac{CapabilityMatching(capa^s_\tau, capa^s_\sigma) \cdot SyntacticMapping(b^s_\tau, b^s_\sigma)}{NbofMappings(capa^s_\tau, capa^s_\sigma)})$$

## 6.3 Complying with Supertype Pre- and Post-conditions

Besides the compliance with the supertype signature, the substitute service has to respect the pre- and post-conditions, of the unavailable services.

As presented in Chapter 2, the pre- and post-conditions rules are expressed using SWRL. SWRL allows users to write Horn-like rules expressed in terms of OWL concepts. We also use SWRL to express invariants and constraints that a service may define. The general form for these rules is expressed as follows [W3C, 2004c]:

$$Body \Rightarrow Head$$

where *body* and *head* are a set of conjunctions of atoms, which are unary or binary predicates, i.e., properties that assigns truth values to combinations of $k$ individuals/variables ($k \in \{1, 2\}$).

Let $P = \{p_i, i = 0..n\}$ be the set of predicates used in the service pre- and post-conditions and $V = \{v_i, i = 0..n\}$ the set of variables. A rule represents the implication $(Body, Head)$, where both $Head$ and $Body$ are functions that associate a set of predicates with a set of variables, and are expressed using a conjunction of these functions. We denote $Body(P, V) \Rightarrow Head(P, V)$

As the unavailable service and the substitute one may not define the same set of rules, complying with the supertype pre- and post-conditions requires matching the predicates and variables used by the service candidates, with the ones defined by the now unavailable service. This means that for each pre-condition defined by the candidate service, there exists a pre-condition defined by the unavailable, which implies the candidate service's one, and inversely for the post-conditions.

Let $R_\tau = (Body_\tau(P_\tau, V_\tau) \Rightarrow Head_\tau(P_\tau, V_\tau))$ be a pre-condition of the unavailable service, and let $R_\sigma = (Body_\sigma(P_\sigma, V_\sigma) \Rightarrow Head_\sigma(P_\sigma, V_\sigma))$ be the corresponding pre-condition of a candidate service for substitution. Verifying Rule [1] 5.6 established in the previous chapter, amounts to verify that the following rule holds.

$$(Body_\tau(P_\tau, V_\tau) \Rightarrow Head_\tau(P_\tau, V_\tau)) \Rightarrow (Body_\sigma(P_\sigma, V_\sigma) \Rightarrow Head_\sigma(P_\sigma, V_\sigma)) \quad (6.1)$$

However, the right and left parts of the implication are expressed using different set of predicates and variables. Thus, we have first to find a matching between the set of predicates and variables $(P_\tau, V_\tau)$ with $(P_\sigma, V_\sigma)$ in order to make uniform Rule 6.1, and then verify whether the rule holds.

The matching raises two main issues:

1. We have to use of the predicates $P_\sigma$ in the rule $R_\tau$, while $R_\tau$ is originally defined using a set of predicates $P_\tau$. This requires to find a matching between $P_\tau$ and $P_\sigma$.

2. Also, we have to find a matching between $V_\tau$ and $V_\sigma$.

Herein, we assume that services employ a commonly-used ontology for expressing the semantic concepts of the predicates they use. For each predicate $p_i \in P_\tau$ used in the rule $R_\tau$, we check the predicate $p_j \in P_\sigma$, such that the semantic concept of $p_i$ is equivalent to the one of $p_j$, i.e., $SemanticDist(p_i, p_j) = \chi$. Several efforts have been focusing on integrating the concepts of heterogeneous ontologies, these efforts can be integrated in our approach to enlarge the matching to other relationships, rather than equivalence only.

Regarding the set of variables, we take advantage of SWRL, which enables to make reference to the URI of the variables involved in the rule. In this way, any variable can be considered as an

[1] $b_\tau.pre[A(s_{pre})/s_{pre}] \Rightarrow b_\sigma.pre$

instance of an XML schema defined in the associated URI. These schemas can be semantically annotated using SAWSDL standard. Then, matching between variables $V_\tau$ and $V_\sigma$ is performed in the same way as matching between super and subtype signatures defined in the previous subsection.

Besides semantic matching, we have to perform a syntactic mapping between the variables in $V_\tau$ to variables in $V_\sigma$. The syntactic mapping between $v_i \in V_\tau$ to $v_j \in V_\sigma$ is possible if:

- There exists a finite set of semantic concepts $C_0, .., C_k$, $(k > 0)$ where $C_0$ is the semantic concept of $v_i$, and $C_k$ is $v_j$'s one.
- There exist a semantic concept $C_l$, $0 < l < k$, such that:
  - for each concept $C_i$, $i \in [0..l-1]$, there exist a lifting XSL transformation that transforms $C_i$ to $C_{i+1}$, and
  - for each concept $C_i$, $i \in [l..k-1]$, there exist a lowering XSL transformation that transforms $C_i$ to $C_{i+1}$.

The syntactic mapping $SyntacticMapping$ between each variable $v_i$ in $V_\tau$ that is involved in $R_\tau$ and the corresponding variable $v_j$ in $V_\sigma$ is computed as follows:

$$SyntacticMapping(V_\tau, V_\sigma) = \begin{cases} 1 & \text{if } \forall i,\ v_i \in V_\tau, \exists v_j \in V_\sigma, \text{such that} \\ & \text{the mapping from } v_i \text{ to } v_j \text{ is possible.} \\ 0 & \text{otherwise.} \end{cases}$$

Let $A_{pre}$ be the abstraction function that corresponds predicates of $P_\tau$ with the ones of $P_\sigma$, using semantic matching between their semantic concepts. Let $A_{var}$ be the abstraction function that corresponds variables of $V_\tau$ with the ones of $V_\sigma$, using semantic matching and syntactic mapping. Rule 6.1 can then be expressed as follows.

$$(Body_\tau[A_{pre}(P_\tau)/P_\tau, A_{var}(V_\tau)/V_\tau] \Rightarrow Head_\tau[A_{pre}(P_\tau)/P_\tau, A_{var}(V_\tau)/V_\tau]) \Rightarrow (Body_\sigma(P_\sigma, V_\sigma) \Rightarrow Head_\sigma(P_\sigma, V_\sigma)) \tag{6.2}$$

Based on the above, if $SyntacticMapping(V_\tau, V_\sigma) \neq 0$, then complying with the supertype pre- and post-conditions consists in verifying rules that follow the same model as Rule 6.2. However, as aforementioned in Chapter 2, SWRL supports only the operators conjunctions and implications. Hence, every rule in the form of Rule 6.2 has to be transformed in a set of conjunctions and implications.

Using Morgan's laws in traditional logic, which state that $A \Rightarrow B \equiv \neg A \vee B$, we have

$$(Body_\tau \Rightarrow Head_\tau) \equiv (\neg Body_\tau \vee Head_\tau) \text{ , and}$$

$$(Body_\sigma \Rightarrow Head_\sigma) \equiv (\neg Body_\sigma \vee Head_\sigma)$$

Hence, Rule 6.1 can be written as follows:

$$(Body_\tau(P_\tau, V_\tau) \wedge \neg Head_\tau(P_\tau, V_\tau)) \vee (\neg Body_\sigma(P_\sigma, V_\sigma)) \vee (Head_\sigma(P_\sigma, V_\sigma))$$

Therefore, the verification of the rule 6.1 is split into 3 rule verifications, where one of the following rules has to be true:

$$Body_\tau[A_{pre}(P_\tau)/P_\tau, A_{var}(V_\tau)/V_\tau] \wedge \neg Head_\tau[A_{pre}(P_\tau)/P_\tau, A_{var}(V_\tau)/V_\tau])$$
$$\neg Body_\sigma(P_\sigma, V_\sigma)$$
$$Head_\sigma(P_\sigma, V_\sigma)$$

such that $A_{pre}(P_\tau) \subseteq P_\sigma$, and $A_{var}(V_\tau) \subseteq V_\sigma$.

If one of the pre- and post-conditions is not verified, then the substitution candidacy of the service does not hold. To this aim, the equation that computes $CD_{Pre-Post}$ is defined as follows.

$$CD_{Pre-Post}(b^s_\tau, b^s_\sigma) = \begin{cases} 1 \text{ if } \forall R^j_\sigma \text{ in the pre-conditions of } b^s_\sigma, \\ \quad \exists R^j_\tau \text{ in the pre-conditions of } s_\tau, \text{ such that:} \\ \quad\quad R^j_\tau \Rightarrow R^i_\sigma \\ \quad \text{and} \\ \quad \forall R^i_\tau \text{in the post-conditions of } b^s_\tau, \\ \quad \exists R^j_\sigma \text{ in the post-conditions of } s_\sigma, \text{ such that:} \\ \quad\quad R^j_\sigma \Rightarrow R^i_\tau \\ 0 \text{ otherwise.} \end{cases}$$

To verify SWRL rules, we reply on the ongoing efforts that focus on implementing reasoner suporting SWRL.

Amon them, we list

1. Hoolet [2] is an implementation of an OWL-DL reasoner that uses a first order prover. Verifying SWRL rule amounts to translate SWRL into First Order Logic and demonstrate reasoning tasks with Hoolet theorem prover;

2. Bossam [3] is a forward-chaining rule engine for the semantic web, that supports OWL inferencing, query processing, SWRL reasoning, etc. Verifying SWRL rule amounts to translate OWL-DL into rules and gives the rules to the Bossam forward chaining engine.

3. RacerPro [4] provides a first implementation that supports processing of rules in a SWRL-based syntax.

## 6.4 Complying with Rules for Runtime Execution Resumption

In the previous chapter, we established the algorithm that finds (if not null) the checkpoint (denoted $ckpt_\sigma$) from which the execution of the substitute service can be resumed, and which leads to a final node that serves the user required results. The structural/state compatibility degree is computed according to the three following steps:

1. Matching between the final nodes of the substitute service and user requested results.

2. The post-conditions of the unavailable service comply with the pre-conditions of the substitute service checkpoint.

3. Matching between the state description of the state that has to transferred and the one of the checkpoint $ckpt_\sigma$.

Let $RequiredCapa$ be the user requested capability such that

$$RequiredCapa = (C_{RequiredFunc}, C_{ProvidedInputs}, C_{RequiredResults})$$

[2]Hoolet: http://owl.man.ac.uk/hoolet/
[3]Bossam:http://projects.semwebcentral.org/projects/bossam/
[4]RacerPro:http://www.racer-systems.com/products/racerpro/index.phtml

Let $b^s_\tau$ be the behavior of the unavailable service, the execution of which has been interrupted at the node $N_{interrupt}$. Let $ckpt_\tau$ the *state transfer* checkpoint a predecessor of $N_{interrupt}$, issued from the sequential decomposition of $b^s_\tau$. Let $b^{s,i}_\tau$ is the actomic action that $ckpt_\tau$ terminates. Let $s_\sigma$ be a candidate service for substitution implementing the behavior $b^s_\sigma$. To realize the above steps, $b^s_\sigma$ should provide

1. a final node $N_{F,\sigma}$ that matches with $C_{RequiredResults}$.
2. a state transfer checkpoints $ckpt_\sigma$ that initiates an atomic action, the pre-conditions of which are implied by the post-conditions of the atomic action that $ckpt_\tau$ completes.
3. the description of the workflow state of the checkpoint $ckpt_\sigma$ matches with $ckpt_\tau$'s one.

Checking the compliance of the post-conditions with the pre-conditions of atomic actions amounts to check iteratively in the backward path of the final node $N_{F,\sigma}$ whether the post-conditions of $b^{s,i}_\tau$ imply the pre-conditions of an atomic action $b^{s,j}_\sigma$. If $b^{s,j}_\sigma$ exists then, $ckpt_\sigma$ is computed after sequentially decomposing $b^s_\sigma$. The compatibility degree associated with the second step consists in computing

$$CD_{Pre-Post}(b^{s,i}_\tau, b^{s,j}_\sigma) = \begin{cases} 1 & \text{if } \exists b^{s,j}_\sigma \in b^s_\sigma, \text{ such that:} \\ & b^{s,i}_\tau.post \Rightarrow b^{s,j}_\sigma.pre \\ 0 & \text{otherwise.} \end{cases}$$

Also, computing the structural/state compatibility degree includes matching between the state descriptions. Hereafter, we define the degree of matching between state descriptions (Section 6.4.1). Then, we compute the compatibility degree with respect to the rule for runtime execution resumption (Section 6.4.2).

### 6.4.1 State Description Compatibility

The key idea that governs the definition of the compatibility over states is to achieve a user-transparent or near transparent service substitution. To this aim, the substitute service has to require less data than (or the same as) the data provided by the last state stored of the unavailable service. Hence, as for the inputs, the state descriptions of the unavailable service and its substitute should be contravariant.

In Chapter 2, we presented our extension of WSFR standard with semantic annotations, which we generate automatically (Chapter 4). Checking the compatibility between state descriptions consists in matching between their respective XML elements. Matching between state descriptions is performed in the same way as matching between the in/out parameters of the signature. We first compute the semantic distance between the semantic description of the XML elements included in the state description $ckpt_\sigma$ with those included in the state description of $ckpt_\tau$. Then, we check whether the syntactic mapping is possible.

Let $SD_{wkf}(ckpt_\sigma)$ be modeled as a set of XML element $E_{\sigma,i}$ making reference to the semantic concept $C_{\sigma,i}$, where $i \in [1..n]$, and $n$ is the cardinality of $SD_{wkf}(ckpt_\sigma)$. Let $SD_{wkf}(ckpt_\tau)$ be modeled as a set of XML element $E_{\tau,j}$ making reference to the semantic concept $C_{\tau,j}$, where $j \in [1..m]$ and $m$ is the cardinality of $SD_{wkf}(ckpt_\tau)$, such that for each $C_{\sigma,i}$, $\exists C_{\tau,j} \mid C_{\sigma,i} \subseteq^s C_{\tau,j}$, in order to comply with the contravariance constraint. We denote $DM$ the degree of matching between state descriptions, which is computed as follows.

$$DM(SD_{wkf}(ckpt_\tau), SD_{wkf}(ckpt_\sigma)) = \prod_{i,j=0}^{n,m} SemanticDist(C_{\sigma,i}, C_{\tau,j}) \cdot \prod_{i,j=0}^{n,m} SyntacticMapping(E_{\tau,i}, E_{\sigma,j})$$

Note that if $\exists i, j$, such that one of the terms $SemanticDist(C_{\sigma,i}, C_{\tau,j})$ or $SyntacticMapping(E_{\tau,j}, E_{\sigma,j})$ is null then $DM(SD_{wkf}(ckpt_\tau), SD_{wkf}(ckpt_\sigma))$ would be null.

### 6.4.2 Compatibility Degree Computing for Runtime Execution Resumption

According to the degree of matching $DM$ between the state descriptions, we define the compatibility degree for the structural/state compatibility (i.e., $CD_{RuntimeExeRes}$).

The structural/state compatibility degree is computed according to the above-stated steps, namely,

1. The degree of matching between the final nodes, denoted $DM(N_{F,\sigma}, C_{RequiredResults})$, which is a boolean parameter $\in \{0, 1\}$, that equals to 1 only when all the required results of the user are included in $SD_{wkf}(N_{F,\sigma})$.

2. $CD_{Pre-Post}(b_\tau^{s,i}, b_\sigma^{s,j})$ which is a boolean parameter in $\{0, 1\}$, enabling to check the existence of an atomic action $b_\sigma^{s,j} \in b_\sigma^s$, the pre-condition of which are implied by the post-conditions of $b_\tau^{s,i}$.

3. The degree of matching, denoted $DM(SD_{wkf}(ckpt_\tau), SD_{wkf}(ckpt_\sigma))$ between the state descriptions of $ckpt_\tau$ and $ckpt_\sigma$.

Thus, the structural/state compatibility degree is set as follows.

$$CD_{RuntimeExeRes}(F_\tau^s, F_\sigma^s) = \prod \begin{cases} DM(N_{F,\sigma}, C_{RequiredResults}) \\ CD_{Pre-Post}(b_\tau^{s,i}, b_\sigma^{s,j}) \\ DM(SD_{wkf}(ckpt_\tau), SD_{wkf}(ckpt_\sigma)) \end{cases}$$

where

$$DM(N_{F,\sigma}, C_{RequiredResults}) = \begin{cases} 1 & \text{if } (\forall p \in C_{RequiredResults}), \exists q \in SD_{wkf}(N_{F,\sigma}) \\ & \text{such that the semantic concept of } p \subseteq^s q\text{'s one.} \\ 0 & \text{otherwise.} \end{cases}$$

Note that similarly to the compatibility degree over signatures, the structural/state compatibility degree is the product of the parameters that have impact on it, if one of them is null, then $CD_{RuntimeExeRes}$ is also null.

## 6.5 Complying with Subtype Invariants and Constraints

The transferred state of the substitute service should comply with the invariants and constraints of the substitute service. To check the compliance with invariants and constraints, we have to check that the following rules hold: $I_\sigma[A(\omega)/\omega]$ and $Constrt_\sigma[A(\omega)/\omega]$, where $\omega$ is the state of the unavaialbels ervice that has to be transferred to the substitute service, $A$ is the abstraction function that associates to each variable in the state of the substitute service, with the corresponding variable in the state of the unavailable service. The association is guaranted if $DM(SD_{wkf}(ckpt_\tau), SD_{wkf}(ckpt_\sigma))$ is not null. Furthermore, $A$ assigns variables values to the state of the unavailable service using syntactic mapping between state variables. Hence, complying with invariants requires that the state resulting from semantic matching and syntactic mapping, i.e., $A(\omega)$ complies with the invariant of the candiadte service. This is reduced to

transform that the rule $I_\sigma(A(\omega)$ into a set of rule that include only conjunctions and implication as in the case of pre- and post conditions and to verify the resulting rule.

Regarding the constraints, we check the compliance only in the case that the checkpoint $ckpt_\sigma$ is included between two nodes, on which there is a constraint. In this case, similarily to the invariants, we transform $Constrt_\sigma[A(\omega)/\omega]$ into a set of rule that include only conjunctions and implications, and verify the resulting rule. Hence, the compatibility degree with respect to the substitute service invariants and constraints is expressed as follows:

$$CD_{Inv-Const}(b^s_\tau, b^s_\sigma) = \begin{cases} 1 & \text{if } \forall I^j_\sigma \text{ invariant of } b^s_\sigma, \\ & \quad I^j_\sigma[A(\omega)/\omega] \text{ holds} \\ & \quad \text{and} \\ & \quad \text{if } \exists n_\rho, n_\psi \in b^s_\sigma, \text{ and a constraint } Constrt(n_\rho, n_\psi), \text{ such that:} \\ & \quad ckpt_\sigma \text{ is included in the path that links } n_\rho \text{ to } n_\psi \\ & \quad \text{then } Constrt_\sigma[A(\omega)/\omega] \text{ must hold.} \\ 0 & \text{otherwise.} \end{cases}$$

## 6.6 Semantic-based Service Selection

In this section, we put the different compatibility degrees computed in the previous section together in order to check whether a candidate service is able to substitue the unavailable service or not. In the case of multiple candidates, we classify the candidates for substituting the unavailable service in order to increase the chances to proceed a successful service substitution.

First, coarse-grained classification puts services into catalogs where each catalog is related to a user requested capability. We then refine the classification within the same catalog according to their compatibility degree. When a failure occurs, the candidate service that presents the greatest compatibility degree is selected. At runtime, the service catalogs are continuously enriched with new services that join the networked environment.

Upon service unavailability, the services belonging to the related catalog are processed by computing their compatibility degree with the unavailable functionality in order to select the service that is best compatible with the unavailable service. Figure 6.2 presents a graphical representation, as a decision graph, of the mechanism that computes the compatibility degree with the unavailable service's functionality.

The decision graph is set as follows. The compatibility degree ($CD$) is initially null. In order to determine its value, the computation of the compatibility degree over signatures ($CD_{Signature}$) is first computed with respect to the semantic and syntactic descriptions. In case that $CD_{Signature}$ is null, the candidacy of the service is not considered. Otherwise, the value of $CD$ is set to $CD_{Signature}$, and the conformance with the unavailable service's pre- and post-conditions is checked. At the end of the compatibility check over pre- and post-conditions, $CD$ is set to the weighted sum of $CD_{Signature}$ and $CD_{Pre-Post}$, or it is null, in the case that $CD_{Pre-Post}$ is null. If $CD$ is null, then the candidacy of the service is not considered. Otherwise, the structural/state compatibility is checked.

Regarding structural/state compatibility, the compatibility degree includes (1) matching between the final nodes' state descriptions and the user required results, (2) checking that the runtime rule for execution resumption is satisfied, and (3) matching between state descriptions.

As specified in Section 6.4, the compatibility between the user request and the final nodes is

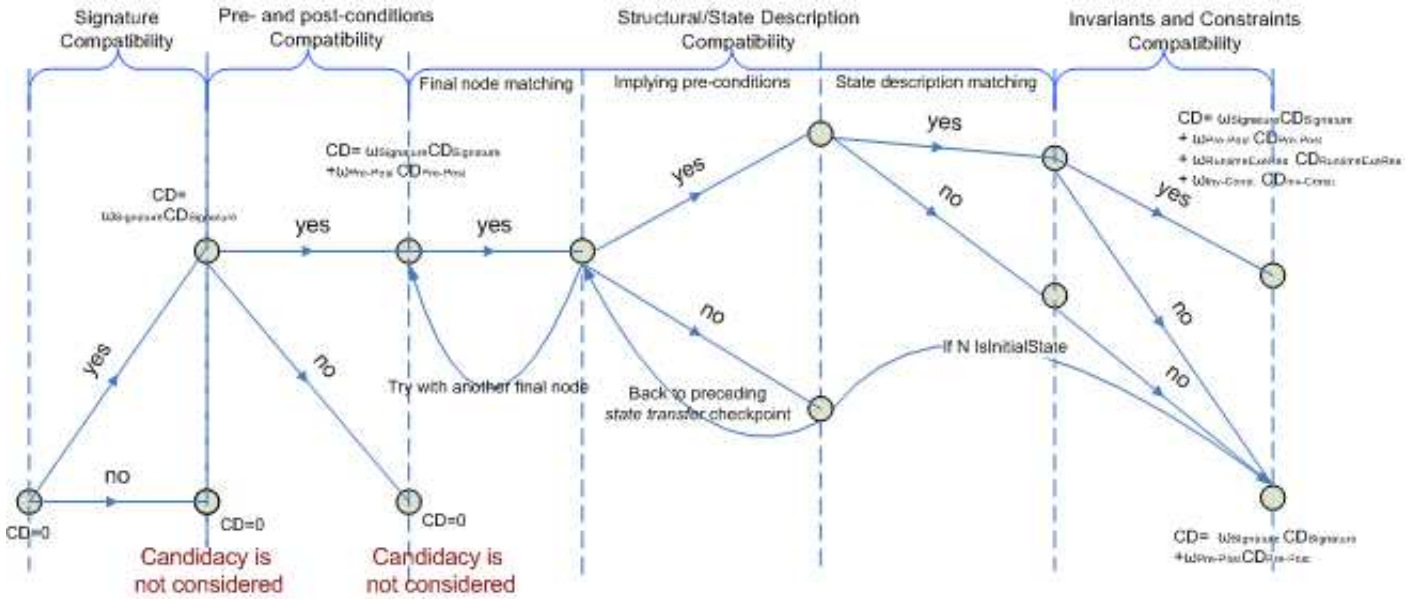



Figure 6.2: Decision graph for computing the compatibility degree

first checked. The existence of such a node is guaranteed through the compliance of the substitute service's signature with the covariance of results. Once a final state $N_{F,sigma}$ that satisfies the user request is found, the compatibility is checked with respect to the pre- and post conditions of atomic actions and state descriptions. This enables to find (if exists) in the backward path of $N_{F,sigma}$, a *state transfer* checkpoint $ckpt_\sigma$ in the workflow of the candidate service, at which the state can be transferred.

In case of incompatibility with $ckpt_\sigma$, the backward path is expanded to the predecessor of $ckpt_\sigma$. In case that the backward path reaches the initial state, then $CD_{RuntimeExeRes} = 0$ and $CD$ is equal to the weighted sum of $CD_{Signature}$ and $CD_{Pre-Post}$.

Otherwise, we check whether the state that should be transferred complies with the invariants and constraints of the candidate service. If it is the case then the compatibility degree also includes the weighted sum of $CD_{RuntimeExeRes}$ and $CD_{Inv-Const}$. Otherwise, it includes none of them, as the state transfer cannot be performed if one of them is null.

Once the compatibility degrees with the services listed in the catalog are computed, the service providing the greatest value of the compatibility degree is selected as a candidate for substitution, and a 'SetState' query is sent to the selected service.

## 6.7 Concluding Remarks

In this chapter, we defined our algorithms to check and evaluate the compatibility between the unavailable service and a set of candidate services for substitution. Once the compatibility is checked and evaluated, a service is selected in order to proceed to the actual substitution of the unavailable service. This requires first to reconfigure all the parties (i.e., the client, and possibly, other services) that have been interacting with the unavailable service, in order to use the substitute service in the place of the unavailable service. The integration of the substitute service in the running SOA systems is elaborated in the next chapter.

*An error doesn't become a mistake until you refuse to correct it.*

Orlando A., Battista

# 7 Reconfiguring SOA Systems

In the previous chapters, we focused on service substitution: Chapter 5 defines the rules that ensure correct service substitution and, Chapter 6 defines our technique for verifying these rules on networked services. Assuming that a service substitute has been selected among a set of candidate services, we investigate, in this chapter, the impact of the service substitution on the client, and potentially, the other services involved in the SOA systems, which are affected by the service unavailability.

More specifically, the runtime reconfiguration consists in substituting the unavailable service with a similar service available in the networked environment, and transferring the state of the unavailable service to its service substitute. Since it is not always possible to find an exact replica of the unavailable service, the behaviors of the unavailable and substitute services may differ. In such cases, the client reconfiguration consists in adapting the client interactions according to the behavior description of the substitute service. Also, in the case of service orchestration, the other still-available services –involved in the orchestration– may have data dependencies with the unavailable service. These services have to be reconfigured according to their data dependency.

The rest of this chapter is structured as follows. In Section 7.1, we focus on the client reconfiguration after service substitution, and in Section 7.2, we investigate the reconfiguration of service orchestrations in the presence of data dependencies with the substituted service. Finally, Section 7.3 provides a summary of our approach, and points out the need for realizing the solution that we propose for runtime service substitution.

## 7.1 Client Reconfiguration

The substitution that we propose allows the substitute service to have a behavior different from the one of the unavailable service. As presented in the previous chapter, we check the structural/state compatibility between the behaviors of the unavailable and substitute services. As a result, if the behaviors are compatible, we find a checkpoint in the behavior of the substitute service at which the state synchronization is possible. As illustrated in Figure 7.1, Figure (a) is a graphical representation of the behavior (denoted $b_\tau$) of the unavailable service (denoted $s_\tau$), and Figure (b) represents a substitute service's one (denoted $b_\sigma$).

Before service unavailability, the client starts interacting with the service $s_\tau$ with respect to its behavior description. A set of operations are invoked and a set of checkpoints are performed. When $s_\tau$ becomes unavailable and a substitute service $s_\sigma$ is found, the compatibility between $b_\tau$ and $b_\sigma$ is checked. Once, we find two *state transfer* checkpoints, $ckpt_\tau$ and $ckpt_\sigma$, that have compatible states, the client reconfiguration is performed in three steps:

1. In a first step (❶ in Figure 7.1), a roll back is performed at the client side in order to invali-

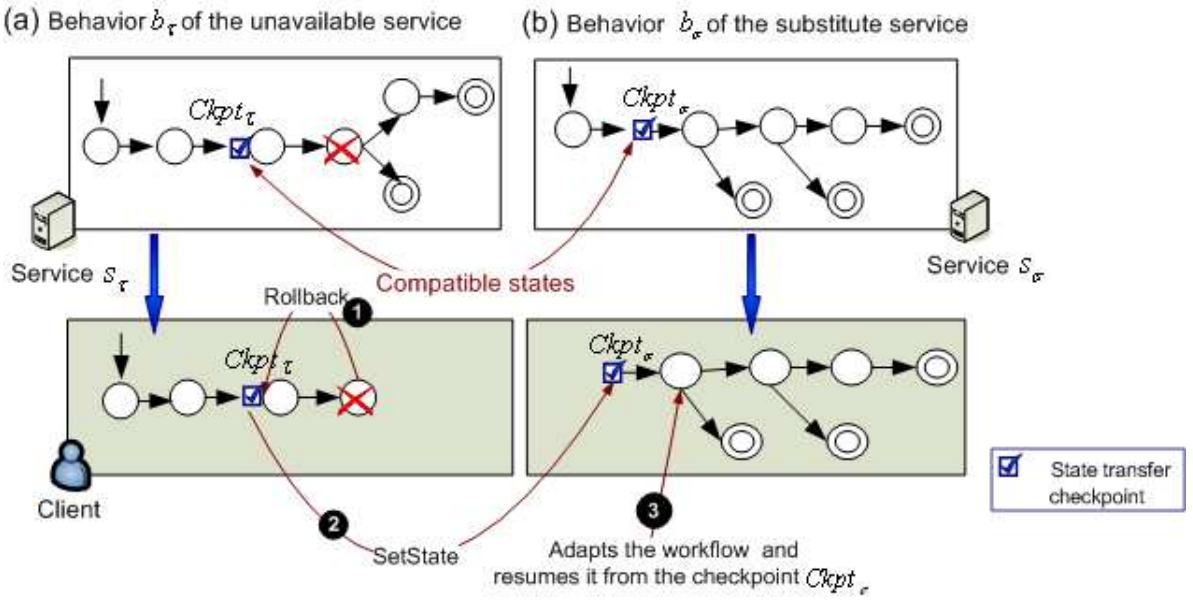



Figure 7.1: Client reconfiguration

date the set of interactions performed from $ckpt_\tau$ till the service unavailability, represented using a cross in the Figure 7.1.

2. The second step (❷ in Figure 7.1) consists in sending a 'SetState' request to the substitute service $s_\sigma$.

3. Finally (step ❸ in Figure 7.1), the client adapts its interactions according to the behavior $b_\sigma$, starting from the checkpoint $ckpt_\sigma$ till reaching a final node.

Note that case that the services are structural/state incompatible, a replay is necessary to put the substitute in an advanced stage of execution. The replay consists in executing the behavior $b_\sigma$ of the substitute service seamlessly to the client, as if the execution has not been interrupted.The replay stops when a user intervention is necessary. The client reconfiguration then includes adapting the workflow with respect to $b_\sigma$, starting from the last node at which the replay has been stopped. In such cases, the interaction with the client does not start with a rolled back state, but with a new state reflecting the state of execution of the substitute service. The client has to be updated according to the replay results.

## 7.2 Reconfiguration of Service Orchestrations

State transfer may invalidate a set of results that are not taken into account into the transferred state. In the case that the unavailable service has been participating in an orchestration may affect still-connected services due to data dependencies. Services that are dependent upon these results should be forced to rollback to a previous checkpoint in order to put the orchestration in global consistent state. As mentioned in Section 3.2.2, the set of these checkpoints is called *consistent recovery line*.

Because services are autonomous, their checkpoints are not *a priori* coordinated to form a consistent recovery line. Thus, the reconfiguration should be figured out on the basis of the

analysis of their data dependencies. Hence, in this section, we first model the data dependency between services as a direct acyclic graph (Section 7.2.1). The resulting representation of data dependency enables to easily detect the data dependency between services at reconfiguration time. Then, we propose our algorithm for rollback propagation, which adapts checkpoint-based recovery to SOA systems (Section 7.2.2). Then, in Section 7.2.3, we integrate the behavior of the substitute service in the orchestration workflow, and update the data flow of the orchestration according to the substitute service behavior in order to resume the execution from a coherent orchestration description.

### 7.2.1 Data Dependency Between Services

In order to model the data dependencies between services participating in a service orchestration, we use a dependency graph that links service checkpoints according to their data dependencies. The dependency is considered between *rollback* checkpoints defined in the service workflows, because they present the checkpoints at which the state can be stored and synchronized.

#### Principles of dependency between checkpoints

To illustrate the data dependency between two checkpoints, we use the example presented in Figure 7.2.

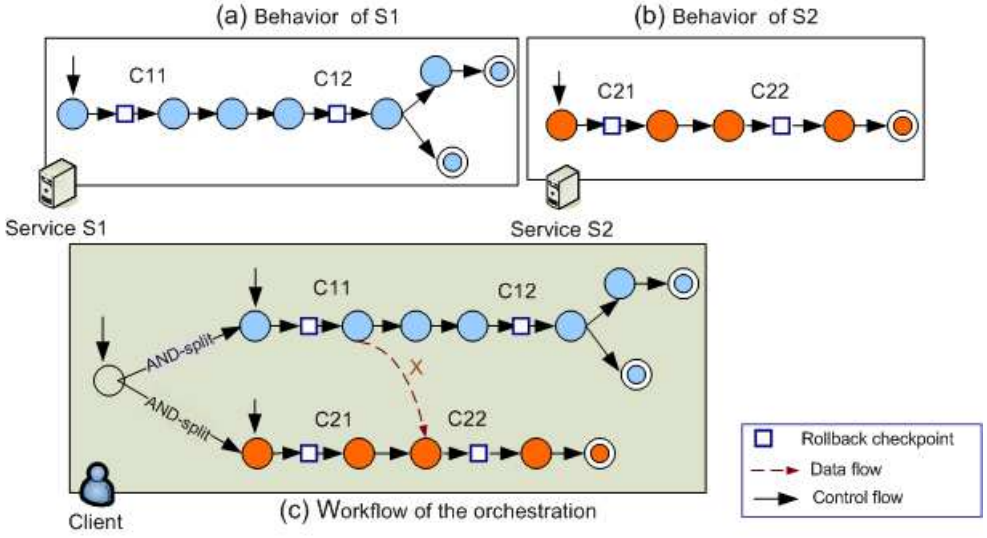



Figure 7.2: Data dependency between two checkpoints in a service orchestration

The service orchestration that we consider integrates two services, namely, $S_1$ and $S_2$. It integrates their behaviors (represented respectively, in Figure 7.2- (a) and (b)) in an AND-split structure. As represented in the orchestration data flow (Figure 7.2- (c)), the value of the parameter $X$ is provided as an output by the service $S_1$, and used as an input by the service $S_2$. Before data passing, the services $S_1$ and $S_2$ have performed respectively the checkpoints $C_{11}$ and $C_{21}$. We define the data dependency between two *rollback* checkpoints as follows.

**Definition 6.** A *rollback* checkpoint (denoted $C_{21}$) depends upon another *rollback* checkpoint (denoted $C_{11}$) means that rolling back to $C_{11}$ implies necessarily rolling back to $C_{21}$.

In the example presented in Figure 7.2, rolling back the service $S_1$ to the checkpoint $C_{11}$ invalidates the previously provided value of $X$, let $V_{pre}$ be this value. Re-executing the nodes after the checkpoint $C_{11}$ may provide a new value of X, , let $V_{pre}$ be this value. $V_{post}$ may different from $V_{pre}$. The service $S_2$ must also roll back to a state that precedes the use of $X$'s value (i.e., $V_{pre}$) as input for its operations; otherwise, the states of the two services would be inconsistent.

### Building the dependency graph

A *dependency graph* is an oriented graph where the nodes represent the individual *rollback* checkpoints of constituent services. The edges represent data dependencies between these checkpoints. An arrow goes from a checkpoint $C_1$ to another $C_2$ to denote that if we rollback to the checkpoint $C_1$, we have to rollback to $C_2$. Dependency detection is based on the orchestration data flow. As defined in Chapter 4, the data flow shows how the data is passed from a node to another as in/out parameters. Since the client middleware is the central entity that sends, receives and manipulates these parameters, tracking data flow at the client side allows determining checkpoint dependencies.

Algorithm 3 presents the set steps that enable the automatic building of the dependency graph for a service orchestration ($O$). Let $O.wkf$ be the aFSA that models the workflow of the service orchestration, which integrates the behaviors of the $N$ services $S_i$, $i \in [1..N]$. Each service $S_i$'s behavior is also modeled using an aFSA (denoted $wkf_i$), such that $wkf_i = (Q_i, \sigma_i, \delta_i, s_{i0}, F_i, QA_i, STCkpt_i, RCkpt_i)$.

After creating a new dependency graph (denoted $O.DG$) that is associated with the orchestration $O$, the second step consists in identifying the set of parameters used in the orchestration (denoted $Param$). As modeled in Chapter 4, the set of parameters for each aFSA $wkf_i$ is modeled as a set $P_i = \cup_j(O_{ij} \cup I_{ij})$, where $I_{ij}$ and $O_{ij}$ are the in/out parameters of the nodes $n_{ij}$, such that $n_{ij} \in Q_i$ of the workflow $wkf_i$. In this way, we have $Param = \cup_{i=0}^{N} P_i$.

Afterwards (in Step 3), Algorithm 3 checks for each parameter $P \in Param$, the data dependencies that result from changing $P$'s value and enriches the dependency graph accordingly. This step is split into 5 steps: from 3.1 to 3.5.

Let $CkpInt_k^P = [C1_k^P..C2_k^P]$ be a checkpoint interval [1] in $wkf_k^P$, which holds between the checkpoint $C1_k^P$ and its immediate successor $C2_k^P$, where $P$'s value is changed (i.e., where $P$ is an out or in/out parameter of a node $n_{kj} \in CkpInt_k^P$ of the workflow of the service $S_k$).

Between two changes of $P$'s value, Algorithm 3 checks the checkpoint intervals $CkpInt_i$ of the other services $S_i$, $i \in [0..N]\backslash\{k\}$ of the orchestration, which consume $P$ without changing its value (i.e., $P$ is an input parameter of a node $n_{ij} \in Q_i$). As we focus on inter-service dependencies, $S_i$ should necessarily be different from $S_k$. We denote $CkpInt$ the set of checkpoint intervals $CkpInt_i$, $i \in [0..N]\backslash\{k\}$ that consume $P$'s value without changing it ($P$ is an input only). This is performed in Steps 3.1 to 3.4.

Then, Step 3.5 adds nodes and their corresponding links to the dependency graph. For each checkpoint interval $CkpInt_i = [C_i..C_i']$ included in $CkpInt$, an arrow that goes from $C1_k^P$ to $C_i$ is added to the orchestration dependency graph ($O.DG$), denoted $C1_k^P \rightarrow C_i$, which means that a roll back of the service $S_k$ to the checkpoint$C1_k^P$ implies the roll back of the service $S_i$ to $C_i$. Furthermore, in order to reduce the number of nodes in the dependency graph as well as the links among them, adding a link among two checkpoints ($C1_k^P \rightarrow C_i$) depends on the existing dependencies in the graph. More specifically, Step 3.5.3 checks whether a predecessor (denoted $C"_i$) of a checkpoint $C_i$ (of a same service $S_i$) depends upon the checkpoint $C1_k^P$, then there is

[1] A checkpoint interval is a set of nodes that are included between two successive checkpoints.

**Algorithm 3**: Constructing Dependency Graph

/* Constructs an oriented dependency graph according to the orchestration workflow */
**Data**: Orchestration description $O.wkf$.
**Result**: Orchestration dependency graph $O.DG$.
**begin**
  /* Step 1: Initialization */
  Initialize $O.DG$= new Graph
  /* Step 2: Identifying the in/out parameters in the orchestrations */
  $Param = \cup_{i=0}^{N} P_i$
  /* Step 3: A loop that parses the orchestration and checks the data dependencies generated by each parameter */
  **foreach** $P \in Param$ **do**
    /* Initialize a marker (Mark) to parse the orchestration description by parts */
    $Mark \leftarrow$ First node $n_{kj}$ where $P$ is output parameter of the service $S_k$
    /* Parse the orchestration description */
    **while** $Mark \neq$ *the final node of* $O.wkf$ **do**
      Step 3.1) Start parsing from $Mark$
      Step 3.2) Detect the checkpoint interval $CkpInt_k^P = [C1_k^P..C2_k^P]$ of the service $S_k$ where $P$'s value is first changed
      Step 3.3) Track $CkpInt$, which is the set of checkpoint intervals $CkpInt_i$ in $wkf_i$ of the service $S_i$, such that $S_i \neq S_k$, in which the parameter $P$ is taken as input of a node $n_{ki} \in Q_i$
      This step is stopped either when changing $P$'s value, or at the end of $O.wkf$
      Step 3.4) $Mark \leftarrow$ the point of $O.wkf$ where the previous step stopped
      Step 3.5) **foreach** $CkpInt_i = [C_i..C_i'] \in CkpInt$ **do**
        3.5.1) Add $C_i$ to $O.DG$
        3.5.2) Create an arrow going from $C1_k^P$ to $C_i$, denoted $C1_k^P \longrightarrow C_i$ in $O.DG$
        /* Optimizing O.DG */
        3.5.3) **if** $\exists C"_i \in Predecessor(C_i)$ *such that* $(C1_k^P \longrightarrow C"_i) \in O.DG || C1_k^P$ *is annotated with* $C"_i$ **then**
          Delete $C1_k^P \longrightarrow C_i$
        **else if** $\exists$ *cycle between* $C1_k^P$ *and* $C_i$ **then**
          /* Checking cycles between Cp and Ci */
          Annotate $C1_k^P$ with $C_i$
          Delete $C1_k^P \longrightarrow C_i$
  Return Orchestration Dependency Graph $O.DG$
**end**

no need to add a dependency between $C_i$ and $C1_k^P$ since rolling back to $C1_k^P$ will imply rolling back to $C"_i$, thus, implicitly rolling back to $C_i$.

However, the above built dependency graph may include cycles. Here below, we focus on freeing the dependency graph from cycles in order to be a DAG (Direct Acyclic Graph), and thus take benefit of the DAG properties in term of graph-traversal and complexity.

### Cycle free dependency graph

The logic behind the dependency graph is based on the precedence relation (relation before) introduced by Leslie and Lamport [Lamport, 1978] over the checkpoints of the services involved in the orchestration. Data are generated and/or parameters' values are changed, and exchanged between nodes included in the preceding or simultaneous checkpoint intervals, producing thereby a data dependency between checkpoints. As presented above, we model these dependencies using a dependency graph. Under certain circumstances, the checkpoint intervals of different services include nodes that present in/out dependencies in inverse directions, introducing dependency cycles between checkpoints.

A cycle is formed when a number of checkpoints in the dependency graph are connected in a closed chain, i.e., when there is a direct path that goes from one checkpoint to the same checkpoint. For instance, consider the case where the values of two parameters, e.g., $X$ and $Y$, are changed within two checkpoint intervals, e.g., respectively $CkpInt_1 = [C_{11}..C_{12}]$ in the service $S_1$, and $CkpInt_2 = [C_{21}..C_{22}]$ in the service $S_2$. Then $X$ is provided as input of a node in the checkpoint interval $CkpInt_2$, and $Y$, in $CkpInt_1$ (as illustrated Figure 7.3 -left).

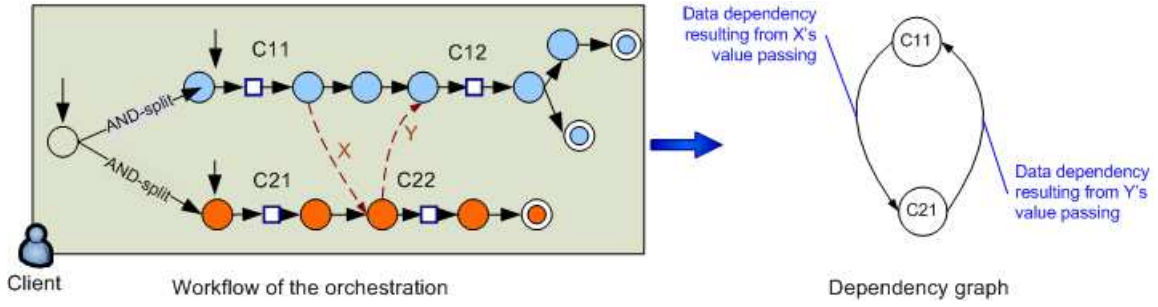



Figure 7.3: Case of cycle in the dependency graph

The dependency graph thus includes a 2-vertice cycle formed by the checkpoints $C_{11}$ and $C_{21}$ (Figure 7.3 -right). The cycle means that in case of rolling back to $C_{11}$, this implies rolling back the service $S_2$ to $C_{21}$ and *vice versa*. Hence, in both cases of rollback, the rollback propagation will include the checkpoints $C_{11}$ and $C_{21}$. If the dependency $C_{11} \rightarrow C_{21}$ has been first detected and an arrow that goes from $C_{11}$ to $C_{21}$ is created, then when detecting the dependency $C_{21} \rightarrow C_{11}$, we annotate $C_{21}$ with $C_{11}$ without introducing an extra arrow between the checkpoints. The meaning of the annotation is that when rolling back the service $S_2$ to the checkpoint $C_{21}$, the rollback should be necessarily be propagated to the checkpoint $C_{11}$ of the service $S_1$. In this way we avoid the creation of cycles in the dependency graph, which facilitates the graph traversal at runtime. This is performed in Step 3.5.3 in Algorithm 3.

### Complexity of the dependency graph

Let $N$ be the number of services involved in the orchestration, each service $S_i$ has $K_i$ checkpoints included in its behavior. Let's call $C = max_{1 \leq i \leq N}(K_i)$. Therefore, in the worst case, the

dependency graph will have $(C \cdot N)$ nodes. Let $M$ be the maximum number of nodes in the dependency graph, $M = C \cdot N$.

As the dependency graph does not include cycles, the maximum number of dependencies of a given checkpoint is equal to $(M-1)$. More specifically, the number of data dependencies can be modeled as the sum of $M$ first terms of an arithmetic sequence $(U_n)$ having as step $r = 1$ where the general term is expressed as follows:

$U_{n+1} = U_n + 1$ for $0 \leq n \leq (M-1)$ and $U_0 = 0$

The maximum number of transitions $(E)$ of the dependency graph is thus the sum of $M$ first terms of $U_n$

$$E = \sum_{i=0}^{M-1} U_i = \frac{M \cdot (M-1)}{2}$$

Based on the above, the number of transitions of the dependency graph is $O(n^2)$, for $n$ number of nodes. Taking advantage of the existing algorithms for graph traversal, the dependencies of a given checkpoint $C$ are extracted after finding the checkpoint $C$ in the dependency graph using Breadth First Search [2] (BFS) and its tree. In the worst case breadth-first search has to traverse all paths to all possible nodes, the time complexity of breadth-first search can be expressed as $O(n+n^2)$ since every node and every transition will be explored. Hence, the time complexity of the dependency graph is polynomial.

### 7.2.2 Rollback Propagation for Service Orchestrations

Based on the above built dependency graph, we present our strategy for rollback propagation in service orchestrations. The strategy that we propose takes its origins from the existing protocols for rollback recovery (presented in Section 3.2.2). It derives from the uncoordinated checkpointing protocol as its the most flexible one: it respects service autonomy in deciding when taking checkpoints. We adapt the rollback propagation according to our definitions of checkpoints and the data dependencies between checkpoints. Here after, we present the high level steps of our rollback propagation algorithm.

The rollback propagation performed by Algorithm 4 takes as inputs (1) the checkpoint (denoted $C$) in the behavior of the unavailable service, at which the synchronization has been performed with the substitute service, (2) the orchestration workflow ($O.wkf$) and the associated dependency graph ($O.DG$). Then, Algorithm 4 recursively computes the recovery line (denoted $RecoveryLine$) of the orchestration, according to the orchestration's dependency graph.

More specifically, the recovery line is first initialized with the checkpoint $C$ in order to invalidate the computation that has been performed by the unavailable service. Then, the rollback is propagated according to the dependencies of $C$, i.e., the checkpoints $C'$ of the still-connected services involved in the running orchestration, which *depend upon* $C$ (i.e., $C \rightarrow C'$ in $O.DG$, or $C$ is annotated with $C'$). These checkpoints are thus added to the recovery line. Recursively, the propagation is extended to the checkpoints that depend upon the newly added checkpoints (i.e., the set of checkpoints $C'$) based on the same dependency graph.

To reduce the number of the checkpoints included in the recovery line, Algorithm 4 further checks if a successor of $C'$ (denoted $C$") has been previously included in the recovery line in order to remove it, since rolling back to $C'$ includes implicitly rolling back all its successors in the same service behavior. Furthermore, in order to avoid checking the dependencies twice for

[2] In graph theory, breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbor nodes, and so on, until it finds the goal.

**Algorithm 4**: Rollback_Propagate($C$)

```
   /* Computes a recovery line according to the dependency graph                    */
   Data: Rolled back checkpoint C, O.wkf and O.DG.
   Result: Orchestration recovery line RecoveryLine.
   begin
       /* Initialization                                                            */
       RecoveryLine=new set
       RecoveryLine.Add(C)
       /* Temporary variables                                                       */
       Checkpoint Temp =new Checkpoint
       /* Checking dependencies with C                                              */
       Temp = C
 7     Label: Propagate
       /* If Temp has dependencies                                                  */
       if Temp ∈ O.DG & ∃ Checkpoint C′: (Temp → C′) ∈ O.DG then
           /* Rolling back only services that are dependent on Temp                 */
           foreach Checkpoint C′:(Temp → C′) ∈ O.DG do
               /* Removing the successors of C′                                     */
               if ∃C" ∈ Successor(C′): C" ∈ RecoveryLine then
                   RecoveryLine.Remove(C")
               RecoveryLine.Add(C′)
               /* Mark Temp with which the dependency is checked                    */
               Mark Temp
       else
           Return RecoveryLine
17     End Propagate
       if ∃ non marked Checkpoint C′ ∈ RecoveryLine then
           /* Checking dependencies with newly added checkpoints                    */
           foreach non marked Checkpoint C′ ∈ RecoveryLine do
               Temp = C′
               /* Recursive Propagation                                             */
               Goto Propagate (7 → 17)
       Return RecoveryLine
   end
```

a same checkpoint, Algorithm 4 marks each checkpoint for which the propagation is computed according to its dependencies. At the end, Algorithm 4 verifies whether there are checkpoints included in the recovery line, and for which the dependencies have not been checked. In such a case, Algorithm 4 calls back the instructions in the block “Propagate" (from 7 to 17) in order to include in the recovery line the dependencies of non-marked checkpoints. Once Algorithm 4 is performed, the computed recovery line comprises all the checkpoints of the component services that present data dependency with the checkpoint $C$.

Our rollback recovery distinguishes itself from the traditional protocols by flexibility in computing the recovery time. since it significantly simplifies the existing rollback techniques. Indeed, it does not rely on any service synchronization or forced checkpoints. It further respects service autonomy while preserving orchestration consistency. However, as individual checkpoints are not coordinated, the risks for domino effect are the same as in uncoordinated checkpointing protocols. The domino effect is mainly caused by the dependencies among constituent services. The less the data dependencies are, the more unlikely is the domino effect. Therefore, the domino effect likelihood is conditioned by the dependencies between the constituent services of an orchestration. It thus relies essentially upon the orchestration design to limit the extent of the rollback propagation.

### 7.2.3 Integrating the Substitute Service in a Running Service Orchestration

The target of the previous section is to put the orchestration in a consistent state after service unavailability. Once the recovery line is computed and the orchestration is put in a consistent state, we have to integrate the behavior ($b_\sigma$) of the substitute service $s_\sigma$ in the workflow of the orchestration in order to resume its execution. The integration of the behavior of the substitute service is similar to the client reconfiguration, presented in Section 7.1: the non-executed part of the behavior of the unavailable service is replaced with the part of the behavior of the substitute service, resulting from the matching algorithms. In the recovery line, the checkpoint $C$ is replaced by the corresponding checkpoint included in the behavior of the substitute service, which results from behaviors matching.

However, as there may be data dependencies between services, we have to re-establish the data flow of the orchestration according to the behavior of the substitute service.

#### Integrating substitute service data flow

The data flow is defined on the basis of the previously defined orchestration. We parse the data annotations of the nodes that are included in the non-performed part of the behavior $b_\tau$ of the unavailable service, and:

1. For each output ($P$) provided by the unavailable service as input to a node $n_i$ in the behavior of another service $s_i$ in the orchestration, we find the node $n_\sigma = (I_\sigma, Op_\sigma, O_\sigma)$ in the aFSA of the substitute service, which provides this output, i.e., $P \in O_\sigma$. We create an annotation in $n_\sigma$ to denote the data dependency between $s_\sigma$ and $s_i$. As presented in Section 4.1, the data dependency is formalized here below:

$$(n_\sigma, \{P\}) \rightarrow (n_i, \{P\})$$

2. For each input ($P$) required by the unavailable service from a node $n_j$ in the behavior of another service $s_j$ in the orchestration, we find the node $n'_\sigma = (I'_\sigma, Op'_\sigma, O'_\sigma)$ in the aFSA of the substitute service, which requires $P$ as input, i.e., $P \in I'_\sigma$. We update the annotation of

$n_j$ to denote the data dependency between $s_j$ and $s_\sigma$. The data dependency is formalized through the update of $n_j$'s annotation as follows:

$$(n_j, \{P\}) \rightarrow (n'_\sigma, \{P\})$$

The case that we cannot find a node $n'_\sigma$ that requires $P$ as input does not introduce incoherence in the orchestration. This only means that the unavailable service requires more inputs than the substitute service, which is accordance with the contravariance of arguments in the subtyping rules. Similarly, the covariance of results in the subtyping rules guarantees that $P$ is provided as output by the substitute service, and thus guarantees the existence of the node $n_\sigma$.

#### Integrating substitute service control flow

Starting from the checkpoint of the unavailable service (denoted $C$) at which the state transfer has been performed, we remove, from the orchestration workflow, all the nodes successors of $C$ that are included in the unavailable service behavior. Then, from $C$, we integrate the part of the control flow of the substitute service that has to be executed, in parallel with the workflows of the other services participating in the orchestration. This is performed using an AND-split workflow pattern. We link $C$ with an AND-split labelled transition, to the checkpoint of the substitute service (denoted $ckpt_\sigma$) of the substitute service, resulting from matching between substitute and unavailable services' behaviors.

#### Verifying deadlock freedom of the orchestration workflow

After updating the workflow of the orchestration, we have verify that the resuming the execution from the transformed workflow would not lead to deadlocks. A *deadlock* is a situation where the execution of a node is infinitely blocked without being able to reach a final node. This would occur when a node provides an output that is required as input by one of its predecessors in the orchestration workflow. If this situation is likely to happen, then a workflow transformation is required in order to free the orchestration from deadlocks. Several efforts have been performed in the domain of workflow transformation and verification [Aalst, 1997, Grossmann et al., 2006, Derbel et al., 2008, Verbeek, 2001]. To verify deadlock freedom after integrating of the substitute service workflow, we re-use the work proposed in [Grossmann et al., 2006].

Once deadlock freedom is verified, the execution can be resumed. While the orchestration execution is progressing, the dependency graph can be re-built, as a background task, according to the new structure of the orchestration workflow in order to anticipate in case of occurrence of another service unavailability.

## 7.3 Concluding Remarks

In this chapter, we figured out the impact of service substitution on the entities participating in the SOA system, particularly, the client and still-available services. We proposed the way to reconfigure the client and the service orchestration in order to resume the execution without introducing inconsistencies. The client reconfiguration consists of updating its interactions according to the substitute service behavior. The orchestration reconfiguration consists of (1) detecting the data dependencies between the services it composes, (2) reconfiguring the still-connected service according to the state synchronization of the substitute service, and (3) integrating the behavior of the substitute service in the orchestration workflow. The realization of our approach along with its experimental study are reported in Part III.

# Part III

# Realization

*A theory is something nobody believes, except the person who made it. An experiment is something everybody believes, except the person who made it.*

Albert, Einstein

# 8

# SIROCCO: ServIce Reconfiguration upOn serviCe unavailability and Connectivity lOss

As presented in Chapter 3, service provisioning can be interrupted as a result of a network disconnection (i.e., "in-middle" failure), and/or a service unavailability (i.e., "near-host" failure) such as service undeployment. In the case of "in-middle" failure, the service continuity can be ensured through network-based solutions, such as handoff or multi-homing. In the case of "near-host" failure, service reconfiguration is necessary to substitute the unavailable service with another one. Combined with network-based solutions [Rong et al., 2007a, Rong et al., 2007b], service reconfiguration can be also an alternative solution to "in-middle" failures when no network connection is available between a client and its service provider.

To this aim, we propose SIROCCO (ServIce Reconfiguration fOr serviCe unavailability and Connectivity lOss) middleware that (1) integrates existing network-based solutions to enable seamless mobility, (2) integrates existing solutions for service discovery and composition, and (3) enhances these functionalities with a support for runtime service reconfiguration in the case of service unavailability. Actually, the main contribution of SIROCCO lies in the support of the runtime service reconfiguration that enables service substitution. In Section 8.1, we present an overview of SIROCCO Architecture components, and provide the basic background on the existing middlewares that we use for handling network-based solutions, and service composition and discovery. We then present a collaboration scenario of the middleware components at runtime, in order to emphasize the role of each of them in runtime service reconfiguration. Furthemore, we detail and integrate into SIROCCO middleware the *SIROCCO service registry* (Section 8.2), and further enhance the middleware with a support for *runtime service management*. *Runtime service management* includes an execution engine (presented in Section 8.3), and our realization of the service reconfiguration. In Section 8.4, we concentrate on the details of our prototype that implements the runtime service reconfiguration, realizing the theoretical solution established in Chapters 4 to 7. We also assess the use of SIROCCO service reconfiguration through an implementation of our train ticket booking scenario, along with a set of experimental results. Finally, Section 8.5 provides the concluding remarks of this chapter.

## 8.1 Middleware Architecture Overview

As illustrated in Figure 8.1, the middleware architecture is layered on top of a legacy networked software platform, and decomposes into two main layers to enable the deployment and the use of distributed applications. Applications include a set of services that have been independently developed and deployed in certain sites over the network. They are invoked when required, to

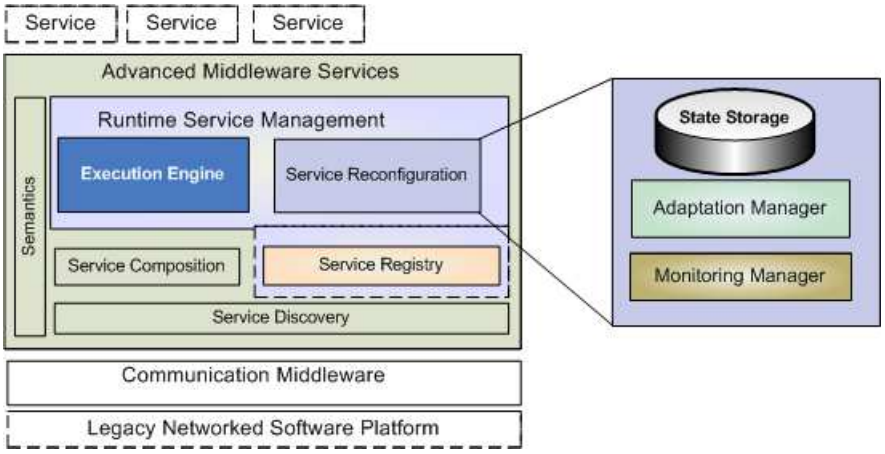



Figure 8.1: SIROCCO middleware architecture

respond to users' requests. A number of these services may potentially serve as candidate service substitutes for a service that becomes unavailable at runtime.

The lower middleware layer deals with service communication in the user environment. It offers the abstraction of an integrated multi-radio network, which comprehensively composes several networks in reach *via* the embedded radio interfaces (e.g., GPRS, WiFi, Bluetooth). As a result, the communication middleware offers the abstraction of an integrated multi-radio interface to the software services of the upper layer, in order to use the various networks in reach to communicate with the services and improve their availability.

The upper middleware layer embeds advanced services related to the consumption of the distributed resources.

- *Service discovery* enables to dynamically find the networked services in the changing networking environment.

- *Service registry* classifies services resulting from the service discovery into catalogs according the functionalities they offer. These services are further enhanced with semantic capabilities description to overcome the restrictions of the syntactic description conformance. Additionally, stateful services may provide a description of their state as well as their recovery capabilities in order to enable reliable service interactions. The *service registry* is periodically updated according to the service availability.

- *Service composition* enables the composition of networked services in order to provide sophisticated service functionalities that serve a user request.

- *Runtime service management* is responsible for correctly interacting with services. It includes an *execution engine* that follows the description of the service behavior in order to foster correct consumption of the service functionality. The *execution engine* is further in charge of coordinating the service orchestrations that result from the service composition.

We propose to enrich the advanced middleware functionalities with a *service reconfiguration* that enables to perform runtime service substitution in the case of "near-host" or "in-middle" failures. Service reconfiguration replaces the unavailable service with a similar service available in the client's networked environment, and synchronizes its state according to the interrupted execution using either state transfer or replay. More specifically, it includes:

- A *monitoring manager* that inspects the execution of the services involved in an interaction with the user, and notifies of service unavailability.
- An *adaptation manager* that dynamically reconfigures services, and potentially, orchestrations that are affected by a service unavailability.
- A *state storage and management* that proactively stores –when possible– the service states, and organizes them by date, so that to ease state selection, and thus, transfer to the substitute service.

At runtime, the *service registry* is involved in the *runtime service management* in order to be provided with a substitute service for a service that becomes unavailable. Hence, we also integrate the *service registry* (in a dashed box in Figure 8.1) to the *runtime service management*.

In the current realization of SIROCCO middleware, we integrate existing middlewares:

- PLASTIC middleware that realizes the communication middleware, and
- iCOCOA middleware that realizes semantic service discovery and composition.

In the following, we provide basic presentations of Plastic middleware 8.1.1 and iCOCOA 8.1.2. Then, we describe the interactions between the different modules that are included in SIROCCO middleware architecture 8.1.3.

### 8.1.1 PLASTIC Multi-radio Communication Middleware

B3G networks combine multiple wireless networking technologies in order to benefit from their respective advantages and specificities. In multi-networks environments, B3G-capable devices (e.g., laptops and PDAs) hold several radio interfaces, such as GPRS [1], WiFi and Bluetooth, and the possibility to switch from one radio interface to another, as illustrated in Figure 8.2. Switching from one radio interface to another increases the possibilities to connect two devices

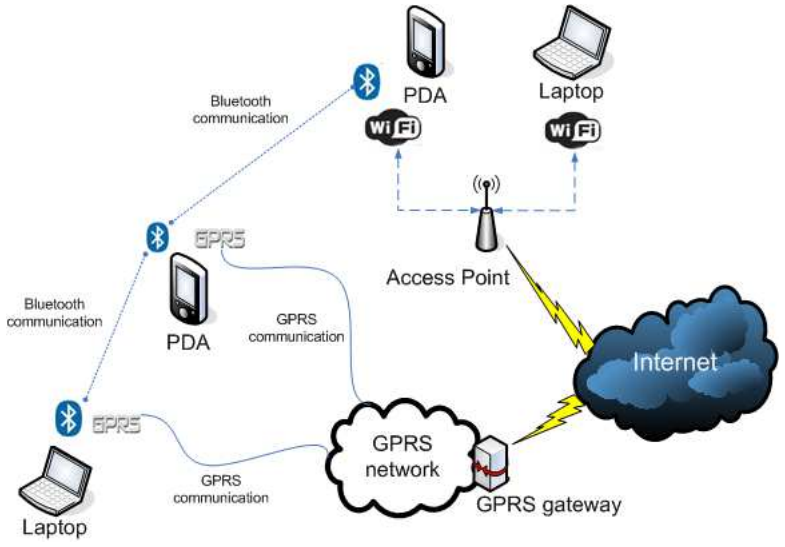



Figure 8.2: Multi-radio communication

using two or more network paths, and thus makes network disconnection recoverable in the case

[1] General Packet Radio Service (GPRS)

of existence of an alternative network connection. In addition, in the case of impossibility of reaching the same device, switching from one radio interface to another increases the perimeter of reachable service providers, and thus the possibilities to find a substitute service on reachable networks.

In this context, PLASTIC [2] (Providing Lightweight and Adaptable Service Technology for Information and Communication) is a platform that develops a design framework for mobile B3G services. PLASTIC-enabled devices then benefit from such a pervasive network by increasing the perimeter of reachable service providers. As presented in Figure 8.3, PLASTIC Middle-

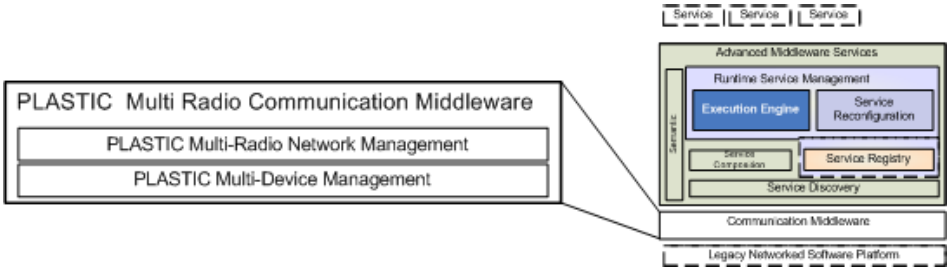



Figure 8.3: PLASTIC Multi-radio Communication Middleware

ware [Caporuscio et al., 2008] integrates a Multi-radio Communication Middleware, which copes with the complexity induced by the heterogeneity of the wireless technologies and makes it transparent to the users. It exploits B3G network abstraction by capturing the various networks and observing their status (e.g., connectivity and QoS). Multi-radio Communication Middleware is designed as a 2-layered architecture: (1) the lower layer integrates the PLASTIC Multi-Radio Device Management component, and (2) the higher layer integrates the PLASTIC Multi-Radio Network layer.

The Multi-Radio Device Management layer [Rong and Caporuscio, 2008] abstracts the B3G network to the upper layer. It manages the low-level characteristics of the perceived networks in terms of functionalities and QoS properties. It is in charge of:

1. Sensing the available networks and retrieving their characteristics (attributes and offered services),

2. Monitoring their status proactively, and

3. Accessing them to exploit the offered services.

The Multi-Radio Device Management layer can be utilized by the upper layers (e.g., Multi-Radio Network) in order to switch between different types of networks during network selection. It can be also exploited directly by the *runtime service management*, or even the application layer. It implements the Multi-radio Network Layer library, which is in charge of managing the entire communication between two devices, as well as the selection of the most appropriate underlying radio network through which carrying on the communication. It supports the following main functionalities:

1. IP address management according to the multiple network radio interfaces of the devices.

2. Radio interface activation and network selection with respect to the required QoS.

[2] IST FP6 STREP Plastic: http://www-c.inria.fr/plastic/the-plastic-middleware

3. Communication facilities, including synchronous unicast [3], and asynchronous multicast [4].

### 8.1.2 iCOCOA Service Discovery and Composition

iCOCOA [Mokhtar et al., 2008a] is a distributed system middleware, particularly targeting open, decentralized, dynamic computing environments, realized by mobile computing systems, in particular, pervasive or ambient intelligence systems, or the Web itself. iCOCOA has been developed and tested in the context of ambient intelligence for the home environment [5] and Systems of Systems (SoS). iCOCOA extends base Web Services middleware by featuring awareness of service semantics, besides plain syntactic service descriptions. As presented in Figure 8.4, iCOCOA

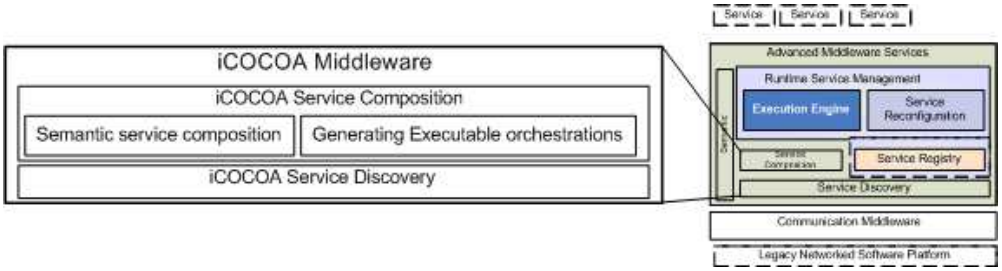



Figure 8.4: iCOCOA service discovery and composition

supports semantic service discovery and composition.

In more detail, iCOCOA supports the three following main functionalities:

1. **Semantic service discovery**
   iCOCOA offers an API [6] that allows user applications to dynamically discover semantic services by specifying user requested capabilities. Based on this specification, services can be discovered, providing the users with the required service capabilities. iCOCOA supports two semantic relationships between the semantic concepts of the required and provided capabilities, namely, equivalence and subsumption. The discovery performs a semantically match of the requested capabilities with those of the networked services based on service functional and non-functional properties.

2. **Semantic service composition**
   Based on the set of discovered services, iCOCOA attempts to weave a workflow of requested capabilities (without binding information) from the behaviors of the services. iCOCOA tries to compose networked services to fit the workflow specification. It performs a matching between the workflow of requested capabilities and services' descriptions. *iCOCOA service composition* is particularly flexible by featuring partial integration of service behaviors and service behaviors interleaving, while at the same time taking care that services are correctly consumed.

3. **Generating executable orchestrations**
   Once an abstract workflow is composed, iCOCOA generates a WS-BPEL executable de-

[3] Synchronous unicast is used to read/write packets to be exchanged during the interaction between client and server user applications

[4] Asynchronous multicast allows user applications to send multicast packets to the members of a given group

[5] Project FP6 IST Amigo, http://www.hitech-projects.com/euprojects/amigo

[6] Application Programming Interface

scription of the composition workflow, which includes bindings to the selected networked services.

### 8.1.3 Execution Life Cycle

We present in this section the inter- and intra-layer collaborations of SIROCCO middleware. These collaborations are performed through message passing between the different modules embedded in the SIROCCO middleware, as modeled in the sequence diagram in Figure 8.5.

SIROCCO middleware functionalities are triggered when receiving a user requested capability, which is first processed by *SIROCCO service registry*. *SIROCCO service registry* calls *iCOCOA service discovery* in order to retrieve all the networked services that semantically match with the required functionality. In the case of multiple radio networks, the *service discovery* switches from one radio interface to another to find all network reachable services that may serve the user request. It then provides the *service registry* with the descriptions of the available WS.

In order to classify the service descriptions into catalogs, the *service registry* calls the *service reconfiguration* in order to check the compatibility of the service descriptions with the user requested capability. In the case of service orchestration, the *service registry* sends to *iCOCOA service composition* the firstly listed service description of each required capability in order to compose the workflow of capabilities and generate an executable BPEL. In the case of a single service consumption, the last step is omitted. In both cases, the BPEL process is sent to the *service reconfiguration*, along with the description of the checkpoint positions.

The *service reconfiguration* performs the necessary transformations on the BPEL process in order to enhance it with *checkpoint management*, *fault handling*, *state access and management* and *dynamic bindings*. It also creates the associated aFSA and data dependency graph. Then, the *service reconfiguration* sends the process to the execution engine in order to be executed. Typically, at this stage, the BPEL execution engine instantiates the set of services defined as partner links in the process and starts the process execution. During the interaction, the *service reconfiguration* is involved in order to store the state of the services that are interacting with the BPEL process.

When a service becomes unavailable the Execution Engine receives a fault message from the Plastic Multi-Radio Communication layer. It then triggers a reconfiguration process in the *service reconfiguration*, which is set as follows.

The *service reconfiguration* tries first to switch from the current radio-interface to another one to reach the same service instance. If the service provider device is reachable on another network, then the execution is resumed without a particular service reconfiguration. In the case that the device is not found, then the *service reconfiguration* proceeds to a service substitution, which is performed through the sequence of messages: 24 till 35, in Figure 8.5.

The *service reconfiguration* first calls the *service registry* for the catalog related to the capability of the unavailable service. The *service registry* performs the last updates of the catalog using the service descriptions resulting from the *service discovery*, and provides the *service reconfiguration* with the list of service descriptions. The *service reconfiguration* computes or updates the compatibility degree of the services in the catalog and selects the service that is best compatible with the substituted service. It potentially performs the necessary syntactic mappings and transfers the state of the substituted service to the selected one in order to synchronize its state accordingly. The *service reconfiguration* transforms the aFSA in the case that the substitute service defines a behavior different from the substituted service.

In the case of service orchestration, the *service reconfiguration* checks the data dependencies according to the checkpoint at which the state transfer has been performed, computes a recovery

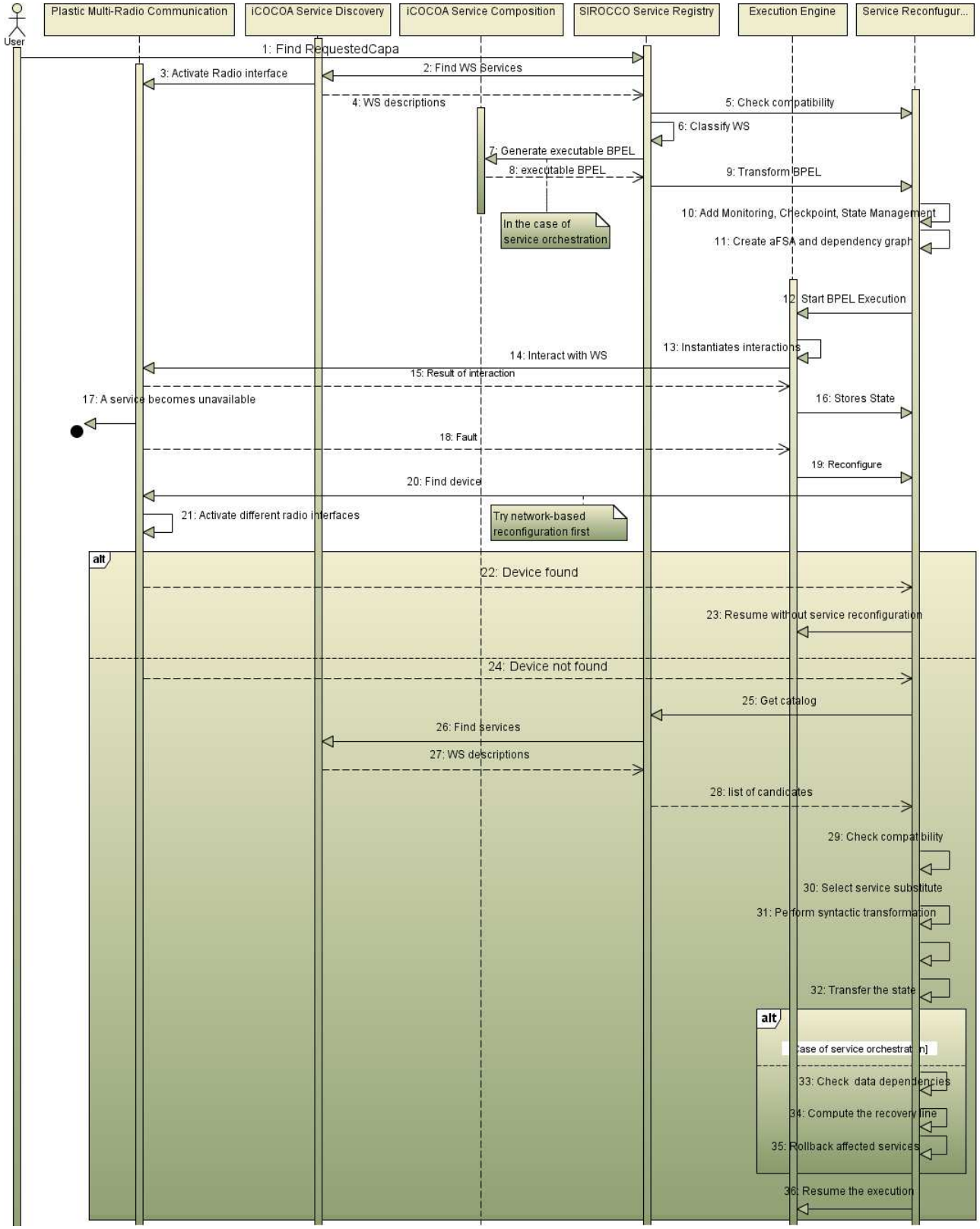



Figure 8.5: Sequence diagram of the collaboration between SIROCCO components

line and rolls back the services that are affected by the substitution. Finally, the execution can be resumed with the substitute service.

## 8.2 SIROCCO Service Registry

Starting from a set of user requested capabilities, *SIROCCO service registry* stores the list of services that can be used to serve a user request, and which are resulting from the service discovery.

### 8.2.1 Architecture

The *service registry* maintains a set of service catalogs. Each catalog corresponds to a different semantic category of services, and therefore it is characterized by an OWL semantic class. The particular ontology that characterizes a catalog is retrieved from the semantic concept of the functional purpose of the user requested capability. Each service catalog is progressively populated (during the lifetime of the orchestration execution) with service descriptions of the available networked services. The organization into catalogs eases the selection of the service substitute upon a service unavailability.

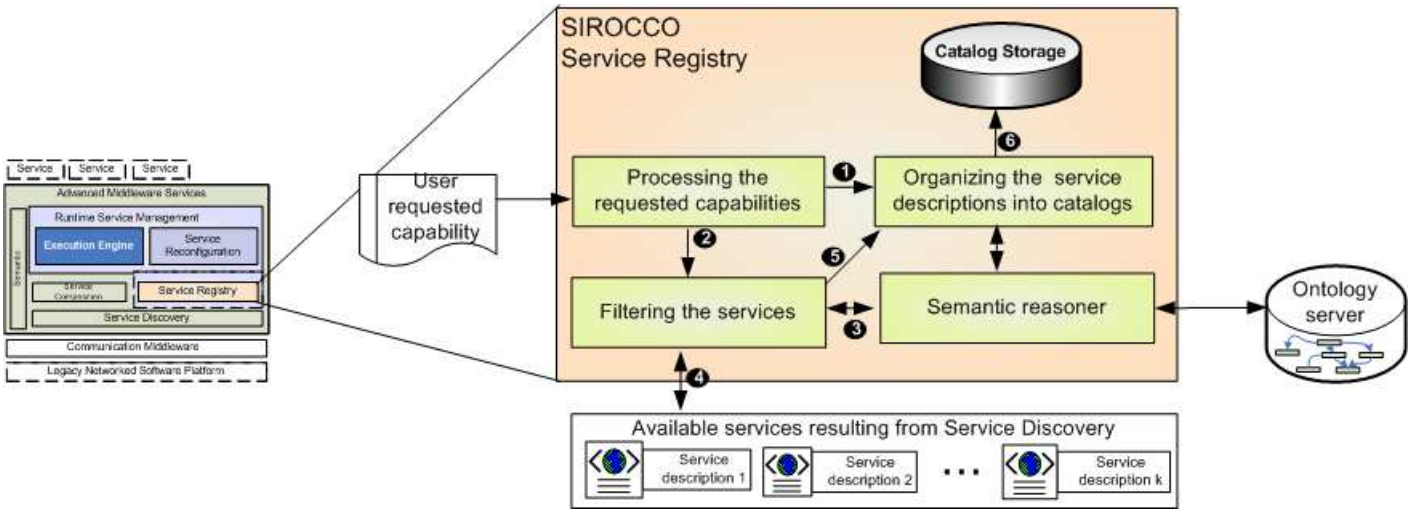



Figure 8.6: SIROCCO service registry

As illustrated in Figure 8.6, *SIROCCO service registry* includes four functional modules, and a storage support for service descriptions, namely:

1. a module for "processing user requested capabilities",
2. a module for "filtering available services" module,
3. a module for "organizing the services",
4. a "semantic reasoner",and
5. a support for "storing service descriptions".

To describe the functioning of each module, we go through the processing life cycle of a given user requested capability, and detail the role of each module.

### 8.2.2 Prototype Implementation

Let $RequiredCapa = (C_{RequiredFunc}, C_{ProvidedInputs}, C_{RequiredResults})$ be one of the user requested capabilities. To proceed to service classification, *SIROCCO service registry* first processes a user requested capability (as illustrated in Figure 8.6). For each requested capability referencing a semantic concept, $C_{RequiredFunc}$, it first checks whether a related catalog is created, or not. If not, it creates a new catalog using the *organizing module* (Step ❶ in Figure 8.6). It then selects the networked services that provide a capability with reference to a semantic concept $C_j$ such that $C_{RequiredFunc} \subseteq^s C_j$. The *filtering module* retrieves the service descriptions resulting from iCOCOA service discovery (Step ❷ in Figure 8.6). As iCOCOA supports only equivalence and subsumption semantic relationships, *SIROCCO service registry* includes a *semantic reasoner* module that retrieves the semantic concepts in a given ontology, which semantically include $C_{RequiredFunc}$ (Step ❸ in Figure 8.6). Then, the *filtering module* uses the set of semantic concepts provided by the *semantic reasoner* in order to make requests to iCOCOA service discovery (Step ❹ in Figure 8.6). In particular, it invokes iCOCOA supported methods, as described in the listing of Figure 8.7.

```
   /* The repository sets the type of matching as follows:                                    */
1  SIROCCOSereReg.setMatcher("exact");
   /* Defining a container to include the available services                                  */
   vector < String[] > services = new vector < String[] > ();
   /* Retrieving services that provide capabilities with a reference to a semantic concept that semantically includes
   the one of the required capability                                                        */
3  foreach semantic concept C_k retrieved by the semantic reasoner module do
4      String[] ser= SIROCCOSerReg.getServicesWithType("http : //target_ontology/MyTypes.owl#C_k");
       services.add (ser);
```

Figure 8.7: Retrieving services from the service discovery

In the listing of Figure 8.7, $SIROCCOSerReg$ represents the instance of iCOCOA service discovery. We first set the matching parameter to "exact" using the method $setMatcher$. This enables to match concepts that are identical, but also concepts that are either explicitly equivalent [7], or implicitly equivalent [8]. We then retrieve all the matching services using the method $getServicesWithType$, which takes as argument the semantic concept of the functional purpose of the capability, and returns a list of service names corresponding to the given semantic type. Using the *semantic reasoner*, the *filtering module* then checks for each retrieved service provided by the *service discovery*, whether the semantic concepts of its inputs are semantically included in the ones of the user provided inputs, and inversely for the semantic concepts of the outputs and the required results. The set of services that comply with the required capability is passed to the *organizing module* in order to be classified into catalogs.

The *organizing module* first locates the catalog related the requested capability ($C_{RequiredFunc}$). Then, for each service providing a capability $Capa_j$, it asks the upper layer (i.e., the *adaptation manager*) to compute the compatibility degrees with respect to the signature $CD_{Signature}$, and the pre- and post-conditions $CD_{Pre-Post}$. If one of them is null, the service is not inserted in the catalog, otherwise, the *organizing module* inserts the service description in the related catalog with respect to its compatibility degree, where the catalog is sorted in decreasing order of compatibility degrees.

[7] Tagged with the $owl : equivalentClass$ relationship.

[8] It has similar properties of another class tagged with the $owl : equivalentProperty$ relationship

To start the execution, the *runtime service management* selects the first listed service in the catalog, in order to perform potential syntactic transformations of the user inputs. In the case of service orchestration, the behavior descriptions of the selected services are passed to iCOCOA Service Composition module in order to generate an executable BPEL process. Once a service $s_k$ is selected, the other services of the same catalog are reorganized with respect to their compatibility with the functionality of the service $s_k$, in order to proactively prepare the service candidates for substituting $s_k$. As the compatibility degree takes into account the semantic and syntactic matching between signatures, if there are service candidates that have the same service descriptions as the selected one, they will be listed on the top of the list in the catalog of $RequiredCapa$.

In the case that the selected service is stateful, the *organizing module* divides the $C_{RequiredFunc}$'s catalog into two categories. The first category contains descriptions of services that are provided with WS-Resource Properties descriptions, while the second category contains all the other services with matching capabilities only. If the unavailable service is not accompanied with a WS-Resource Properties description, the first category of services is empty.

## 8.3 Execution Engine

The *runtime service management* is in charge of correctly invoking services in order to consume the service provided functionalities. It includes an Execution Engine that is responsible for correctly handling the interactions between the client and the services in order to support service provisioning. The service provisioning is enhanced with reliability through the use of the *service reconfiguration*. In this section, we focus on the execution engine that is integrated in SIROCCO middleware.

### 8.3.1 Architecture

As presented in Chapter 2, we describe service behaviors using BPEL processes. Tremendous BPEL execution engines are candidates to be integrated in SIROCCO. For maintainability reasons, we essentially focus on the ones that are *free* and *open source*. These execution engines are: ActiveBPEL Community Edition [Act, 2009], BEXEE (BPEL Execution Engine) [Dubuis et al., 2004], PXE (BPEL Process eXecution Engine) [PXE, 2005], Sliver (A SOAP and BPEL execution engine for mobile devices) [Hackmann, 2006], Orchestra [Orc, 2005], and ODE (Orchestration Director Engine) [ODE, 2006], which are all implemented in Java. ActiveBPEL has been lately commercialized, and the community edition lacks several functionalities. Since 2004, BEXEE has no longer been actively developed. PXE lacks documentation. Sliver is dedicated for mobile devices, which can be used when migrating our approach on lightweight devices. This migration is one of our future work directions. Also, only a part of listed execution engines are provided with a visual designer for BPEL processes, which are ODE and Orchestra. We draw up a summary about a set of characteristics of these engines in Appendix A, where ODE and Orchestra can be equally used. In SIROCCO, we integrate Apache ODE as execution engine.

The ODE BPEL execution engine cannot interact by itself with the outside world. For this it relies on an "integration layer" that provides it with communication channels for the runtime interactions, which can be AXIS 2 libraries [9] or JBI (Java Business Integration) message bus. As we are interested in Web services interactions, we choose AXIS 2 as a container instead of JBI.

[9] http://ws.apache.org/axis2/

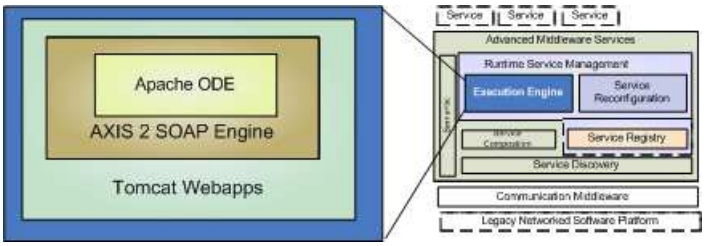



Figure 8.8: Execution engine

The fundamental function of the AXIS 2 libraries is to allow the execution engine to communicate via Web Service interactions. Hence, in our realization, we deploy ODE in AXIS 2 Web services container.

Apache Axis 2 Web services engine requires itself a servlet engine that enable to deploy web application. In our realization, we use Apache Tomcat [10] as a servlet engine. Thus, AXIS 2 is deployed as a Web application in the Tomcat container, and ODE is deployed as a Web application in AXIS 2 container.

Still, ODE and all the existing BPEL execution engines lacks flexibility and several functionalities that are required to perform service reconfiguration. For instance, none of the BPEL execution engines enable to change the BPEL process activities at runtime. Whilst, this functionality is required to assess the feasibility of one of our main contributions, which consists in enabling the substitute service to implement a different behavior from the one of the substituted service. Other shortage of the existing execution engines are detailed in the following section. To overcome this shortage and lack of flexibility, we implemented our prototype for BPEL execution engine: SIROCCO Execution Engine.

### 8.3.2 Prototype Implementation

Hereafter, we briefly present the functionalities supported by SIROCCO Execution Engine.

#### SIROCCO Execution Engine

The objective behind SIROCCO Execution Engine implementation is not to introduce another tool in the existing tool set, but to study the feasibility of our dynamic reconfiguration approach. Once this feasibility is assessed, we will migrate our solution on ODE in order to be used outside our research sphere.

SIROCCO Execution engine takes as input the aFSA that represents the transformation of BPEL processes. It offers the basic, and yet necessary, functionalities that enable (1) to suspend an execution when a service becomes unavailable, and (2) to resume the execution with a transformed aFSA. The main difference between SIROCCO and existing execution engines lies in the runtime management of the BPEL process: SIROCCO Execution engine uses the graph traversal technique to execute a BPEL process, while the others use a pre-compiled process that pre-fixes all its interaction dependencies with the external entities (i.e., WS). Nevertheless, SIROCCO does not support full-featured BPEL orchestrations e.g., pick activities, switch and wait activities, are not supported.

At the current stage of implementation, we migrated a set of functionalities of the *runtime service management* to ODE execution engine, while others that require to enrich ODE with

[10] http://tomcat.apache.org/

further functionalities, and thus are still tested only on SIROCCO Execution Engine. In order to test the functionalities that have been migrated to ODE, we realized a set of stateful Web services using Globus toolkit [gt4, 2005].

### Realizing Stateful Web Services

Globus toolkit 4 (GT4) [gt4, 2005] is the most mature, well documented tool that enable to realize and deploy stateful Web services. It includes "Java WS Core" runtime component that provides APIs and tools for developing WSRF services and offers a run-time environment capable of hosting them.

GT4 offers a set of libraries for stateful Web services, so as they can be easily deployed in the Globus Web applications container, which integrates AXIS 1 SOAP engine. However, the integration of the Globus application layer on top of SIROCCO middleware raised several issues. Indeed as ODE is based on AXIS 2 libraries, and Globus is based on a prior version of the SOAP engine (i.e., AXIS 1), which induces SOAP header heterogeneity.

Hereafter, we detail our solution to resolve software integration issues.

### Software integration issues: ODE and GLOBUS

The issue resides in enabling the interaction between ODE processes and Globus stateful services, as illustrated in ❶ Figure 8.9. On the one hand, ODE is able to invoke services using AXIS 2

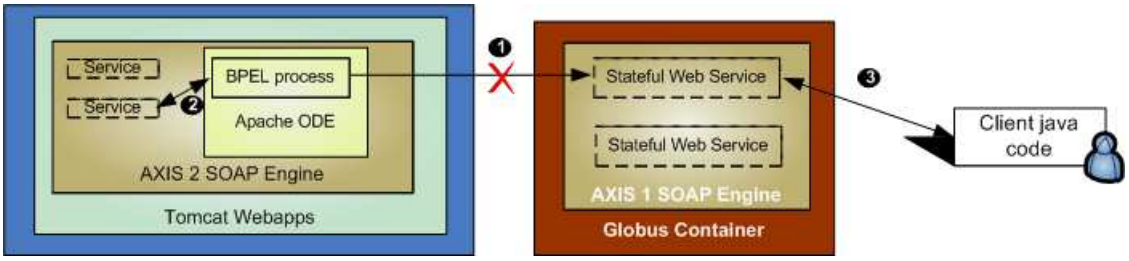



Figure 8.9: ODE and Globus integration issues

SOAP engine (❷ in Figure 8.9), and on the other hand, Globus Web service can be invoked from clients using AXIS 1 libraries (❸ in Figure 8.9).

The solution that comes naturally to overcome the integration issue, consists in creating a "proxy" service that has the code of the client for Globus Web services, and which is deployed in AXIS 2 Web service container in order to be invoked by ODE. As illustrated in Figure 8.10, this includes to integrate AXIS 1 libraries in addition to the ones of AXIS 2, in order to enable the "proxy" service to interact with the related Globus Web service. To realize this solution, we integrate an extra layer in the middleware: Software Integration Layer, which is set between the *runtime service management* and the application layer that provides stateful services. In the Software Integration layer, we implement the following functionalities.

First, we generate the Java code of the "proxy" service automatically from the code of the Globus Web service and its WSDL description (❶ in Figure 8.10). The sources of the service code generator are provided in Appendix B (Section B.4). We then compile the "proxy" service and generate its WSDL description using AXIS 2 libraries (❷ in Figure 8.10). We add the "partnerlinkType" definition to the generated WSDL so as to enable the BPEL execution engine to recognize it, and invoke it (❸ in Figure 8.10). The "partnerlinkType" is added using XSL

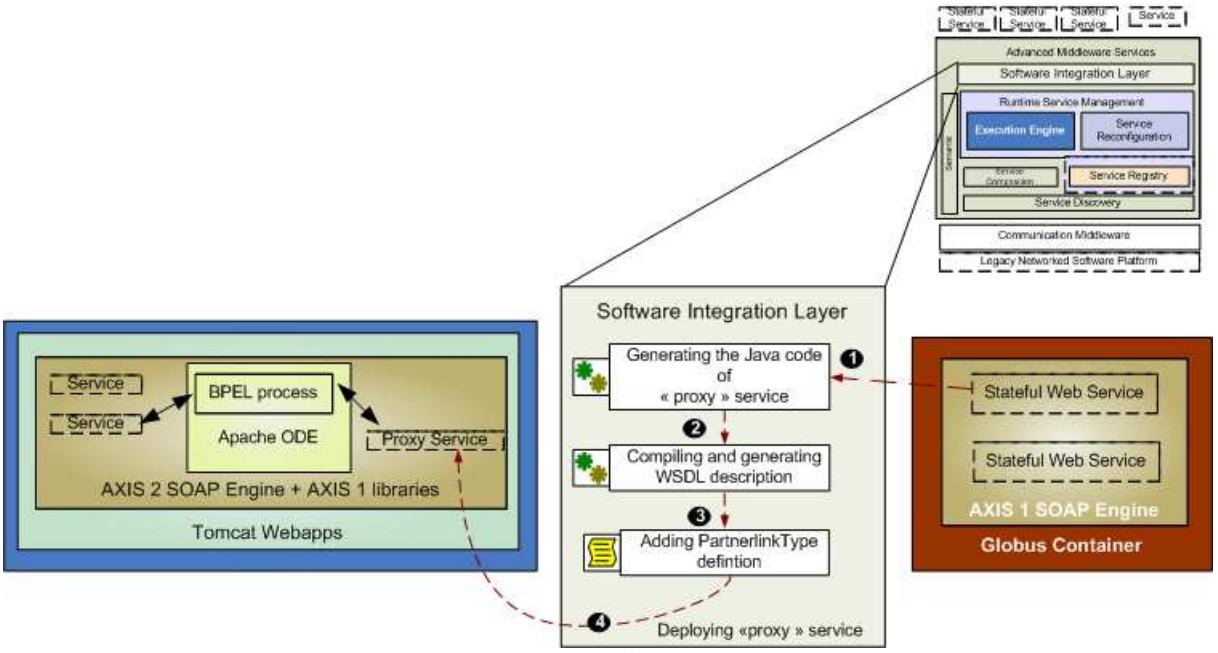



Figure 8.10: Overcoming integration issues

Transformations, the code of which is provided in Appendix B (Section B.5). Finally, we deploy the "proxy" service in AXIS 2 Web applications container (❹ in Figure 8.10).

Besides the BPEL execution engine, the *runtime service management* integrates a *service reconfiguration* that implements our solution for runtime service substitution. The next section details the architectural and technical details of the service reconfiguration.

## 8.4 Service Reconfiguration

In the following section, we first present an architectural overview of the *service reconfiguration*, and then present the details of our prototype implementation.

### 8.4.1 Architecture

The *service reconfiguration* is divided into three layers, namely, a *monitoring manager*, an *adaptation manager* and a *state management* modules, as illustrated in Figure 8.11.

#### Monitoring manager

The lowest layer includes a *monitoring manager* which is in charge of detecting the service unavailability. The objective of the *monitoring manager* is to make the Execution Engine aware of a service failure, in order to call the *adaptation manager* and reconfigure the failed execution. To this aim, a possible solution consists in making the execution engine retrieving the errors generated by the lower layers (i.e., AXIS 2 SOAP engine).

It consists of capturing AXIS 2 errors, and alerting the execution engine of the failure in order to call the *adaptation manager*. The current version of ODE supports this solution that we integrate in SIROCCO using the notion of *activity failure and recovery*, which are detailed in the following.

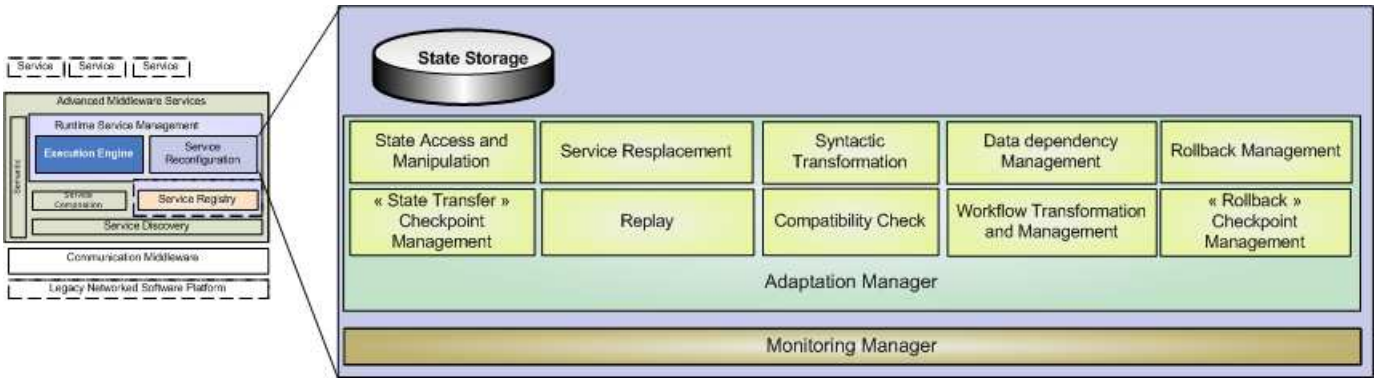



Figure 8.11: Service reconfiguration

- *Activity failure*

  In BPEL specification terminology, a service returns a *fault* in response to a request it cannot process. A process may also raise a fault internally when it encounters a terminal error condition, e.g., a faulty expression or false join condition. failures are non-terminal error conditions that do not affect the normal flow of the process. ODE enables to keep the process definition simple and straightforward by delegating failure handling to the execution engine. For example, when the process is unable to determine the service endpoint, it generates a failure. Then, the process would either terminate or require fault handling and recovery logic to proceed past this point of failure.

- *From failure to recovery*

  A failure condition is triggered by AXIS 2 SOAP engine, in the place of a response or fault message . The $< invoke >$ activity that originated the fault consults its failure handling and decides how to respond. In order to make the activity throwing a fault on failure, we set the attribute *faultOnFailure* to *true*, as described in the listing of Figure 8.12.

$$< ext : failureHandling xmlns : ext ='' http : //ode.apache.org/activityRecovery'' >$$
$$< ext : faultOnFailure > true < /ext : faultOnFailure >$$
$$< /ext : failureHandling >$$

Figure 8.12: Activity failure and recovery in ODE

The activity that is enriched with the above attribute will throw the *activityFailure* fault. In the case that the activity does not specify failure handling using this extensibility element, it inherits from the failure handling policy of its parent activity, recursively up to the top-level activity of the process. In this way, we use inheritance to specify the failure handling policy of all the activities in the process, using a single *failureHandling* extensibility element. Hence, we add the script listed in Figure 8.12 in the root activity (i.e., after the first $< sequence >$ activity) of the BPEL process, so as all the child-activities can inherit from it.

In practice, the *monitoring manager* is provided with an XSLT script that adds to the BPEL process the "activity failure" option as well as the definition of the required *namespace* to import. The XSLT transformation is listed in Appendix B, Section B.1.

### Adaptation manager

The *adaptation manager* is the main module of the *service reconfiguration*. As illustrated in Figure 8.11, the *adaptation manager* integrates a set of modules, each of them has a specific role in performing runtime service reconfiguration.

As illustrated in Figure 8.13, these modules are:

1. *Checkpoints management*, which is split into 2 modules: the one is responsible for *rollback* checkpoints and, the other for *state transfer* ones.

2. *State access and manipulation*, which is responsible for communicating with the upper *state storage* module.

3. *Service replacement*, which is responsible for changing the service endpoint in order to interact with the selected service in the place of the service that becomes unavailable.

4. *Compatibility check*, which is responsible for checking the compatibility between two service descriptions, computing the compatibility degree between them, and selecting the service that is best compatible with the substituted service. At state transfer, it is also in charge of matching between state descriptions and verifying the compliance with the pre-, post-conditions, invariants and constraints. As a result, it provides (if exists) the checkpoint in the substitute service workflow at which the state transfer should be performed.

5. *Syntactic transformation*, which is responsible for applying syntactic mapping between XML elements.

6. *Replay*, which is responsible for executing a part of the substitute behavior seamlessly to the client when state transfer is not possible.

7. *Workflow transformation and management*, which is responsible for transforming a BPEL process into an aFSA in order to reason on it. It is also responsible for updating the client interactions and the orchestration workflow according to the substitute service workflow.

8. *Data dependency management*, which is responsible for creating and maintaining consistent data dependency graph.

9. *Rollback management*, which is responsible for synchronizing state of the orchestration according to the recovery line.

### State storage

First, as the networked environment may include infrastructure-less communications, service reconfiguration cannot rely on the service provider to seamlessly transfer the state to its substitute. Therefore, the service state has to be stored at, and transferred by, the client middleware.

The *state storage* is included in SIROCCO to proactively store –when possible– the service states, and sort them by date order (using their timestamp and the service end point at which they have been checkpointed) in order to provide the substitute service with the last state stored of the unavailable service.

Previous states (to the last one) of the unavailable service can be useful in the case that the substitute service cannot synchronize with the last state but a prior one. The *state storage* module is also in charge of logging the messages exchanged between the client and the executing services in order to enable the replay with a substitute service.

### 8.4.2 Prototype Implementation

In our prototype, the main complexity resides in the implementation of the adaptation manager. However, as explained in Section 8.3, ODE is not flexible enough to support all the functionalities required by the adaptation manager modules. Hence, as illustrated in Figure 8.13, we split the *adaptation manager*'s modules into three sets, where:

- Three modules do not require special functionalities from the execution engine,and thus, have been migrated from *SIROCCO execution engine* to ODE. These modules are the *state transfer checkpoint management*, the *state access and management*, and the *service replacement* modules.

- Four modules require the ability to change the BPEL process structure at runtime. The change of the process structure at runtime is supported only by *SIROCCO execution engine*. These modules are the *workflow transformation and management*, the *data dependency management*, the *rollback checkpoint management*, and the *rollback management*.

- The other modules are still at design stage including *compatibility check*, *syntactic transformation* and *replay* modules.

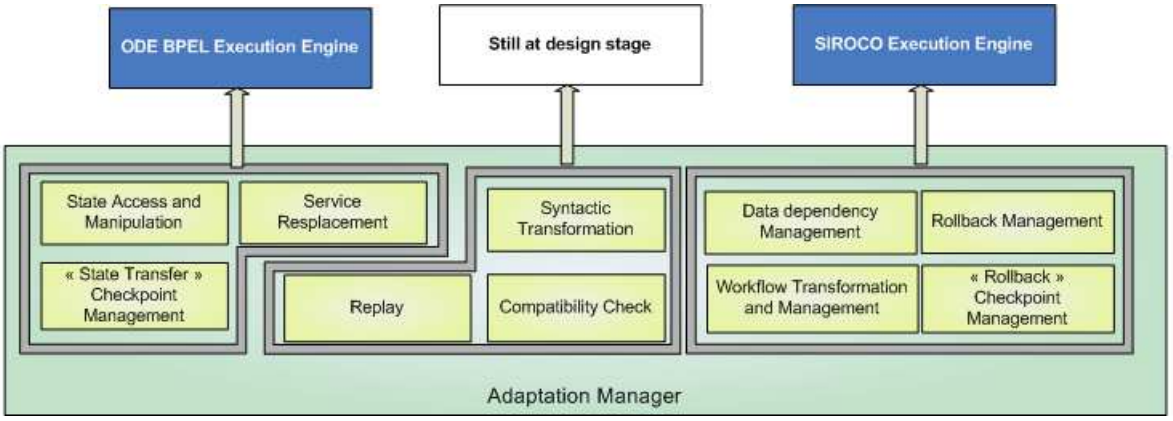



Figure 8.13: Adaptation manager

#### Checkpoints management

SIROCCO enables to choose between executing a BPEL process with, or without any, reconfiguration handling. In both cases, service developers provide a description of the service behavior using a BPEL process. For services that are enhanced with checkpointing and recovery operations, the service developer has to provide an extra description that points out the checkpoints positions in the BPEL process. The checkpoints positions are provided as a set of XPATH [11][W3C, 2007d] expressions related to the BPEL process. The XPATH expressions enable to select set of BPEL activities in a BPEL process that are included within a checkpoint interval, in order to be encapsulated in a *scope*.

[11]XPATH is a W3C recommendation, which is a syntax for defining parts of an XML document. It uses path expressions to navigate in XML documents.

**Scope activity in BPEL** WS-BPEL allows scopes to delimit a part of the process that is meant to be reversible in a process by specifying compensation or fault handlers. In our realization, we use the < *scope* > activity to delimit checkpoint intervals. The checkpoints can be either *rollback* or *state transfer*. The integration of extra < *scope* > does interfere with the BPEL process logic. Even though the BPEL process define a number of < *scope* > activities, these activities can be nested in our checkpoint intervals without any impact on the process logic the context of a scope is affected only by the execution of its enclosed activities.

Note that, following BPEL specification, the scopes in a process cannot interleave, they can be only nested. To deal with this limitation of BPEL specification, we assume that any *state transfer* checkpoint is also a rollback checkpoint, while the opposite is not necessary true. This assumption is realistic enough to be supported by real world services. Indeed, services that enable to synchronize their state at a given point of their behavior according to a given, can naturally synchronize their state according to one of their own previously-reached states. The scopes that correspond to *rollback* checkpoint intervals are then nested within the scopes that correspond to *state transfer* ones. as illustrated in Figure 8.14.

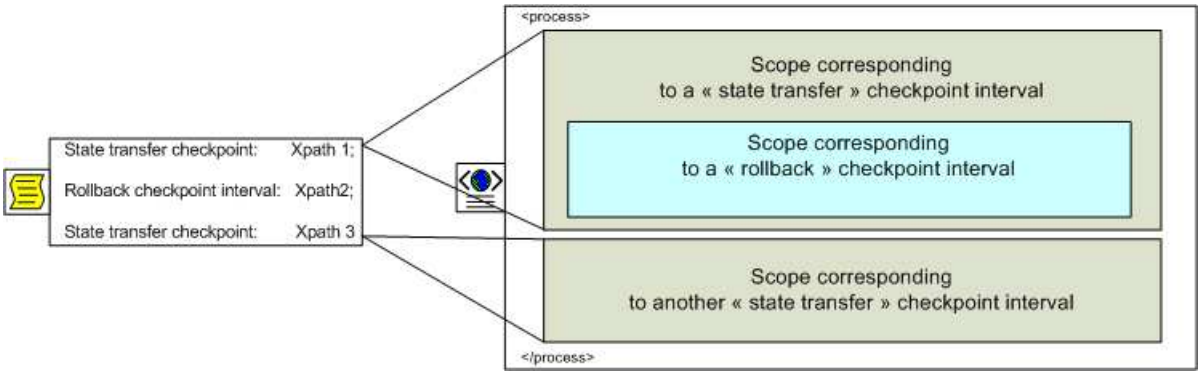



Figure 8.14: Checkpoints integration in the BPEL process

Once the checkpoint intervals are delimited, we need to enhance the BPEL process with checkpoints. The notion of checkpoint is not explicitly supported by BPEL. Nevertheless, BPEL enables the definition of fault handling activities, which we use to perform the required activities associated with a checkpoint.

**Adding checkpoints in BPEL** As defined in the previous chapters, Also, at these checkpoints, the client middleware can ask a substitute service for synchronizing its state according to a given state. the *state transfer checkpoint management* module enriches the scopes with the fault handlers that enable to encapsulate the reconfiguration strategy. At each *state transfer* checkpoint, a fault handler is added, which catches all faults that may raise in order to perform the reconfiguration strategy. The listing in Figure 8.15 describes the structure added in the scope corresponding to each *state transfer* checkpoint interval.

Note that the BPEL specification does not enable fault handler to point out activities outside its associated < *scope* >. This limits the extent of a rollback to a single scope. This limitation does not enable to perform a rollback propagation on several checkpoint intervals, and in the case of service composition, we cannot perform a rollback on several services.

These limitations lead us to implement rollback functionalities of SIROCCO middleware independently of the existing BPEL execution engines and, test them on our SIROCCO Execution Engine. SIROCCO Execution Engine does not use the notion of scopes for rollback, but it is

```
< scope >
  < faultHandlers >
    < catchAll >
      < sequence >
      ...
      < /sequence >
    < /catchAll >
  < /faultHandlers >
  ...
< scope >
```

Figure 8.15: Fault handler support in BPEL processes

capable to interpret the description of the *rollback* checkpoint in the aFSA.

### State access and manipulation

At the end of each scope, the client middleware is enabled to ask the services that provide recovery operations for their state, and store it in the *state storage*. The state is queried from the service using *GetState* recovery operation. When a service becomes unavailable, the client middleware retrieves the last state stored of the substituted service, and sends a *SetState* request to the substitute service in order be able to synchronize its state according to the transferred state.

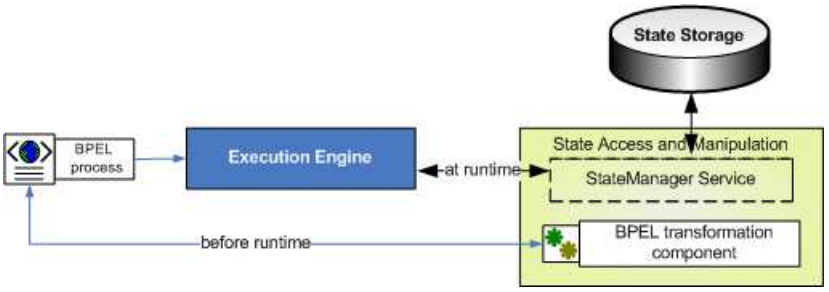



Figure 8.16: State access and manipulation module

The *state access and manipulation* module includes two entities:

1. A *BPEL transformation* component that acts before runtime by enriching the BPEL process with a number of activities, enabling state access and manipulation. It adds $< invoke >$ activities of the *GetState* and *SetState* operations of the services that are involved in the BPEL process.

2. A *state manager* Web service acts at runtime by storing the service state in the *state storage* and supplying the BPEL process with the service states, when required. To this aim, it implements two operation, namely, *StoreState* and *SupplyState* that can be invoked by the BPEL process.

The listing in Figure 8.17 describes the XSL transformations that are performed at the end of each scope. In order to enable the state storage, the *BPEL transformation* component adds an $< invoke >$ activity to the *GetState* operation of the service, to which the scope is associated (Line 8). It also adds an $< invoke >$ activity to the *state manager* service in order to store the last activity returned state in the *state storage* (Line 11).

```
1  < repeatUntil xmlns =" http : //docs.oasis − open.org/wsbpel/2.0/process/executable" >
     < scope >
      < faultHandlers > ... < /faultHandlers >
      ....
      < xsl : call − template name =" add_comment" >
       < xsl : with−param name =" valueComment" > Save the state of the service < /xsl : with−param >
      < /xsl : call − template >
8     < invoke name =" @partnerLink_getState" partnerLink =" @partnerLink"
        portType =" @portType" operation =" getState"
        outputVariable =" @partnerLink_getState_out"/ >
11    < invoke name =" Invoke_StoreState" partnerLink =" StateManager"
        portType =" nsStateManeger : StateManegerServicePortType" operation =" StoreState"
        inputVariable =" @partnerLink_getState_out"/ >
14    < assign >
       < copy >
         < from > true() < /from >
         < to > $scope_done < /to >
       < /copy >
19    < /assign >
     < /scope >
     < condition > $done_invoke < /condition >
22 < /repeatUntil >
23 < assign xmlns =" http : //docs.oasis − open.org/wsbpel/2.0/process/executable" >
     < copy >
      < from > false() < /from >
      < to > $scope_done < /to >
     < /copy >
28 < /assign >
```

Figure 8.17: First set of BPEL transformations performed by the state access and manipulation module

The *BPEL transformation* component also sets a flag (*scope_done*) to true, denoting that the scope has been executed without faults, and that the last state is stored (Line 14). If a fault occurs right after the flag setting, the scope will not be re-executed. Otherwise, the scope will be executed again with the new service. The re-execution is ensured using a loop that includes the entire scope and a guarded condition that enables to terminate the loop execution, only when the value of *scope_done* is true (from Line 1 to Line 22). After each loop, the *scope_done* value is set to false before starting the next scope (Line 23).

```
1  < faultHandlers >
     < catchAll >
     ...
       < assign >
         < copy >
          < from > true() < /from >
          < to > $update_state < /to >
          < /copy >
       < /assign >
     ...
     < /catchAll >
12 < /faultHandlers >
   < sequence >
     < if >
       < condition > $update_state < /condition >
       < xsl : call − template name =" add_comment" >
         < xsl : with − param name =" valueComment" >
   Transfer the state of the old service to the new one < /xsl : with − param >
       < /xsl : call − template >
       < sequence >
20       < invoke name =" Invoke_storeState" partnerLink =" StateManager"
           portType =" nsStateManeger : StateManegerServicePortType" operation =" SupplyState"
           inputVariable =" @partnerLink"
           outputVariable =" @partnerLink_SupplyeState_out/ >
         < assign >
          < copy >
            < from > $ < xsl : value − ofselect =" StateManager"/ > _SupplyState_out.parameters <
   /from >
            < to > $ < xsl : value − ofselect =" @partnerLink"/ > _setState_in.parameters < /to >
          < /copy >
         < /assign >
         < invoke name =" @partnerLink_setState" partnerLink =" @partnerLink"
          portType =" @portType" operation =" setState"
          inputVariable =" @partnerLink_setState_in"
          outputVariable =" @partnerLink_setState_out"/ >
       < /sequence >
     < /if >
     ....
   < /sequence >
```

Figure 8.18: Second set of BPEL transformations performed by the state access and manipulation module

In addition, the BPEL transformation component sets a flag (*update_state*) to true at the end of each fault handler, as listed in Figure 8.18 (from Line 1 to Line 12). This flag presents the condition that must fold in order to invoke a *SetState* operation after the fault handler. The < *invoke* > activity to the *SetState* operation of the substitute service is put in the scope outside the fault handler (FH) (Line 20), in order to re-execute the FH if a disconnection occurs

when executing the *SetState* invoke activity.

### Service replacement

The BPEL process defines the set of partner links related to the services it has to interact with. These partner links are statically declared before runtime, and cannot be modified during the process execution. However, when one of these services becomes unavailable, we need to assign a new value to the related partner link. Replacing a service with another one requires the partner link definition to point out the substitute service in the place of the service that becomes unavailable. To support dynamic binding, we use the notion of End Point Reference (EPR) from the WS-Addressing specification [W3C, 2004d]. The definition of EPR enables the value of partner links to be dynamically assigned, and thus, interacting with partners that were not known at the time of defining the process. The WS-Addressing standard provides the XML schema of the endpoint reference type, which is described in Appendix B (Section B.2).

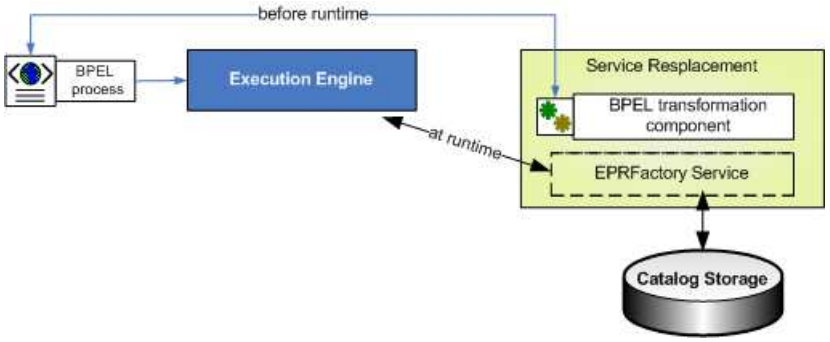



Figure 8.19: Service replacement

The dynamic binding and service replacement are performed using the *service replacement* module, which includes two entities:

1. A *BPEL transformation* component that acts before runtime by transforming the BPEL process, in order to enhance it with dynamic bindings, and

2. An *EPRFactory* service is deployed as Web service, and which is responsible for interacting with the BPEL process at runtime in order to provide it with a reference to the substitute service(s). To this aim, the *EPRFactory* service implements two operations, namely, $hasReplacementEPR$ and $getAddress$.

The BPEL transformation component adds the partner link, the required namespaces, and the required variables of the *EPRFactory* service in the BPEL process in order to enable their interaction, as listed in Figure 8.20.

To enable the service replacement when a service becomes unavailable, the *BPEL transformation* component adds a set of BPEL activities in the fault handler of each scope, in order to provide the running instance of the BPEL process with a reference to the substitute service. The listing of Figure 8.21 is generated by the BPEL transformation component at the beginning of each fault handler. The code of the BPEL transformation component that enables these XML transformations is provided in Appendix B (Section B.3).

The *BPEL transformation* component first adds an $<$ *invoke* $>$ activity of the operation $hasReplacementEPR$ provided by the *EPRFactory* service to check the availability of candidate services that are able to replace the unavailable one. This operation takes as input the catalog

```
< process...
  xmlns : nsFactory =" http : //factory.samples/"
  xmlns : nsFactoryXsd =" http : //factory.samples/xsd" >
  ...
<! − −import the wsdl of the EPR Factory service − − >
< import location =" EPRFactoryService.wsdl" namespace =" http : //factory.samples/xsd"
  importType =" http : //schemas.xmlsoap.org/wsdl/"/ >
  ...
<! − −EPR factory partnerLink − − >
< partnerLink name =" EPRFactory" partnerRole =" EPRFactoryServicePortType_provider"
  partnerLinkType =" nsFactory : EPRFactoryServicePortType_PL"
  initializePartnerRole =" yes"/ >
< /partnerLinks >
<! − −variables for the EPR Factory − − >
< variable name =" partnerLinkType_in" messageType =" nsFactory : partnerLinkTypeRequest"/ >
< variable name =" invoke_getAddress_out" messageType =" nsFactory : getAddressResponse"/ >
< variable name =" invoke_notify_error_out" messageType =" nsFactory : notifyErrorResponse"/ >
< variable name =" invoke_has_EPR_out" messageType =" nsFactory :
hasReplacementEPRResponse"/ >
```

Figure 8.20: Defining a reference to the EPRFactory service

identifier (i.e., the semantic concept) and returns a boolean output, *invoke_has_EPR_out* (Line 1). Depending on the result provided by the invocation *hasReplacementEPR*, it either replaces the endpoint reference of the unavailable service with the one of the substitute service, or it re-throws the fault (Line 7). In the case of availability of service candidates, an $< invoke >$ activity of the operation *getAddress* is added in the conditional structure to retrieve the endpoint reference of the substitute service (Line 9). The returned value of the endpoint reference *invoke_getAddress_out* is then assigned to partner link of the service being substituted (from Line 15 to Line 20). In this way, the following $< invoke >$ activities in the process will use the last assigned value of the partner link.

At runtime, a BPEL process invokes the operations *hasReplacementEPR* and *getAddress*. The *EPRFactory* service inspects the string passed in the input argument by the process. To enable the search in the *service registry*, the input string corresponds to the semantic concept of functional purpose of the requested capability. Then, the *EPRFactory* service checks in the *catalog storage* of the *service registry*, the catalog corresponding to the requested service, and returns the EPR that includes the URI, the PortType definition and the service name of the first-listed service.

One of the limitation of existing BPEL execution engines lies in their incapacity to dynamically change the operation name in the process activities. Hence, at the current stage of implementation, the replacement service module that is migrated to ODE, supports the substitution of services that match syntactically with the substituted service. This limitation joins the other previously mentioned limitations that lead us to implement our SIROCCO Execution Engine. The following modules of the *adaptation manager* modules are tested on *SIROCCO execution engine*.

#### Workflow transformation and management

The *workflow transformation and management* module takes as input the BPEL executable process that corresponds to the composite service behavior, or the orchestration that has to be executed. In the case of the composite service behavior, the *workflow transformation and*

```
1  < invoke name =" Invoke_notifyError" partnerLink =" EPRFactory"
     portType =" nsFactory : EPRFactoryServicePortType"
     operation =" hasReplacementEPR"
     inputVariable =" partnerLinkType_in"
     outputVariable =" invoke_has_EPR_out"/ >
   < if >
7    < condition > $invoke_has_EPR_out.parameters/nsFactoryXsd : return/text() < /condition >
     < sequence >
9      < invoke name =" Invoke_getAddress" partnerLink =" EPRFactory"
         portType =" nsFactory : EPRFactoryServicePortType"
         operation =" getAddress"
         inputVariable =" partnerLinkType_in"
         outputVariable =" invoke_getAddress_out"/ >
       <! − −Update the partnerlink of the service to invoke − − >
15     < assign >
         < copy >
           < from > $invoke_getAddress_out.parameters/nsFactoryXsd : return < /from >
           < to partnerLink =" Name of the service partner link"/ >
           < /copy >
20     < /assign >
     < /sequence >
   < else >
     < rethrow/ >
   < /else >
   < /if >
```

Figure 8.21: Replacing the value of the partner link

*management* module also takes as input the description of the checkpoints that are supported by the service behavior. In the case of service orchestration, it takes as input, besides the orchestration BPEL process, the individual BPEL descriptions of the involved services as well as their checkpoint descriptions. As illustrated in Figure 8.22, the *workflow transformation and management* module provides as output the transformation of the BPEL process into an aFSA.

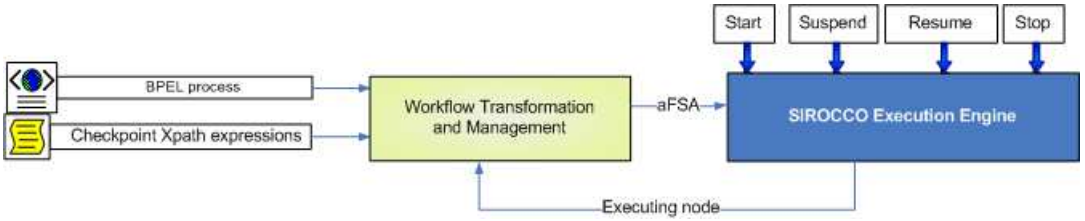



Figure 8.22: Workflow transformation and management

The *workflow transformation and management* performs the following functionalities:

1. Before runtime, it transforms the BPEL process into an aFSA.

2. It integrates the *state transfer* and *rollback* checkpoints in the aFSA.

3. At runtime, when a service becomes unavailable and a service substitute is selected, the *workflow transformation and management* module integrates in the workflow of the substitute service, in the aFSA that has not been executed.

a) It decomposes the behavior of the substituted service into a sequence to two workflows, where the first corresponds to the executed part and the second the rolled back and non-performed part of the behavior.

b) It transforms the behavior of the substitute service into an aFSA.

c) Then, it decomposes the behavior of the substitute service into a sequence of two workflows. The decomposition is performed on the basis of the compatibility check that returns the compatible checkpoint at which the state transfer can be performed.

4. It establishes the data flow of the substitute service, and integrates it in its aFSA.

5. In the case of service orchestration, it updates the data flow of the orchestration.

#### Data dependency management

In the case of service orchestration, the *data dependency management* module checks the data dependencies between checkpoints of the service behaviors that are compositing the orchestration. More specifically, the data dependency is checked between the *state transfer* checkpoint of substituted service at which the state transfer will be performed, and the *roll back* checkpoints of the still available services. It takes as input the aFSA that resulted from the *workflow transformation and management* module, and establishes the data dependency graph for the orchestration. As we assume that *state transfer* checkpoints coincide with a set of *rollback* checkpoints, the data dependency graph is set with regard to all the checkpoints of the service behaviors. The dependency graph is established before starting the execution of the orchestration in order to anticipate the service unavailability. After service substitution, it updates the data dependency graph with respect to the transformed aFSA.

#### Rollback checkpoint management

At runtime, when a service becomes unavailable, the *rollback checkpoints management* module is responsible for computing the recovery line on the basis of the data dependency graph. It takes as input the *state transfer* checkpoint at which the state transfer has been performed in the substitute service and the data dependency graph, and checks all the *rollback* checkpoints that are dependent on it, and returns the set of checkpoints that forms the recovery line. The recovery line is an association between services and rollback checkpoints, it associates with each *rollback* checkpoint, the service endpoint on which the rollback has to be performed.

#### Rollback management

The *rollback management* module puts the orchestration back to a consistent global state with respect to the recovery line computed by the *rollback checkpoint management*. The *rollback management* module is a generic client that takes as input a service endpoint, and the operation that it has to invoke. Given a service endpoint, the *rollback management* module implements an operation *GetRollbackOp* that parses the service WSDL in order to retrieve its operations, as well as their related semantic annotation, i.e., the value of the attribute $sawsdl : modelreference$. It then retrieves the syntactic name of the operation annotated with the *OperationImpact#Rollback*, and invokes it by giving the checkpoint identifier as input argument.

### 8.4.3 Evaluation: Dynamic Reconfiguration Assessment for Stateful Web Services

The previous sections detailed how SIROCCO middleware implements runtime service substitution. We have shown that for tool-support limitation, we implemented our SIROCCO Execution Engine to assess the feasibility of our solution and perform a set of tests, then we started to migrate our approach on a commonly agreed on BPEL execution engine: ODE. As the migration to ODE requires enriching ODE with extra features such as rollback and support of BPEL process transformation, a set of modules of the *adaptation manager* have not yet been migrated. These modules are essentially related to service orchestration. In this section, we assess our prototype implementation into two steps: One step is assessed using ODE, which shows that automatic runtime service substitution is feasible. The assessment is performed in various scenarios using an implementation of our train ticket booking service. The other step is assessed using SIROCCO Execution Engine, which shows the effect of the rollback on reconfiguring service orchestrations.

#### Assessing state transfer in service substitution

The behavior of train ticket booking service includes two atomic action: one atomic action that enables the selection of the train ticket, and the other for confirming and paying the selected tickets. In our scenario, we deployed four replicas of the train ticket booking service, denoted, *train booking service 1* to *4*. As illustrated in the sequence diagram of Figure 8.23, we tested three cases of failures in order to assess the automatic service substitution at runtime.

Starting from a BPEL process and the four stateful replicas of the train ticket booking service, the integration layer generated and deployed automatically the "proxy" related to the four replicas, and the *service reconfiguration* performed the required transformations on the BPEL process to handle checkpoints, state management and dynamic bindings. Then, we start the execution and monitor the trace of the execution through the exchanged messages between ODE execution engine and Globus container where the stateful Web services are deployed. The 3 cases of failures are described in the following.

1. In the first case of failure, the *runtime service management* tries to interact with a service that is undeployed and it fails. There is no special reconfiguration is such a case, but calling for a new EPR provided by the *EPRFactory* service. The *runtime service management* invokes first $hasReplacementEPR$ operation, it then invokes the operation $getAddress$. This case of failure assesses the feasibility of the dynamic binding.

2. Then, the *runtime service management* instantiates the candidate service according to the result of the $getAddress$ operation. It invokes $SelectSeatPreferences$ operation. Then, it completes the first atomic action and tries to get the service state. In this case, the failure occurs at the recovery operation. *Runtime service management* instantiates the candidate service after invoking the operation $hasReplacementEPR$ and $getAddress$ of the *EPRFactory* service. It re-executes the first atomic action, which includes the operation $SelectSeatPreferences$. Then, it gets the service state successfully, and stores it in the *state storage* using the operation $StoreState$. This case of failure emphasizes that our substitution ensures data consistency: as long as the state has not been stored, the previously performed computation is not taken in account in the transferred state.

3. *Runtime service management* asks the user whether s/he would like to confirm and pay, or only confirm the selected ticket. The user replies with an acceptation for payment. The *runtime service management* proceeds then to the confirmation of the train tickets. At

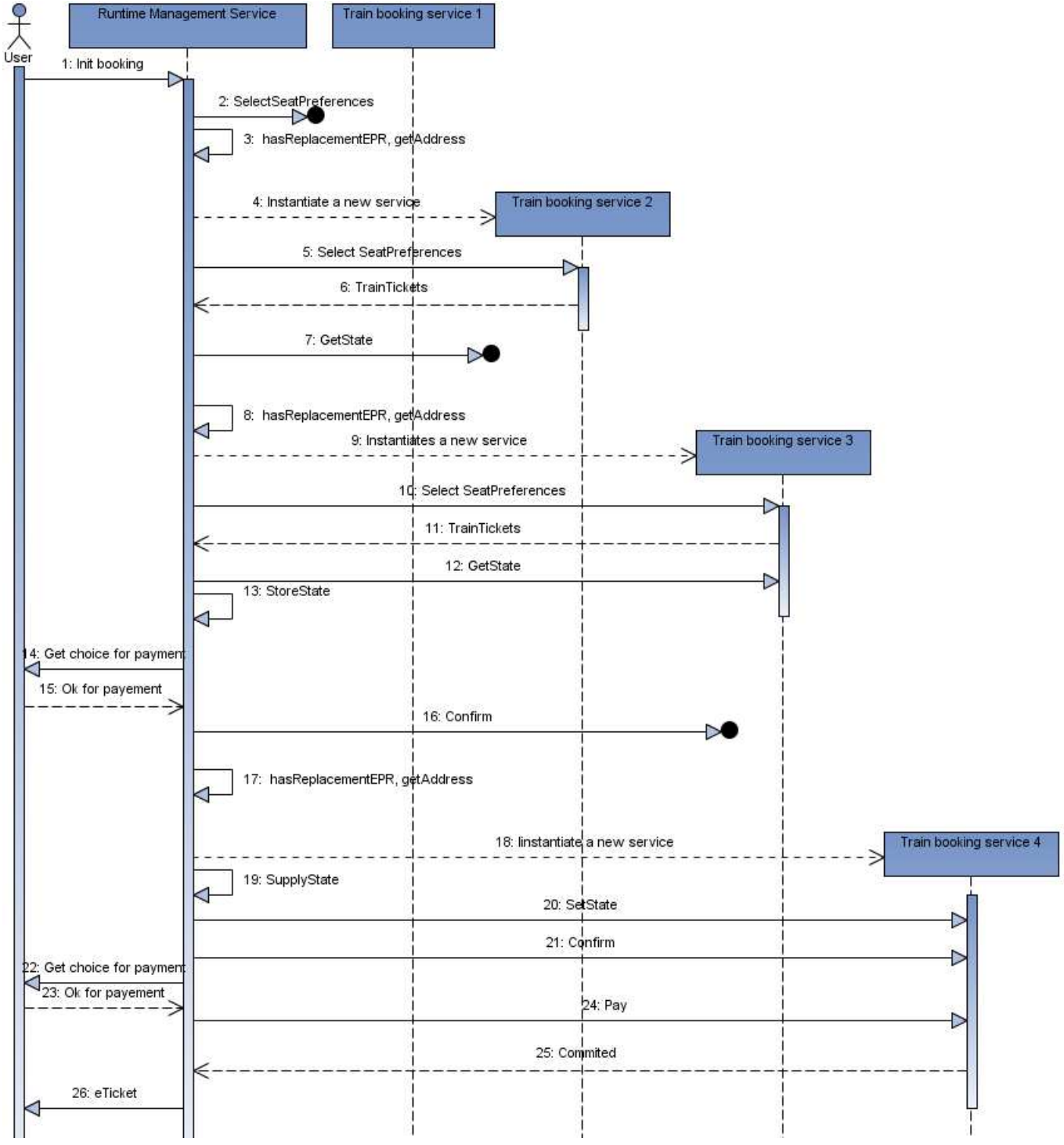



Figure 8.23: Sequence diagram of the train ticket booking scenario

this stage, the "train booking service 3" becomes unavailable. The *runtime service management* retrieves a new service candidate from the *EPRFactory* and invokes the operation *SupplyState* provided by the *state access and management* module. It invokes the *SetState* operation of the newly instantiated candidate service, and restarts executing the second atomic action execution. As the user choice is included in the second atomic action, the request for the user choice is re-executed after the state transfer. Then, the "train booking service 4" completes the execution normally. This case of failure emphasizes the gain in terms of computation as we resume the execution from the last checkpointed activity.

### Evaluating rollback propagation in service orchestration

To evaluate the advanced functionalities of the *service reconfiguration* when considering service orchestrations, we used SIROCCO Execution Engine and performed a number of experiments. The prototype and all our experiments were based on the AXIS 1 SOAP engine [12] and the Apache Tomcat [13] application server.

The main benefit from using SIROCCO Execution Engine for the executing of service orchestrations is the ability to dynamically change the orchestration workflow and perform rollback on still available services, where interleaving service behaviors and checkpoint intervals is possible. Hereafter, we use the term *enhanced-orchestration* to refer to an orchestration enriched with scopes, fault handlers and checkpoints. Respectively, we use the term *original-orchestration* to refer to an orchestration that does not include any reliability handling.

We performed two sets of experiments. In the first set, we compared the execution time of enhanced-orchestrations against the execution time of the original-orchestration in various scenarios of normal execution (i.e., there were no unavailable services during the orchestrations execution). In the second set of experiments, we measured the execution time of enhanced orchestrations in various failure scenarios that can not be handled by the original-orchestrations.

In both sets of experiments we used BPEL orchestrations that combined 5 Web services $(WS_1, WS_2, \ldots, WS_5)$, each one of which offered 10 operations. These operations perform calculating tasks. They do not store an implementation state, all the required parameters for processing a calculating task are either provided as input, or retrieved from the WS-resource the service maintains, or from both of them. The absence of implementation state enables to play on the checkpoints positions when performing the testing evaluation. In this way, changing a checkpoint position does not introduce data incoherence in the service behavior execution.

The control flow of the orchestrations was derived from a combination of two well-known workflow patterns (Sequence and AND-Split). Specifically, each orchestration consists of a flow activity that comprises 5 sequence activities $(SQ_1, SQ_2, \ldots, SQ_5)$ which execute concurrently. Each sequence $SQ_i$ consists of 10 basic activities $A_{SQ_{i1}}, \ldots A_{SQ_{i10}}$ which invoke the operations of $WS_i$. The data flow dependencies between the activities were set according to the following pattern: the output messages of the service operations invoked in activities $A_{SQ_{ij}}$, $j \in [1 \ldots 9]$ have been used for constructing input messages for the service invocations of the activities $A_{SQ_{(i+1)(j+1)}}$, $A_{SQ_{(i+2)(j+1)}}$, $A_{SQ_{(i+3)(j+1)}}$. In both sets of experiments, the *SIROCCO service reconfiguration* was deployed on an 1.6 GHz Intel Centrino, with 1GB RAM, while the services were deployed on 1.7 Intel Pentium, with 1 GB RAM, communicating using a WIFI network connection.

Finally, in both sets of experiments, we used 4 different cases of orchestration checkpointing, where we varied the number of operations of each service that change the state of the service as follows: 1, 2, 5 and 10 operations per service. Therefore, we varied the number of checkpointing

[12] http://ws.apache.org/axis/index.html
[13] http://tomcat.apache.org/

activities introduced in the orchestrations from 5 to 50. More specifically, in each case, we increased the number of checkpoints per service $WS_i$, reaching (in the last case) 1 checkpoint per operation (i.e., 10 checkpoints/service). In the first case, the checkpoints are performed after executing 5 operations of each service $WS_i$, thus we have 5 checkpoints for the whole orchestration (referenced in the following as case 1). Then, in the second case (case 2), the checkpoints are performed every 4 operations of each service $WS_i$, reaching 10 checkpoints for the whole orchestration. Afterwards, in the third (resp., fourth) case, the checkpoints are performed after every 2 operations (resp., every operation) of each service $WS_i$, reaching 25 (resp., 50) checkpoints for the whole orchestration.

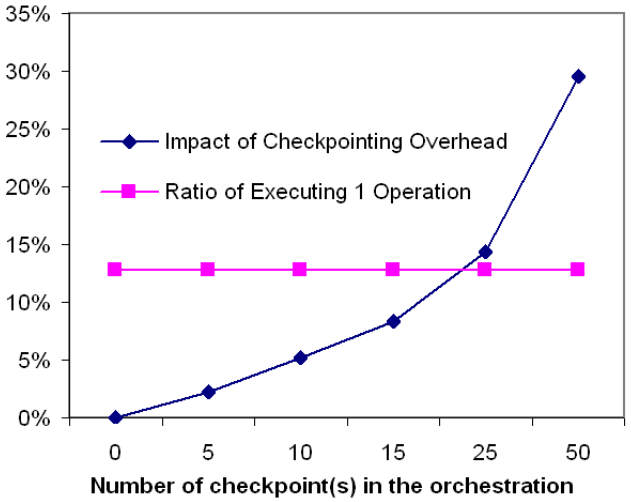



Figure 8.24: Impact of the checkpointing overhead on the orchestration execution time

To evaluate the impact of the checkpointing overhead on the orchestration execution, we first measured the execution time ($ET$) with and without checkpoints for each of the above described cases. The impact of checkpointing is then computed as a ratio of the checkpointing-induced overhead over the orchestration execution time in the best case – noted $ET$ in $BC$ (i.e., without checkpointing).

$$\text{Impact of the Checkpointing Overhead} = \frac{\text{ET with checkpoints - ET in BC}}{\text{ET in BC}}$$

Figure 8.24 provides the result of the first set of experiments, where each checkpointing overhead value is measured as an average of 10 measurements of the orchestration execution time. We compared the checkpointing overhead to the time required to execute one operation of the orchestration that represents 12.7% of the $ET$ in $BC$. We noticed that the checkpointing overhead is remains reasonable in the 3 first cases where the checkpoints number is less that 50% of the number of invocations. In the case 3, it takes 14.37% of the $EC$ in $BC$ and it reaches 29.53% in the worse case tested (1 checkpoint per operation –case 4).

Regarding the recovery overhead, we considered the same orchestration, injecting disconnection events for $WS_1$ progressively at different stages of its execution (i.e., at the activities $A_{SQ_{1,1-i}}$ where $i \in [1..10]$). To evaluate the recovery overhead, we compared the execution time of the orchestration ($ET'$), which includes the disconnection notification and the recovery time, to its execution time $ET$ in $BC$ (i.e., without checkpointing and without disconnection).

$$\text{Impact of the Recovery Overhead} = \frac{\text{ET' - ET in BC}}{\text{ET in BC}}$$

For each disconnection at $A_{SQ_{1,1-i}}$, we measured thus the impact of the recovery overhead on the orchestration execution using two recovery methods: (i) restarting the overall execution of the orchestration (called "restarting-based recovery"), and (ii) using our proposed rollback-based recovery. For the latter, the measurements have been made for each of the checkpointed orchestrations corresponding to the 4 cases described above. The recovery time includes in the case of "restarting-based recovery" the time to invalidate all the interactions prior to the disconnection, and to resume the orchestration execution from its beginning till the disconnection point. In the case of "rollback-based recovery", the recovery time comprises (i) the time to invalidate the set of interactions performed with the disconnected service, (ii) the time to compute the recovery line, and (iii) the re-execution time to get the orchestration back to the point at which the execution has been interrupted. First, we measured the mean time to compute the recovery line at each

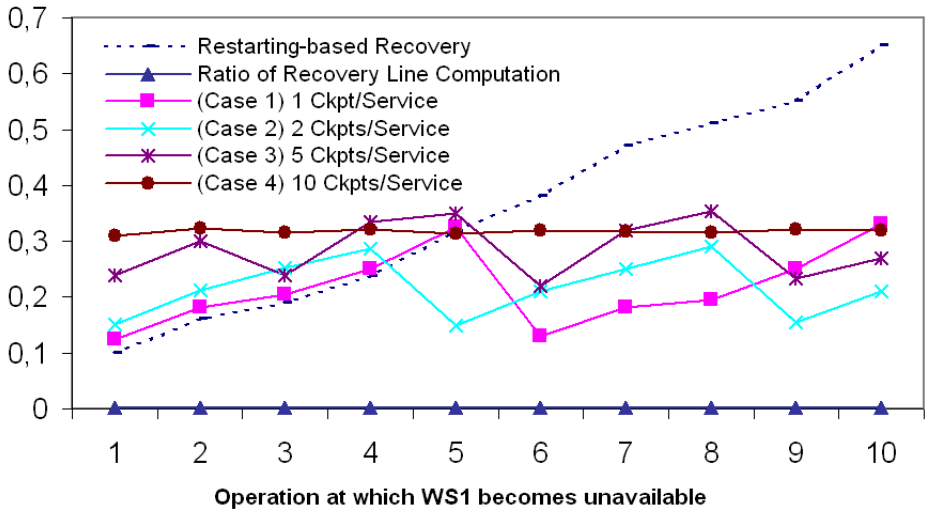



Figure 8.25: Impact of the recovery overhead on the orchestration execution time

case of disconnection and for each checkpointed orchestration (represented by the curve "Overhead of Recovery Line Computation" in Figure 8.25). We noticed that this time represents 0.2% (in the worse case) of the orchestration $ET$ in $BC$, deducing thereby that most of the overhead introduced by the rollback recovery is due to *checkpointing* and the *dependencies* between the orchestrated services. The curves in Figure 8.25 (Cases 1 to 4) illustrate the recovery overhead for the 4 cases of checkpointed orchestration. The recovery overhead is affected by the checkpoints' *number* and *positions* within the workflow of the orchestrated services. Indeed, it is important to note that there is a tradeoff between the checkpointing overhead and the time required to recover from a connectivity loss. The denser the checkpoints are, the less we may have to rollback and thus the less the recovery time is. However, as the number of checkpoints increases, the orchestration execution time increases as well, thus the denser the checkpoints are, the higher the checkpoint overhead is (see the curve "Checkpointing Overhead" in Fig. 8.24). Furthermore, we noticed that the checkpoints position plays an important role on the recovery overhead. Indeed, in the 4 cases of checkpointed orchestration, there is a noticeable difference between the recovery overhead before and after a checkpoint. Finally, we noticed that when the $WS_1$ becomes unavailable at $A_{SQ_{1,1-i}}$ ($i \in [1..4]$), the recovery overhead is higher for rollback-based recovery compared to the restarting-based recovery, due to checkpointing overhead. However, as the execution progresses and $WS_1$ becomes unavailable at $A_{SQ_{1,1-i}}$ ($i \in [5..10]$), the rollback-based recovery –in the 4 cases tested– performs better than restarting-based recovery, as the saved computation

covers the checkpointing overhead, in comparison to the time required to re-execute the whole orchestration.

## 8.5 Concluding Remarks

In this Chapter, we presented our SIROCCO middleware implementation for dynamic service substitution in the domain of WS. Through our realization, we assessed the feasibility of our theoretical approach that has been established in the previous parts of this thesis. We introduced a number of mechanisms that enhance BPEL processes with dynamic binding and state awareness, at runtime, while respecting the structural logic of the processes. However, as the existing BPEL execution engines turn quickly to be short in flexibility and runtime management support, we implemented our own BPEL SIROCCO Execution Engine in order to test the advanced reconfiguration functionalities such as rollback propagation and workflow dynamic transformation at runtime. However, SIROCCO Execution Engine supports the basic functionalities related to the runtime reconfiguration. The objective behind its implementation is not to compete with the existing BPEL execution engines, but to assess the feasibility our reconfiguration algorithms, and evaluate them freely without any technological limit. Still, some modules of the adaptation manager such as the *compatibility check* and *replay* have to be implemented to fully assess our theoretical solution. As a next step to the current implementation, we target to integrate the full reconfiguration-supported functionalities of SIROCCO Execution Engine into ODE.

*When I am working on a problem I never think about beauty. I only think about how to solve the problem. But when I have finished, if the solution is not beautiful, I know it is wrong.*

Buckminster, Fuller

# 9 Conclusions and Future Research Directions

With the advent of wireless networks, software computing systems have evolved to reach a dimension of pervasiveness. Heterogeneous, open, dynamic software systems populate todays' computing environments, making computing facilities accessible anywhere, at any time. However, consuming these computing facilities raises several challenges due to software platforms heterogeneity, openness and dynamics. Service-oriented architectures (SOA) deal with heterogeneity of software systems, by abstracting the computing facilities that populate the environment as services. Services have well-known, standardized descriptions that enable to consume their functionalities in a standardized way independently of their software platform. Still, the environment's openness and dynamics have to be faced out in order to overcome service unavailability and ensure continuity in service provisioning. In this thesis, we tackled the issue of service unavailability due to service failures or network disconnection without any insurance of a future reconnection, in order to enhance todays' SOA systems with reliability.

In this chapter, we draw up an overview of the approach that we proposed (Section 9.1). We then emphasize the lessons learned from this approach (Section 9.2). We finally wrap up this thesis with our future research directions that present the continuity of the herein-presented work (Section 9.3).

## 9.1 Overview of the Proposed Approach

As we are interested in enhancing SOA systems with reliability, we first started by focusing on service-oriented paradigm, and more specifically, on the notion of service. We established a conceptual model that includes the main concepts that define a service and their dependencies. In our study, we have taken a user-oriented perspective. From our perspective, the service capability, behavior and state turn to be the essential concepts that have to be defined when considering the definition of a service, because of the role they play for enabling service consumption.

On the way to find a solution for reliability in SOA systems, we investigated the dependability basic concepts and the dependability means that cope with service unavailability. In this investigation, fault tolerance means respect the most the environment dynamics and openness. The fault tolerance (FT) means have been widely used in closed distributed systems, in particular, with built-in replication and checkpoint-based rollback recovery to tolerate component unavailability. Still, FT means applied in closed distributed systems are not applicable as they are in SOA systems: exact replicas, code determinism, and forced/coordinated checkpointing are too restrictive for SOA systems as they do not respect service autonomy and loose coupling. We discussed the limits of applicability of these FT means in open, dynamic SOA system, stressing thereby the need to adapt them to SOA systems specifics [Fredj et al., 2006, Zarras et al., 2006].

The review of the state of the art showed that existing service substitution approaches in SOA systems adapt the application of FT means in SOA systems, but they are far to realize dynamic service substitution without restricting the service environment or service autonomy and loose coupling.

The approach that we proposed consists in substituting at runtime stateful, composite services, while allowing heterogeneity in service capability, service behavior and service state between the substituted service and its substitute. The matching between capabilities is based on semantic inclusion relationship, which is more flexible that exact or equivalent matching. The behavior of substitute service can present different structure than the unavailable service. The state description of the substitute service also can be differently defined from the one of the unavailable service. To allow such flexibility, the need for formal definition of service substitution was required in order to establish the set of rules that ensure correct runtime service substitution. To this aim, we adapted the subtyping notion issued from object oriented design, so as to establish a hierarchical order between services. Hierarchy allows the use of service groups as type families, in which a group of services (subtype) may serve as substitute candidate for a service included in another group (supertype) that is higher in the hierarchy. Still, the rules issued from the adaptation of subtyping do not take into account runtime behavior execution. To ensure correct runtime substitution, we applied Hoare's rules for program composition and replacement, issued from his axiomatic for computer programming.

The established rules enabled to check whether a service is able to serve as a substitute for another service, or not. In particular, we defined a compatibility degree between services that, not only, enables to determine whether a service complies with the substitution rules, but also, in the case of multiple service candidates for substitution, it enables to classify service according to their degree of compatibility with the substituted service. This classification enables to select the service that is best compatible with the substituted service.

Once the compatibility with candidate services is checked and a service is selected, we define a set of strategies for synchronizing the substitute service either by transferring the state of the unavailable service or by reconstructing the state of unavailable service in the substitute service using the replay technique. In the case of service orchestration, we delimit the impact of the substitution on still-available services, due to their data dependencies, and reconfigure them in order to restore the orchestration consistency. This is performed by adapting uncoordinated checkpointing rollback protocol according to our definition of checkpoints for SOA systems. Once the services of the orchestration are reconfigured, we reconfigure the client by integrating the behavior of the substitute service in the orchestration workflow. The proposed approach is assessed through our implementation of SIROCCO middleware.

## 9.2 Learned Lessons

The novelty of the proposed approach resides essentially in the flexibility it brings in the domain of dependability in SOA systems. SOA systems are dynamic and open, which makes their worldwide popularity. However, when it comes to reliability, the existing solutions turns to be constrainful, or irrealistic to be applied in open, dynamic SOA system environments. For instance, network-based solutions relies on the support of the core network infrastructure for ensuring the constant network connection between the clients and their services, which is not realizable in infrastructure-less networks. Other solutions for SOA systems assume the existence of exact replica in the networked environment.

In our approach, we take advantage of these solutions, wand enhance them with dynamic service substitution. We consider that if the network infrastructure provides a possibility to

reach the same service instance reachable through an alternative network or path, we would opt for this solution rather than substituting the service. In the case of absence of network support, we provide an alternative solution using service reconfiguration [Rong et al., 2007b, Rong et al., 2007a]. In the case of service substitution, exact replica is not set as a necessary condition to replace the unavailable service [Fredj et al., 2008]. Nevertheless, if we have the possibility to find an exact replica, we integrate it as a substitute service instead of any other candidate service, using the classification of the candidate services. Otherwise, we select the service that present the most compatible description with the unavailable service, and perform the substitution [Fredj et al., 2009]. Our approach is founded on the basis of formal theories that have been widely used and assessed, which ensures the correctness of the runtime substitution, even when services are not identically implemented or described.

## 9.3 Future Research Directions

In the continuity of the approach proposed in this thesis, a number of major aspects can be carried out. We organize them into three categories with accordance to the expected evolutions of our approach over time: short term, middle term and long term research directions.

**Short term directions** are essentially related to the realization of our approach. Many aspects can be followed to evolve the current state of the realization. Among them, we retrieve three immediate evolutions:

1. Integrating the full functionalities of our service reconfiguration into ODE execution engine, in order to provide a standalone support for runtime service substitution for Web services.

2. The previous evolution implies naturally to fully implement the *compatibility check* and *replay* modules and evaluate them, in order to be integrated in the SIROCCO service reconfiguration.

3. Another aspect of evolution consists in migrating the approach on lightweight devices, where only a part functionalities will be deployed on resource-constrained device, such as the *execution engine*, the *state access and management*, and the *rollback* modules. While, the other functionalities of SIROCCO middleware will be deployed on central or distributed devices in the user's environment, such as the *service composition* and the *compatibility check* modules.

**Middle term directions** concern mostly the research aspects that can nicely complement the aspects that have been investigated in our approach.

In our approach, we essentially considered the case of service orchestration. While, the service choreography presents many challenging issues that have to be investigated, in order to take into account its specific aspects and adapt our approach accordingly.

Furthermore, in our approach, we focused in the user request in order to find services that serve the requested capability. However, besides the request capability, the user request may be enriched with the definition of the side effects that would affect the user. The definition of side effects would then be considered when selecting the candidate service for substitution.

**Long term directions** envision the use of our approach beyond the scope of stateful composite Web services, to reach the domain of distributed, real time (DRT) systems. These systems present

strict time constraints that have to be respected, in addition to potentially constrained resources. For instance, in the railway domain, the services provided to the trains have specific Safety Integrity Level (SIL) [1]. Meaning that, even though a service is provided with a SIL that equals 1 (the lowest level), the reconfiguration has to respect the system constraint otherwise other (non critical, but still important) systems may be highly damaged. In these systems, the recovery time is not computed according to an average (mean time to recover $MTTR$), but with respect to the worst case of reconfiguration time, as uncertainty is not allowed. The reconfiguration time has to be provided and respected in all circumstances, in order to check whether the reconfiguration respects the system constraints, or not. The dynamic reconfiguration that we proposed in this thesis, has to gain in maturity in order to be applied in DRT systems.

[1]SIL measures the level of safety that services must satisfy. It is evaluted from 1 to 4, where SIL 4 is the highest level of safety required.

# A

# BPEL Execution Engines

| BPEL Execution Engine and the related organization | Container | Language | Visual designer | Licence |
|---|---|---|---|---|
| Oracle | runs in any standard servlet container such as Apache Tomcat | Java | no | GNU General Public License (GPL) |
| ActiveBPEL from Active endpoints | runs in any standard servlet container such as Apache Tomcat | Java | no | GNU General Public License (GPL) |
| Orchestra from BULL SAS-OW2 consortium | entreprise vesion: JOnAS, Light version:tomcat | Java | Zenflow | LGPL License |
| BEXEE at the Berne University of Applied Sciences, School of Engineering and Information Technology | AXIS Webapp | Java – bexee is no longer actively developed | no | GNU General Public License (GPL) |
| PXE from FiveSight | not mentioned | Java | no | CPL (core) / MIT License (extensions) |
| ODE from Apache | tomcat webapp | Java | eclipse plugin | Apache License |
| Sliver from Washington university St Louis | implements its SOAP server | Java | no | GNU Lesser General Public License |

Table A.1: Summary of BPEL execution engines

# B

# XSL Transformations and Code Generation

## B.1 XSL Transformations performed by the Monitoring Manager

```
< sequencexmlns = "http : //docs.oasis − open.org/wsbpel/2.0/process/executable" >
    < xsl : copy − ofselect = "@ ∗ "/ >
    < xsl : call − templatename = "preambule"/ >

    < xsl : elementname = " {name(.)} " >
    < xsl : copy − ofselect = "@ ∗ "/ >
    < xsl : apply − templates/ >
    < /xsl : element >
< /sequence >

    < /xsl : template >
    < xsl : templatename = "preambule" >
    < xsl : call − templatename = "lineJump"/ >
    < xsl : call − templatename = "add_comment" >
      < xsl : with − paramname = "valueComment" > EnablingOdetocaptureAxis2errors <
/xsl : with − param >
    < /xsl : call − template >
    < ext : failureHandlingxmlns : ext = "http : //ode.apache.org/activityRecovery" >
      < ext : faultOnFailure > true < /ext : faultOnFailure >
    < /ext : failureHandling >
    < xsl : call − templatename = "lineJump"/ >
< /xsl : template >
```

## B.2 EPR XML Schema

```
< xs : elementname = "EndpointReference"type = "wsa : EndpointReferenceType"/ >
< xs : complexTypename = "EndpointReferenceType" >
    < xs : sequence >
      < xs : elementname = "Address"type = "wsa : AttributedURI"/ >
     < xs : elementname = "ReferenceProperties"type = "wsa : ReferencePropertiesType"
          minOccurs = "0"/ >
       < xs : elementname = "PortType"type = "wsa : AttributedQName"minOccurs =
```

```
"0" / >
      < xs : elementname = "ServiceName"type = "wsa : ServiceNameType"minOccurs =
"0" / >
     < xs : anynamespace = "##other"processContents = "lax"minOccurs = "0"maxOccurs =
"unbounded" / >
   < /xs : sequence >
   < xs : anyAttributenamespace = "##other"processContents = "lax" / >
< /xs : complexType >
```

## B.3 BPEL Transformation Component of the Service Replacement

**Defining a reference to the EPRFactory service**

```
<! − −Process − − >
< xsl : templatematch = "bpel : process" >
   < processxmlns = "http : //docs.oasis − open.org/wsbpel/2.0/process/executable"
       xmlns : sref = "http : //docs.oasis − open.org/wsbpel/2.0/serviceref"
       xmlns : soap = "http : //schemas.xmlsoap.org/wsdl/soap/"
       xmlns : nsFactory = "http : //factory.samples/"
       xmlns : nsFactoryXsd = "http : //factory.samples/xsd" >
     < xsl : copy − ofselect = "@ ∗" / >
     < xsl : copy − ofselect = "namespace :: ∗[name()! =′′]" / >
       < xsl : copy − ofselect = "bpel : import" / >

     < importlocation = "EPRFactoryService.wsdl"
       namespace = "http : //factory.samples/"
       importType = "http : //schemas.xmlsoap.org/wsdl/" / >

     < importlocation = "EPRFactoryService.wsdl"
       namespace = "http : //factory.samples/xsd"
       importType = "http : //schemas.xmlsoap.org/wsdl/" / >
     < xsl : apply − templates/ >
   < /process >
< /xsl : template >

   <! − −PartnerLinks − − >
< xsl : templatematch = "bpel : partnerLinks" >
   < partnerLinksxmlns = "http : //docs.oasis−open.org/wsbpel/2.0/process/executable" >
     < xsl : copy − ofselect = "bpel : partnerLink" / >
     < xsl : call − templatename = "add_comment" >
       < xsl : with − paramname = "valueComment" > EPRFactory partnerLink < /xsl :
with − param >
     < /xsl : call − template >
     < partnerLinkname = "EPRFactory"
       partnerRole = "EPRFactoryServicePortType_provider"
       partnerLinkType = "nsFactory : EPRFactoryServicePortType_PL"
       initializePartnerRole = "yes" / >      < /partnerLinks >
```

```
</xsl:template>

    <!--Variables-->
<xsl:templatematch="bpel:variables">
    <variablesxmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable">
      <xsl:copy-ofselect="bpel:variable"/>
      <xsl:call-templatename="add_comment">
        <xsl:with-paramname="valueComment">variables for theEPR factory</
xsl:with-param>
      </xsl:call-template>
      <variablename="partnerLinkType_in"
            messageType="nsFactory:partnerLinkTypeRequest"/>
      <variablename="invoke_getAddress_out"
            messageType="nsFactory:getAddressResponse"/>
      <variablename="invoke_notify_error_out"
            messageType="nsFactory:notifyErrorResponse"/>
      <variablename="invoke_has_EPR_out"
            messageType="nsFactory:hasReplacementEPRResponse"/>
    </variables>
</xsl:template>
```

**Replacing the value of the partner link**

```
<faultHandlers>
    <catchAll>
      <sequence>
        <xsl:call-templatename="add_comment">
          <xsl:with-paramname="valueComment">TrytoobtainanewEPR</xsl:
with-param>
        </xsl:call-template>
        <invokename="Invoke_notifyError"partnerLink="EPRFactory"
       portType="nsFactory:EPRFactoryServicePortType"operation="hasReplacementEPR"
       inputVariable="partnerLinkType_in"outputVariable="invoke_has_EPR_out"/>
        <if>
        <condition>$invoke_has_EPR_out.parameters/nsFactoryXsd:return/text()</
condition>
          <sequence>
            <invokexmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
              name="Invoke_getAddress"partnerLink="UrlFactory"
             portType="nsFactory:UrlFactoryServicePortType"operation="getAddress"
          inputVariable="partnerLinkType_in"outputVariable="invoke_getAddress_out"/>
          <xsl:call-templatename="add_comment">
          <xsl:with-paramname="valueComment">Update the EPR of the service to invoke</
xsl:with-param>
          </xsl:call-template>
          <assignxmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable">
            <copy>
```

```
                < from > $invoke_getAddress_out.parameters/nsFactoryXsd : return <
/from >
              < topartnerLink = "@partnerLink" / >
            < /copy >
          < /assign >
          < /sequence >
        < else >
          < xsl : call − templatename = "add_comment" >
            < xsl : with − paramname = "valueComment" > No new EPR longer available,
             rethrow the exception < /xsl : with − param >
          < /xsl : call − template >
          < rethrow/ >
        < /else >
        < /if >
      < /sequence >
    < /catchAll >
< /faultHandlers >
```

## B.4 Generating "Proxy" Service for Globus Web Services

```
<?xmlversion = "1.0"encoding = "UTF − 8"? >
< xsl : stylesheetxmlns : xsl = "http : //www.w3.org/1999/XSL/Transform"version =
"2.0" >
    < xsl : outputmethod = "text" / >
    < xsl : templatematch = "/" >
      < xsl : apply − templatesselect = "system/service" / >
    < /xsl : template >
    < xsl : templatematch = "system/service" >
package < xsl : value−ofselect = "concat(translate(generalBase,′ /′,′ .′),′ .′, translate(subBase,
′/′,′ .′),′ .′, translate(package,′ /′,′ .′))" / > .axis2;
importjavax.xml.namespace.QName;
importorg.apache.axis.message.addressing.Address;
importorg.apache.axis.message.addressing.EndpointReferenceType;
importorg.oasis.wsrf.properties.WSResourcePropertiesServiceAddressingLocator;
importorg.oasis.wsrf.properties.SetResourceProperties_PortType;
importorg.oasis.wsrf.properties.SetResourceProperties_Element;
importorg.oasis.wsrf.properties.GetMultipleResourceProperties_Element;
importorg.oasis.wsrf.properties.GetMultipleResourcePropertiesResponse;
importorg.oasis.wsrf.properties.UpdateType;
importorg.apache.axis.message.MessageElement;
importorg.oasis.wsrf.properties.GetResourcePropertyResponse;
import < xsl : value−ofselect = "concat(translate(generalBase,′ /′,′ .′),′ .′, translate(subBase,′ /′,
′.′),′ .′, translate(package,′ /′,′ .′))" / > .globus.impl. < xsl : value − ofselect = "class" / >
QNames;
import < xsl : value − ofselect = "translate(generalBase,′ /′,′ .′)" / > . < xsl : value −
ofselect = "translate(subBase,′ /′,
′.′)" / > .stubs. < xsl : value − ofselect = "name" / > .service. < xsl : value − ofselect =
```

```
"class"/ > Service;
import < xsl : value − ofselect = "translate(generalBase,'/','.')"/ > . < xsl : value −
ofselect = "translate(subBase, '/','.')"/ > .stubs. < xsl : value − ofselect = "name"/ > . <
xsl : value − ofselect = "class"/ > PortType;
import < xsl : value − ofselect = "translate(generalBase,'/','.')"/ > . < xsl : value −
ofselect = "translate(subBase,'/',
'.')"/ > .stubs. < xsl : value − ofselect = "name"/ > .service. < xsl : value − ofselect =
"class"/ > ServiceAddressingLocator;
< xsl : for − eachselect = "operations/operation[count(input) = 0]" >
import < xsl : value − ofselect = "translate(../../generalBase,'/','.')"/ > . < xsl : value −
ofselect = "translate(../../subBase,
'/','.')"/ > .stubs. < xsl : value − ofselect = "../../name"/ > . < xsl : value − ofselect =
"name"/ >;
< /xsl : for − each >
importjava.util.∗;
importjava.io.∗;
importjava.beans.XMLEncoder;
importjava.beans.XMLDecoder;
publicclass < xsl : value − ofselect = "class"/ > Facade {
    private < xsl : value − ofselect = "class"/ > PortTypemethods_port;
    privateSetResourceProperties_PortTyperesource_port;
    public < xsl : value − ofselect = "class"/ > Facade()throwsException {
        < xsl : value − ofselect = "class"/ > ServiceAddressingLocatorlocatorPort = new <
xsl : value − ofselect = "class"/ > ServiceAddressingLocator();
       StringserviceURI = "http : //127.0.0.1 : 8081/wsrf/services/ < xsl : value−ofselect =
"subBase"/ > / < xsl : value − ofselect = "package"/ > / < xsl : value − ofselect =
"class"/ > Service";
        //Createendpointreferencetoservice
        EndpointReferenceTypemethods_endpoint = newEndpointReferenceType();
        methods_endpoint.setAddress(newAddress(serviceURI));
      this.methods_port = locatorPort.get < xsl : value−ofselect = "class"/ > PortTypePort
(methods_endpoint);
       WSResourcePropertiesServiceAddressingLocatorlocatorResource = newWSResource
PropertiesServiceAddressingLocator();
      this.resource_port = locatorResource.getSetResourcePropertiesPort(methods_endpoint);
    }
    publicStringgetState()throwsException {
        HashMapmapState = newHashMap();
        < xsl : for − eachselect = "resources/resource" >
        GetResourcePropertyResponse < xsl : value − ofselect = "lower − case(name)"/ >
RP = this.methods_port.getResourceProperty(< xsl : value − ofselect = "../../class"/ >
QNames.RP_ < xsl : value − ofselect = "upper − case(name)"/ >);
        String < xsl : value − ofselect = "lower − case(name)"/ >=< xsl : value − ofselect =
"lower − case(name)"/ > RP.get_any()[0].getValue();
        mapState.put(" < xsl : value − ofselect = "name"/ > ",< xsl : value − ofselect =
"lower − case(name)"/ >);
        < /xsl : for − each >
        ByteArrayOutputStreamarray = newByteArrayOutputStream();
```

```
    XMLEncoderencoder = newXMLEncoder(newBufferedOutputStream(array));
    encoder.writeObject(mapState);
    encoder.close();
    returnarray.toString();
  }
  publicStringsetState(StringparamState)throwsException{
    ByteArrayInputStreamarray = newByteArrayInputStream(paramState.getBytes());
    XMLDecoderdecoder = newXMLDecoder(newBufferedInputStream(array));
    HashMapmapState = (HashMap)decoder.readObject();
      < xsl : for − eachselect = "resources/resource" >
    this.methods_port.set < xsl : value−ofselect = "name"/ > RP(mapState.get(" < xsl :
value − ofselect = "name"/ > "));
      < /xsl : for − each >
    return"done";
  }
  < xsl : for − eachselect = "resources/resource" >
  publicStringget < xsl : value − ofselect = "name"/ > ()throwsException{
     GetResourcePropertyResponse < xsl : value − ofselect = "lower − case(name)"/ >
RP = this.methods_port.getResourceProperty(< xsl : value − ofselect = "../../class"/ >
QNames.RP_ < xsl : value − ofselect = "upper − case(name)"/ >);
    String < xsl : value − ofselect = "lower − case(name)"/ >=< xsl : value − ofselect =
"lower − case(name)"/ > RP.get_any()[0].getValue();
    return < xsl : value − ofselect = "lower − case(name)"/ >;
  }
  < /xsl : for − each >
  < xsl : for − eachselect = "operations/operation" >
  public < xsl : iftest = "count(output) = 0" > String < /xsl : if >< xsl : iftest =
"count(output) > 0" >< xsl : value−ofselect = "output/type"/ >< /xsl : if >< xsl : value−
ofselect = "concat( ' ', lower−case(name))"/ > (< xsl : iftest = "count(input) > 0" >< xsl :
value − ofselect = "concat(input/type, ' ', input/name)"/ >< /xsl : if >)throwsException{
    < xsl : iftest = "count(output) > 0" > return < /xsl : if > this.methods_port. < xsl :
value − ofselect = "lower − case(name)"/ > (< xsl : iftest = "count(input) = 0" > new <
xsl : value − ofselect = "name"/ > () < /xsl : if >< xsl : iftest = "count(input) = 1" ><
xsl : value − ofselect = "input/name" >< /xsl : value − of >< /xsl : if >);
    < xsl : iftest = "count(output) = 0" > return"done";< /xsl : if >
  }
  < /xsl : for − each >
}
  < /xsl : template >
< /xsl : stylesheet >
```

## B.5 Adding the PartnerlinkType to the WSDL Description of the "Proxy" Service

```
<?xmlversion = "1.0"encoding = "UTF − 8"? >
< xsl : stylesheetxmlns : xsl = "http : //www.w3.org/1999/XSL/Transform"version =
```

```
"2.0"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:ns1="http://org.apache.axis2/xsd"
    xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
    xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
    xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype">
    <xsl:outputmethod="xml"indent="yes"/>
    <xsl:templatematch="/">
      <xsl:apply-templates/>
    </xsl:template>
    <xsl:templatematch="wsdl:definitions">
      <wsdl:definitionstargetNamespace="@targetNamespace">
       <xsl:namespacename="axis2"><xsl:value-ofselect="@targetNamespace"/><
/xsl:namespace>
        <xsl:namespacename="ns"><xsl:value-ofselect="@targetNamespace"/>
xsd</xsl:namespace>
        <xsl:apply-templates/>
        <xsl:call-templatename="buildPartnerLinkType"/>
      </wsdl:definitions>
    </xsl:template>
    <xsl:templatematch="wsdl:service">
      <wsdl:servicename="@name">
        <xsl:apply-templates/>
      </wsdl:service>
    </xsl:template>
    <xsl:templatematch="wsdl:port">
      <wsdl:portname="@name"binding="@binding">
        <xsl:apply-templates/>
      </wsdl:port>
    </xsl:template>
    <xsl:templatematch="soap:address">
      <soap:addresslocation="http://localhost:8080/ode/processes/../../@name"/>
    </xsl:template>
    <xsl:templatematch="soap12:address">
      <soap12:addresslocation="http://localhost:8080/ode/processes/../../@name"/>
    </xsl:template>
    <xsl:templatematch="http:address">
      <http:addresslocation="http://localhost:8080/ode/processes/../../@name"/>
    </xsl:template>
    <xsl:templatematch="*">
      <xsl:copy-ofselect="."/>
    </xsl:template>
    <xsl:templatename="buildPartnerLinkType">
     <plnk:partnerLinkTypename="/wsdl:definitions/wsdl:service/@namePartnerLinkType">
```

$$< plnk : rolename = "/wsdl : definitions/wsdl : service/@namePortType_provider" portType = "axis2 : /wsdl : definitions/wsdl : service/@namePortType" / >$$
$$< /plnk : partnerLinkType >$$
$$< /xsl : template >$$
$$< /xsl : stylesheet >$$

# C

# Scientific Contributions

**Book Chapters/Journals**

- SISS 2009: Handbook of Research on Service Intelligence and Service Science: Evolutionary Technologies and Challenges. Manel Fredj, Apostolos Zarras, Nikolaos Georgantas, Valérie Issarny. Dynamic Maintenance of Service Orchestrations. Book Chapter In Service Intelligence and Service Science. Dickson K.W. Chiu, Patrick C. K. Hung & Ho-fung Leung Editors. 2009.
- RODIN Project Book 2006 : Rigorous Development of Complex Fault-Tolerant Systems. Apostolos Zarras, Manel Fredj, Nikolaos Georgantas, Valérie Issarny. Engineering Reconfigurable Distributed Software Systems: Issues Arising for Pervasive Computing. Book Chapter In Rigorous engineering of fault tolerant systems. Michael Butler, Cliff Jones, Alexander Romanovsky & Elena Troubitsina Editors. LNCS. 2006.

**International Conference**

- 2008 IEEE Congress on Services (SERVICES 2008)
  Manel Fredj, Nikolaos Georgantas, ValÃĺrie Issarny , Apostolos Zarras. Dynamic Service Substitution in Service-Oriented Architectures . In Proceedings of the IEEE Services 2008-SCC 2008, SOA Industry Summit. July 2008, Hawaii, USA.

**Demonstration**

- Middleware 2007
  Letian Rong, Thomas Wallet , Manel Fredj, Nikolaos Georgantas. Mobile Medical Diagnosis: an m-Health Initiative through Service Continuity in B3G. Accepted in Middleware 2007 Conference (Demo). November 2007, California, USA.

**Workshops**

- Engineering of Software Services for Pervasive Environments 2007 (ESSPE'07)
  Letian Rong, Manel Fredj, ValÃĺrie Issarny, Nikolaos Georgantas. Mobility Management in B3G Networks: a Middleware-based Approach. In Proceedings of the ESSPE Workshop. September 2007, Dubrovnik, Croatia.
- 4th Minnema Workshop 2006
  Manel Fredj, Apostolos Zarras, Nikolaos Georgantas, ValÃĺrie Issarny. Connectivity Loss in Pervasive Computing Environments. In Proceedings of the 4th MiNEMA Workshop. July 2006, Sintra, Portugal.

# Bibliography


[gt4, 2005] (2005). Globus Toolkit 4. Web site, Globus Alliance, Available at http://www.globus.org/alliance/.

[Orc, 2005] (2005). Ochestra. Technical report, BULL SAS-OW2 consortium, Available at .

[PXE, 2005] (2005). PXE: BPEL Process eXecution Engine. Technical report, FiveSight Technologies, Available at http://sourceforge.net/projects/pxe.

[ODE, 2006] (2006). ODE: Orchestration Director Engine. Technical report, Apache, Available at http://ode.apache.org/index.html.

[Act, 2009] (2009). AcriveBpel Community Edition. Technical report, ActiveEndpoints, Available at http://www.activevos.com/community-open-source.php.

[Aalst, 1997] Aalst, W. M. P. v. d. (1997). Verification of workflow nets. In *ICATPN '97: Proceedings of the 18th International Conference on Application and Theory of Petri Nets*, pages 407–426, London, UK. Springer-Verlag.

[Alvisi et al., 1999] Alvisi, L., Rao, S., Husain, S. A., de Mel, A., and Elnozahy, E. (1999). An analysis of communication-induced checkpointing. In *FTCS '99*, page 242, USA. IEEE Computer Society.

[Avizienis et al., 2001] Avizienis, A., Laprie, J., and Randell, B. (2001). Fundamental concepts of dependability.

[Bell, 2008] Bell, M. (2008). *Service-Oriented Modeling: Service Analysis, Design, and Architecture*. Wiley & Sons.

[Ben Mokhtar, 2007] Ben Mokhtar, S. (2007). *Semantic Middleware for Service-Oriented Pervasive Computing*. PhD thesis, Paris VI, Université Pierre et Maris Curie, Paris.

[Benatallah et al., 2004] Benatallah, B., Casati, F., and Toumani, F. (2004). Web service conversation modeling: A cornerstone for e-business automation. *IEEE Internet Computing*, 8(1):46–54.

[Benatallah et al., 2003] Benatallah, B., Casati, F., Toumani, F., and Hamadi, R. (2003). Conceptual modeling of web service conversations. *CaiSE 03: Proceedings of the international Conference on Advanced Information Systems Engineering*, pages 449–467.

[Bentahar et al., 2007] Bentahar, J., Maamar, Z., Benslimane, D., and Thiran, P. (2007). Using argumentative agents to manage communities of web services. In *AINAW '07: Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops*, pages 588–593, Washington, DC, USA. IEEE Computer Society.

[Bernard, 2006] Bernard, G. (2006). Invited paper: Middleware for next generation distributed systems: Main challenges and perspectives. *Database and Expert Systems Applications, International Workshop on*, 0:237–240.

[Berners-Lee et al., 2001] Berners-Lee, T., Hendler, J., and Lassila, O. (2001). *The Semantic Web*. In Scientific American.

[Birman et al., 1991] Birman, K., Schiper, A., and Stephenson, P. (1991). Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314.

[Birman et al., 2004] Birman, K., van Renesse, R., and Vogels, W. (2004). Adding high availability and autonomic behavior to web services. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 17–26, Washington, DC, USA. IEEE Computer Society.

[Bishop, 1995] Bishop, P. (1995). *Software Fault Tolerance*. John Wiley & Sons, Inc., New York, NY, USA.

[Boley et al., 2001] Boley, H., Tabet, S., and Wagner, G. (2001). Design Rationale of RuleML: A Markup Language for Semantic Web Rules.

[Buckley, 2006] Buckley, J. (2006). Conference: From RFID to the Internet of things, Pervasive Networked Systems. Final report, Conference organised by DG Information Society and Media, Networks and Communication Technologies Directorate, CCAB, Brussels.

[Budhiraja et al., 1993] Budhiraja, N., Marzullo, K., Schneider, F. B., and Toueg, S. (1993). The primary-backup approach. pages 199–216.

[Bultan et al., 2003] Bultan, T., Fu, X., Hull, R., and Su, J. (2003). Conversation specification: a new approach to design and analysis of e-service composition. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 403–410, New York, NY, USA. ACM.

[Calore et al., 2007] Calore, F., Lombardi, D., Mussi, E., Plebani, P., and Pernici, B. (2007). Retrieving substitute services using semantic annotations: A foodshop case study. In ter Hofstede, A. H. M., Benatallah, B., and Paik, H.-Y., editors, *Business Process Management Workshops*, volume 4928 of *Lecture Notes in Computer Science*, pages 508–513. Springer.

[Campbell and Randell, 1986] Campbell, R. H. and Randell, B. (1986). Error recovery in asynchronous systems. *IEEE Trans. Softw. Eng.*, 12(8):811–826.

[Caporuscio et al., 2008] Caporuscio, M., Eikerling, H.-J., K.Liotopoulos, F., Moungla, H., Raverdy, P.-G., Toulis, P., Carughi, G. T., and Xinidis, K. (2008). PLATIC Middleware Delivrable 3.3: Assessment and Revision. Report, Available at http://www-c.inria.fr/plastic/dissemination/plastic-reports/public-deliverables.

[Cardelli, 1997] Cardelli, L. (1997). Type systems. In Tucker, A. B., editor, *The Computer Science and Engineering Handbook*, pages 2208–2236. CRC Press.

[Cardelli and Wegner, 1985] Cardelli, L. and Wegner, P. (1985). On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17:471–522.

[CHANDRA et al., 2001] CHANDRA, B., DAHLIN, M., GAO, L., and NAYATE, A. (2001). End-to-end wan service availability.

[Chandy and Lamport, 1985] Chandy, K. M. and Lamport, L. (1985). Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75.

[Chomsky, 1956] Chomsky, N. (1956). Three models for the description of language. *Information Theory, IEEE Transactions on*, 2(3):113–124.

[Cicekli and Yildirim, 2000] Cicekli, N. K. and Yildirim, Y. (2000). Formalizing workflows using the event calculus. In *DEXA '00: Proceedings of the 11th International Conference on Database and Expert Systems Applications*, pages 222–231, London, UK. Springer-Verlag.

[Corcho and Gómez-Pérez, 2000] Corcho, O. and Gómez-Pérez, A. (2000). A roadmap to ontology specification languages. In *EKAW '00: Proceedings of the 12th European Workshop on Knowledge Acquisition, Modeling and Management*, pages 80–96, London, UK. Springer-Verlag.

[DeCandia et al., 2007] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. (2007). Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220.

[Défago et al., 2004] Défago, X., Schiper, A., and Urbán, P. (2004). Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36:2004.

[Derbel et al., 2008] Derbel, B., Mosbah, M., and Gruner, S. (2008). Mobile agents implementing local computations in graphs. In *ICGT '08: Proceedings of the 4th international conference on Graph Transformations*, pages 99–114, Berlin, Heidelberg. Springer-Verlag.

[Dubuis et al., 2004] Dubuis, E., Fornasler, P., and kowalski, P. (2004). Bexee: BPEL Execution Engine. Technical report, Berne University of Applied Sciences, School of Engineering and Information Technology, Available at http://bexee.sourceforge.net/.

[Elnozahy et al., 2002] Elnozahy, E. N. M., Alvisi, L., Wang, Y.-M., and Johnson, D. B. (2002). A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3).

[Fielding, 2000] Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine.

[Flé and Roucairol, 1985] Flé, M.-P. and Roucairol, G. (1985). A language theoretic approach to serialization problem in concurrent systems. In *FCT '85: Fundamentals of Computation Theory*, pages 128–145, London, UK. Springer-Verlag.

[Floyd, 1967] Floyd, R. W. (1967). Assigning meanings to programs. In *Proc. Sympos. Appl. Math., Vol. XIX*, pages 19–32. Amer. Math. Soc., Providence, R.I.

[Fredj et al., 2008] Fredj, M., Georgantas, N., Issarny, V., and Zarras, A. (2008). Dynamic service substitution in service-oriented architectures. pages 101–104.

[Fredj et al., 2006] Fredj, M., Zarras, A., Georgantas, N., and Issarny, V. (2006). Adaptation to connectivity loss in pervasive computing environments. *Proceedings of the 4th MiNEMA Workshop*, 20.

[Fredj et al., 2009] Fredj, M., Zarras, A., Georgantas, N., and Issarny, V. (2009). *Dynamic Maintenance of Service Orchestrations.*

[Gaudel et al., 2003] Gaudel, M.-C., Issarny, V., Jones, C., Kopetz, H., Marsden, E., Moffat, N., Paulitsch, M., Powell, D., Randell, B., Romanovsky, A., Stroud, R., and Taiani, F. (2003). Final version of the DSoS conceptual model. *DSoS Project (IST-1999-11585) Deliverable CSDA 1.*

[Glinz, 2007] Glinz, M. (2007). On non-functional requirements. *IEEE International Conference on Requirements Engineering*, 0:21–26.

[Gómez-Pérez and Corcho, 2002] Gómez-Pérez, A. and Corcho, O. (2002). Ontology specification languages for the semantic web. *IEEE Intelligent Systems*, 17(1):54–60.

[Grossmann et al., 2006] Grossmann, G., Schrefl, M., and Stumptner, M. (2006). Verification of business process integration options. In Dustdar, S., Fiadeiro, J. L., and Sheth, A. P., editors, *Business Process Management*, volume 4102 of *Lecture Notes in Computer Science*, pages 432–438. Springer.

[Group, 2001] Group, I. S. (2001). Web Services Flow Language (WSFL 1.0). Technical report.

[Guo et al., 2005a] Guo, R., Chen, D., and Le, J. (2005a). Matching semantic web services across heterogeneous ontologies. In *CIT '05: Proceedings of the The Fifth International Conference on Computer and Information Technology*, pages 264–268, Washington, DC, USA. IEEE Computer Society.

[Guo et al., 2005b] Guo, R., Le, J., and Xia, X. (2005b). Capability matching of web services based on owl-s. *Database and Expert Systems Applications, 2005. Proceedings. Sixteenth International Workshop on*, pages 653–657.

[Hackmann, 2006] Hackmann, G. (2006). Sliver. Technical report, Washington university St Louis, Available at http://mobilab.cse.wustl.edu/projects/sliver/.

[Hau et al., 2005] Hau, J., Lee, W., and Darlington, J. (2005). A semantic similarity measure for semantic web services. In *Web Service Semantics Workshop at WWW (2005*.

[Helal et al., 1996] Helal, A. A., Bhargava, B. K., and Heddaya, A. A. (1996). *Replication Techniques in Distributed Systems*. Kluwer Academic Publishers, Norwell, MA, USA.

[Hendler et al., 1990] Hendler, J., Tate, A., and Drummond, M. (1990). Service adaptation through trace inspection. *AAAI*, pages 61–77.

[Hoare, 1969] Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580.

[Hoare, 1971] Hoare, C. A. R. (1971). Procedures and parameters: An axiomatic approach. *Engeler, E. (Ed.), Lecture Notes in Mathematics*, 188:102–116.

[IETF, 2000] IETF (1997,2000). HTTP State Management Mechanism. Rfc 2109 and rfc 2965, IETF, Available at http://tools.ietf.org/html/rfc2109 and http://tools.ietf.org/html/rfc2965.

[Issarny et al., 2007] Issarny, V., Caporuscio, M., and Georgantas, N. (2007). A perspective on the future of middleware-based software engineering. In *FOSE '07: 2007 Future of Software Engineering*, pages 244–258, Washington, DC, USA. IEEE Computer Society.

[Issarny et al., 2005] Issarny, V., Sacchetti, D., Tartanoglu, F., Sailhan, F., Chibout, R., Levy, N., and Talamona, A. (2005). Developing ambient intelligence systems: A solution based on web services. *In Automated Software Engineering*, 12.

[Joseph et al., 1995] Joseph, A., deLespinasse, A., Tauberand, J., Gifford, D., and Kaashoek, M. (1995). Rover: a toolkit for mobile information access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*.

[Kaashoek and Tanenbaum, 1991] Kaashoek, M. and Tanenbaum, A. (1991). Group communication in the amoeba distributed operating system. *Distributed Computing Systems, 1991., 11th International Conference on*, pages 222–230.

[Kaiser and Pu, 1992] Kaiser, G. E. and Pu, C. (1992). Dynamic restructuring of transactions. In *Database Transaction Models for Advanced Applications, chapter 8*, pages 265–295. Morgan Kaufmann.

[König et al., 2008] König, D., Lohmann, N., Moser, S., Stahl, C., and Wolf, K. (2008). Extending the compatibility notion for abstract ws-bpel processes. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 785–794, New York, NY, USA. ACM.

[Krafzig et al., 2004] Krafzig, D., Banke, K., and Slama, D. (2004). *Enterprise SOA: Service-Oriented Architecture Best Practices (The Coad Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

[Kramer and Magee, 1990] Kramer, J. and Magee, J. (1990). The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306.

[Lamport, 1978] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565.

[Laprie et al., 1992] Laprie, J., Avizienis, A., and Kopetz, H., editors (1992). *Dependability: Basic Concepts and Terminology*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[Li and Fuchs, 1990] Li, C.-C. and Fuchs, W. (1990). Catch-compiler-assisted techniques for checkpointing. *Fault-Tolerant Computing, 1990. FTCS-20. Digest of Papers., 20th International Symposium*, pages 74–81.

[Liskov and Wing, 1994] Liskov, B. H. and Wing, J. M. (1994). A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841.

[Maamar et al., 2008] Maamar, Z., Sheng, Q. Z., and slimane, D. B. (2008). Sustaining web services high-availability using communities. In *ARES*, pages 834–841. IEEE Computer Society.

[Marian et al., 2008] Marian, T., Balakrishnan, M., Birman, K., and van Renesse, R. (2008). Tempest: Soft state replication in the service tier. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'08)*, Anchorage, Alaska, USA.

[Microsoft, 2001] Microsoft (2001). XLANG - Web Services for Business Process Design. Technical report, Available at http://xml.coverpages.org/XLANG-C-200106.html.

[Mokhtar et al., 2008a] Mokhtar, S. B., Bianco, S., Georgantas, N., Issarny, V., and Thomson, G. (2008a). iCOCOA : Inria's COnversation-based service COmposition in pervAsive computing environments. Technical report, Inria, Project-team Arles, Available at http://www-rocq.inria.fr/arles/download/iCOCOA/index.html.

[Mokhtar et al., 2007] Mokhtar, S. B., Georgantas, N., and Issarny, V. (2007). Cocoa: Conversation-based service composition in pervasive computing environments with qos support. *J. Syst. Softw.*, 80(12):1941–1955.

[Mokhtar et al., 2008b] Mokhtar, S. B., Preuveneers, D., Georgantas, N., Issarny, V., and Berbers, Y. (2008b). Easy: Efficient semantic service discovery in pervasive computing environments with qos and context support. *J. Syst. Softw.*, 81(5):785–808.

[Netscape, 1999] Netscape (1999). Persistent client state - HTTP cookies - Preliminary specification. Technical report, Netscape, Available at http://web.archive.org/web/20070805052634/http://wp.netscape.com/newsref/std/cookie_spec.html.

[OASIS, 2006a] OASIS (2006a). Web Services Resource 1.2 (WS-Resource). Technical report, OASIS, Available at http://docs.oasis-open.org/wsrf/wsrf-ws_resource-1.2-spec-os.pdf.

[OASIS, 2006b] OASIS (2006b). Web Services Resource Framework (WSRF) v1.2 Specification. Technical report, OASIS Standard, Available at http://www.globus.org/wsrf/.

[OASIS, 2006c] OASIS (2006c). Web Services Resource Properties 1.2 (WS-ResourceProperties). Technical report, OASIS, Available at http://docs.oasis-open.org/wsrf/wsrf-ws_resource_properties-1.2-spec-os.pdf.

[OASIS, 2007] OASIS (2007). Web Services Business Process Execution Language. Technical report, OASIS Standard, Available at http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html.

[OMG, 2001] OMG (2001). Model Driven Architecture (MDA). Technical report, OMG, Available at http://www.omg.org/mda/specs.htm.

[Osrael et al., 2006] Osrael, J., Froihofer, L., and Goeschka, K. M. (2006). What service replication middleware can learn from object replication middleware. In *MW4SOC '06: Proceedings of the 1st workshop on Middleware for Service Oriented Computing (MW4SOC 2006)*, pages 18–23, New York, NY, USA. ACM.

[Paolucci et al., 2002] Paolucci, M., Kawamura, T., Payne, T. R., and Sycara, K. P. (2002). Semantic matching of web services capabilities. In Horrocks, I. and Hendler, J. A., editors, *1rst International Semantic Web Conference*, volume 2342 of *Lecture Notes in Computer Science*, pages 333–347. Springer.

[Papazoglou, 2003] Papazoglou, M. P. (2003). Service -oriented computing: Concepts, characteristics and directions. In *WISE '03: Proceedings of the Fourth International Conference on Web Information Systems Engineering*, page 3, Washington, DC, USA. IEEE Computer Society.

[Papazoglou and Dubray, 2004] Papazoglou, M. P. and Dubray, J. J. (2004). A survey of web service technologies,technical report dit -o4 -058. University of Trento, Department of Information and Communication Technology.

[Papazoglou and Georgakopoulos, 2003] Papazoglou, P. and Georgakopoulos, D., editors (2003). *Service-oriented computing*, volume 46. In Communications of the ACM.

[Peti, 2002] Peti, P. (2002). The concepts behind time, state, component, and interface - a literature survey. In *Survey*, Vienna.

[Poledna, 1996] Poledna, S. (1996). *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism*. Kluwer Academic Publishers, Norwell, MA, USA.

[Powell et al., 1991] Powell, D., Bey, I., and Leuridan, J., editors (1991). *Delta Four: A Generic Architecture for Dependable Distributed Computing*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[Randell, 1975] Randell, B. (1975). System structure for software fault tolerance. In *Proceedings of the international conference on Reliable software*, pages 437–449, New York, NY, USA. ACM.

[Rappaport, 2001] Rappaport, T. (2001). *Wireless Communications: Principles and Practice (2nd Edition)*. Prentice Hall PTR.

[Rong and Caporuscio, 2008] Rong, L. and Caporuscio, M. (2008). Middleware Developer's Guide: Multi-Radio Device Management Layer. Technical report, INRIA, Available at http://gforge.inria.fr/frs/?group_id=699.

[Rong et al., 2007a] Rong, L., Fredj, M., Issarny, V., and Georgantas, N. (2007a). Mobility management in b3g networks: a middleware-based approach. In *ESSPE '07: International workshop on Engineering of software services for pervasive environments*, pages 41–45, New York, NY, USA. ACM.

[Rong et al., 2007b] Rong, L., Wallet, T., Fredj, M., and Georgantas, N. (2007b). Mobile medical diagnosis: an m-health initiative through service continuity in b3g. In *Middleware '07: Proceedings of the 8th ACM/IFIP/USENIX international conference on Middleware*, pages 1–2, New York, NY, USA. ACM.

[Rosa et al., 2005] Rosa, F. D., Malizia, A., and Mecella, M. (2005). Disconnection prediction in mobile ad hoc networks for supporting cooperative work. *IEEE Pervasive Computing*, 4(3):62–70.

[Ruggaber and Seitz, 2001] Ruggaber, R. and Seitz, J. (2001). A transparent network handover for nomadic corba users. In *ICDCS '01: Proceedings of the The 21st International Conference on Distributed Computing Systems*, page 499, Washington, DC, USA. IEEE Computer Society.

[Salatge and Fabre, 2007] Salatge, N. and Fabre, J.-C. (2007). Fault Tolerance Connectors for Unreliable Web Services. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 51–60.

[Satyanarayanan, 2001] Satyanarayanan, M. (2001). Pervasive computing: Vision and challenges. *IEEE Personal Communications*, 8:10–17.

[Sens, 1995] Sens, P. (1995). The performance of independent checkpointing in distributed systems. *hicss*, 00:525.

[Singh and Huhns, 2005] Singh, M. P. and Huhns, M. N. (2005). *Service-Oriented Computing: Semantics, Processes, Agents*. John Wiley and Sons.

[Strom and Yemini, 1985] Strom, R. and Yemini, S. (1985). Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3).

[Su et al., 2008] Su, J., Bultan, T., Fu, X., and Zhao, X. (2008). Towards a theory of web service choreographies. pages 1–16.

[Thompson, 1991] Thompson, S. (1991). *Type Theory and Functional Programming.*

[Van Der Aalst et al., 2003] Van Der Aalst, W., Ter Hofstede, A., Kiepuszewski, B., and Barros, A. (2003). Workflow patterns. *Distrib. Parallel Databases*, 14(1):5–51.

[Verbeek, 2001] Verbeek, H. M. W. (2001). Diagnosing workflow processes using woflan. *The Computer Journal*, 44.

[W3C, 1999] W3C (1999). XSL Transformations (XSLT). Technical report, W3C Standard, Available at http://www.w3.org/TR/xslt.

[W3C, 2004a] W3C (2004a). OWL-S: Semantic Markup for Web Services. Technical report, W3C Standard, Available at http://www.w3.org/Submission/OWL-S/.

[W3C, 2004b] W3C (2004b). OWL Web Ontology Language. Technical report, W3C Standard, Available at http://www.w3c.org/TR/owl-ref.

[W3C, 2004c] W3C (2004c). SWRL: A Semantic Web Rule Language Combining OWL and RuleML. Technical report, W3C, Available at http://www.w3.org/Submission/SWRL/.

[W3C, 2004d] W3C (2004d). Web Services Addressing (WS-Addressing). Technical report, W3C standard, Available at http://www.w3.org/Submission/ws-addressing/.

[W3C, 2007a] W3C (2007a). Semantic Annotations for WSDL and XML Schema. Technical report, W3C Standard, Available at http://www.w3.org/TR/sawsdl/.

[W3C, 2007b] W3C (2007b). Simple Object Access Protocol (SOAP) 1.2. Technical report, W3C Standard, Available at http://www.w3.org/TR/soap/.

[W3C, 2007c] W3C (2007c). Web Services Description Language (WSDL) Version 2.0. Technical report, W3C Standard, Available at http://www.w3.org/TR/wsdl20/.

[W3C, 2007d] W3C (2007d). XML Path Language (XPath) 2.0. Technical report, W3C Standard, Available at http://www.w3.org/TR/xpath20/.

[Wolf, 1998] Wolf, T. (1998). *Replication of Non-Deterministic Objects.* PhD thesis.

[Wombacher et al., 2004] Wombacher, A., Fankhauser, P., and Neuhold, E. (2004). Transforming bpel into annotated deterministic finite state automata for service discovery. In *ICWS '04: Proceedings of the IEEE International Conference on Web Services*, page 316, Washington, DC, USA. IEEE Computer Society.

[Wombacher et al., 2005] Wombacher, A., Mahleko, B., and Neuhold, E. J. (2005). Ipsi-pf - a business process matchmaking engine based on annotated finite state automata. *Inf. Syst. E-Business Management*, 3(2):127–150.

[Yang, 1997] Yang, Q. (1997). *Intelligent Plannning - A Decomposition and Abstraction Based Approach*.

[Zarras et al., 2006] Zarras, A., Fredj, M., Georgantas, N., and Issarny, V. (2006). Engineering reconfigurable distributed software systems: Issues arising for pervasive computing. Lecture Notes in Computer Science. Springer.

[Zhang et al., 2006] Zhang, X., Hiltunen, M. A., Marzullo, K., and Schlichting, R. D. (2006). Customizable service state durability for service oriented architectures. In *EDCC '06: Proceedings of the Sixth European Dependable Computing Conference*, pages 119–128, Washington, DC, USA. IEEE Computer Society.

[Zheng and Lyu, 2008] Zheng, Z. and Lyu, M. (2008). Ws-dream: A distributed reliability assessment mechanism for web services. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'08)*, Anchorage, AK.

[Zhong et al., 2002] Zhong, J., Zhu, H., Li, J., and Yu, Y. (2002). *Conceptual Graph Matching for Semantic Search*.