Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Mestrado em Engenharia Informática

A Middleware for Service Oriented Computing in Dynamic Environments

Danilo Manmohanlal (28068)

Lisboa
(2010)

Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

A Middleware for Service Oriented Computing in Dynamic Environments

Danilo Manmohanlal (28068)

Orientador: Prof. Doutor Herve Paulino

Dissertação apresentada na Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa para a obtenção do Grau de Mestre em Engenharia Informática.

Lisboa
(2010)

To all my friends

Acknowledgements

Before I begin introducing this thesis, I would like to recognize the people that supported me during my academic time. First, I thank my advisor Hervé Paulino, he is responsible for proposing this thesis to me and for the same being completed successfully. I want to acknowledge as well his continuous support, encouragement, and advice throughout my dissertation.

I would also like to thank:

- CITI for the opportunity to work in this project with a scholarship.
- Ricardo Dias and Emanuel Couto for their support in understanding the Polyglot framework.
- Professor João Lourenço for his support and motivation in the final days of the project.
- To all professors that helped me during my course since day one.
- Pedro Bernardo and Bruno Teixeira for pulling an all nighter with me in the last day to deliver the dissertation.
- Last, but not least, it is a pleasure to thank those who made this thesis possible: Hélio Dolores, André Costa, Pedro Ribeiro, Marcelo Martins, Farnel, Vera Viegas, Pedro Fernandes, Bruno Félix, Maria Café, Nuno Luís, João Cartaxo, Rui Domingues, Ana Martins, Vitor Varela, Tiago Borges, João Ferreira, Rui Mourato, Ricardo Silva, Rui Rosendo, Andreia Tomás, Eduardo Evangelista, Pedro Pereirinha, Amável Santo, Bruno Alves, Marisa Fernandes, Rita Pires, Inês Andrade, Noel Moreira, João Gomes, Ana Maria Pinto, João Ruivo, Luis Miranda, Ângelo Sarmento, A Outra, (Un)Hapiness, Sara S. ... and everyone that helped me during my course and project.

Summary

The last years have witnessed a convergence on the SOA paradigm from industrial processes enterprises (like logistics or manufacturing), using standards for data and communication. SOA promotes reusability, interoperability and loose-coupling of applications.

The convergence towards SOA shows that we are leading to an infrastructure composed by several heterogeneous devices, the "Internet of Things". In this infrastructure everything can be abstracted as a service, such as household appliances, mobile devices, or industrial machinery. It is expected that this trend will continue, and as these devices interoperate in service composition, new functionalities may be discovered.

Existing approaches for service composition, namely in business processes, are too bound to BPEL. Several alternatives and extensions of BPEL have been developed, but they feel more like patches than solutions. In this context SeDeUse [29] model has been proposed as an exercise to define new language constructs promoting a separation from service awareness and use. The model also relies on a middleware layer to support the execution of the application in dynamic environments.

The goal of this dissertation is to instantiate the SeDeUse model in a widely used programming language in order to provide a framework for its assessment and for its future development. The work consists on implementing a concrete syntax for the model, a compilation process, and a middleware layer. The syntax contains the new language constructs that are integrated in the hosting language. The compilation process is responsible for service definition and code generation. Finally, the middleware acts as a support for the application (generated code) requests.

We have seamlessly integrated SeDeUse in the Java programming language and developed a functional prototype. To assess the prototype capability, three scenarios were developed in which we demonstrated that our implementation provides a new, and simpler, approach for abstracting resources as services.

Keywords: SOA, Service Composition, Abstractions, Services, Internet of Things.

Sumário

Nos últimos anos empresas da indústria como, logística ou produção têm vindo a convergir no uso do paradigma SOA ao usar normas para dados e comunicação. Este paradigma promove a reutilização, interoperabilidade e acoplamento fraco de aplicações.

Esta convergência, por parte de várias empresas, em relação a SOA mostra que estamos a caminhar para uma infraestrutura composta por vários dispositivos heterogéneos, a "Internet of Things". Nesta infraestrutura é possível abstrair tudo em serviços tais como, eletrodomésticos, dispositivos móveis ou máquinas industriais. É esperado que esta tendência continue e que os serviços possam trabalhar em conjunto com outros numa composição de serviços de modo a fornecer novas funcionalidades.

O trabalho que existe nesta área de composição de serviços, nomeadamente em processos de negócio, está muito ligado à tecnologia BPEL. As alternativas ou extensões que existem, ao BPEL, não se configuram como uma solução plena. Neste contexto foi proposto o modelo SeDeUse [29] como um exercício de definição de novas construções que promove uma separação sobre a definição do serviço e o seu uso. O modelo também apresenta uma camada intermédia que funciona de suporte à aplicação durante a sua execução.

O objetivo desta dissertação é instanciar o modelo SeDeUse numa linguagem de programação de uso geral de modo a obter uma plataforma em que se possa aferir a sua funcionalidade e servir para desenvolvimento futuro. O trabalho consiste em implementar uma sintaxe concreta para o modelo, um processo de compilação e uma camada intermédia. A sintaxe contém as novas construções que são integradas na linguagem base. O processo de compilação é responsável pela definição do serviço e geração de código. A camada intermédia funciona como suporte aos pedidos da aplicação.

O modelo foi integrado de uma forma harmoniosa na linguagem Java e foi desenvolvido um protótipo funcional. Para aferir da funcionalidade e expressividade do protótipo foram desenvolvidos três cenários de teste. Com a realização dos testes

podemos concluir que a implementação deste modelo fornece uma nova abordagem para o uso de recursos do dia a dia como serviços de uma forma simples e intuitiva.

Palavras-chave: SOA,Composição de Serviços, Abstracções, Serviços, Internet of Things.

Contents

List of Figures

List of Tables

1

Introduction

Service-Oriented Architecture (SOA) [23] is currently the most popular paradigm for developing distributed applications over the Web. This architectural style emphasizes that application functionalities are provided as services, a self-contained and independent set of functions available for use through an interface.

SOA promotes reusability, interoperability and the loose-coupling of applications. For instance, previous legacy systems that worked independently can now be wrapped as services and made available through interfaces over the network. This feature improves interoperability between heterogeneous applications and technologies. In the business world SOA had a major impact improving business-to-business (B2B) collaboration. Previous business systems were tightly-coupled, any change in one system would cause changes in several dependent subsystems [5]. B2B applications can now provide a better offer in services to end-users and also to collaborating enterprises.

Service-Oriented Computing (SOC) [28] is a paradigm that uses services to build distributed applications in a rapid, low cost and simple composition way, in homogeneous and heterogeneous environments. SOC utilizes services to represent autonomous and platform independent entities, i.e., entities that can be used by any framework. The promise of SOC is to achieve a world where services can be built and integrated quickly into a loosely-coupled network of services.

Web Services are the most promising technology that implements the SOC concept, they are fundamental for developing and executing services distributed through the web and available through interfaces. This technology revolutionize the world of service integration and overcomes previous limitations in application integration such as,

interoperability and firewall problems [17].

Service composition is one the benefits of the SOC technology. It allows the construction of complex compound applications from single services and from different locations. Business Process Execution Language for Web Services (BPEL) [20] is the technology community standard for composing business processes. Its main goal is to provide an environment where business processes are composed in a flexible manner, by the means of orchestration. However, the existing coordination models in SOC, choreography and orchestration have several limitations of which we emphasize, target mostly static environments where locations of services are already known. This does not conform to SOC use in dynamic environments.

Another important aspect is mobile computing in service use. Today almost every person has a mobile device (mobile phones, pda's, laptops). The access of services for mobile devices has increased the range of users that can benefit from their use. For instance, a user that cannot attend a conference in person, with the use of a mobile device and Internet he can be present at the reunion [2]. Mobile devices can self adjust to surrounding environment, in the previous example if the network conditions get worse then the device can adjust voice and image quality to a lower level but still maintain connectivity. Mobile phones with GPS support can assist a person in finding a certain location or place, a pda can be used to schedule meetings, and so on. The options for service use are endless.

Today with Internet available everywhere and in almost every computational device many possibilities come up. Services that are deployed across the world can be used by users located in different locations. For example, a user that intends to print the contents of a message or of a file that is reading in his mobile device can search the surrounding network for a printer that matches his preferences. In a reunion, if relevant information is in a mobile device, then it can search for a display in the room and pass the information for a better visualization. These two examples show that publicly available resources can be abstracted as services. Every operation performed by users in mobile devices or any kind of computational device can be abstracted as a service, we are moving towards an Internet of Things [37]. Therefore this area of describing abstractions for services and business processes accessible through mobile devices in dynamic environments is a current research topic.

1.1 Problem Statement and Work Goals

BPEL is a very popular orchestration framework between and within enterprises, its popularity comes from the ability to develop business processes and define in which

order the services are invoked, making business applications more flexible.

Business processes are nowadays more dynamic and enterprises need to adapt to changes in the environment. Any modification and improvement in the composition process requires much time spent by developers in rearranging the composition. BPEL presents itself as the standard for Web Service composition, it pretends to achieve business process integration and define in which order the services are executed, making business applications more flexible. Although BPEL has much popularity in the business world, when working in dynamic environments the framework needs improvement. In BPEL interaction between business process and services or clients are defined using *partner links*, that are defined at design time. This is one of BPEL limitations due to its static logic of composition, other problems are now presented:

Data manipulation: It is not possible to create variables dynamically, they are declared declared at design time and at process level. BPEL cannot tell if two variables are composed of the same primitive type, this reduces BPEL flexibility [13].

Fixed participants: BPEL assumes that the number of participants in the process is known at design time and this number cannot be changed, this restrains BPEL use in dynamic settings [15].

Dynamic binding: In BPEL links from business processes to concrete services are defined at design time, this results in so the service tightly-coupled applications between providers and consumers. If the partner service address changes the process fails [12].

Failure recovery: The framework allows to define simple recovery statements but for more complex recovery plans such as rollbacks, alternative execution models or service replacement it is not possible [24].

Dynamic environment: Business processes evolve in time and new services that integrate the composition should not require significant changes in the process workflow [40].

However, in regard these problems there is some work done, namely BPEL extensions and alternatives have been developed. These approaches that are discussed in more detail in Subsection 2.2.1.

The current composition schemes used for orchestration and choreography are mostly designed to work in business processes with a static environment, but today communications and services are dynamic. Alternatives and extensions to BPEL that were mentioned feel more like patches than solutions. In this context the SeDeUse [29] was

proposed as an exercise to define new language constructs and a software layer for service use in dynamic environments. The main goal for this thesis is to instantiate the SeDeUse model in a widely used programming language in order to provide a framework for its assessment and for its future development. The work consists on implementing a concrete syntax for the model, a compilation process, and a middleware layer.

1.2 Proposed Solution

The focus of this thesis is to develop an initial implementation of the framework. The SeDeUse model hides from user the SOC characteristics in dynamic environments by using a middleware layer between the application and services. This layer provides dynamic discovery, management and acquisition of services.

The model separates functional from non-functional requirements. It uses a two layer approach based on studied frameworks such as, WS-Binder [10], MASC [12], and AOP [40]. The two layers are:

- **Service Awareness Layer (SAL)**, defines the *kind* of services to be used on the application, i.e., services that are discovered in the network. Also attributes are defined to constrain the search space.

- **Service Use Layer (SUL)**, defines a simple coordination model for service use

Listing 1.1 presents an example of a SAL definition for a printer. The use of attributes such as, **colors**, **paper** and **type**, will constrain the search to providers that have black and white printers with the ability to print on letter paper.

This printer kind is identified by the *Printer* identifier, which can be used in the SUL layer, as presented in Listing 1.2. In this example *MyPrinter* is abstracted on both the **doc** parameter as well as in a provider of *Printer*.

Listing 1.1: A printer

```
Printer {
  colors = "blackandwhite",
  paper = "letter"
}
```

Listing 1.2: Using a service

```
use Printer in MyPrinter(String doc) { Printer.print(doc) }
new MyPrinter(``myDocument'')
```

SeDeUse does not define a complete language, thus it must rely on a hosting language. Our work will focus on the instantiation of the model in the Java language to carry computational work and interact with the middleware. A pre-processing step is required to translate SAL and SUL definitions into Java code, as illustrated in Figure 1.1.

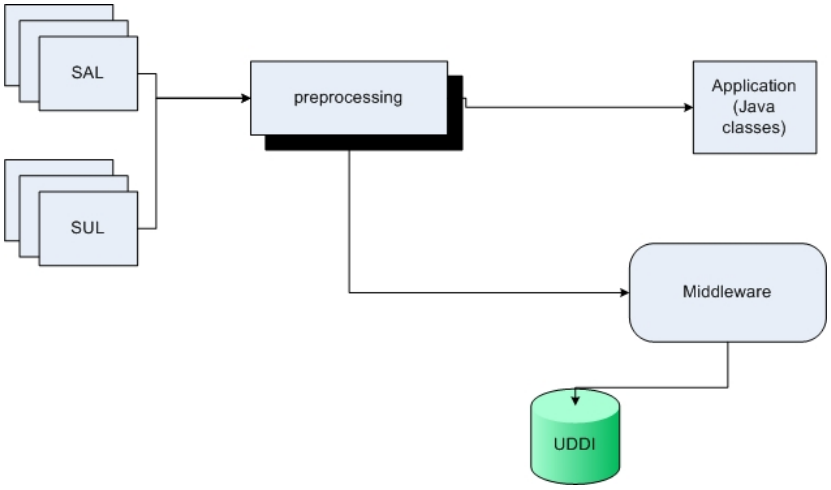

Figure 1.1: Compilation process

Note that during the preprocessing stage, a connection to the middleware can be performed for service discovery purposes. This approach allows for a early and late binding technique. The generated Java code will be able to interact with the middleware layer, also to be implemented in Java language. It will be a software layer with the ability to manage service bindings and interaction, providing an interface for the upper layer applications in order to respond to application requests. The role of the middleware layer is illustrated in Figure 1.2.

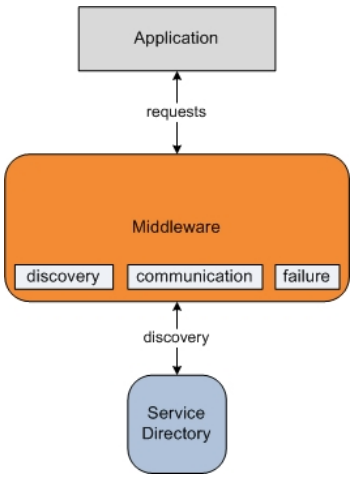

Figure 1.2: Middleware process

1.3 Contributions

This dissertation provides the following contributions.

Sedj: The concrete instantiation of the SeDeUse model in the Java language, which we named as Sedj. It expresses the definition of a concrete syntax for the model's constructs in Java.

Prototype: The development of an operational prototype that includes:

- **SAL/SUL:** The translation of SAL and SUL components to the Java language.
- **Middleware:** The implementation of a software layer able to define interfaces for the lower and upper layer, generation of proxies for service invocation, communication with service registry, and handle dynamic nature of the execution environment(ex: service replacement, matching).
- **Interface Mapping:** The implementation of a mechanism for interface mapping between services.

Evaluation: Test and analysis of the Sedj prototype in terms three possible real world scenarios.

1.4 Document Outline

The remainder of this dissertation is organized has follows. Chapter 2 describes the related work that most closely relates to the objectives of this thesis. Chapter 3 describes the SeDeUse model for service oriented computing in dynamic environments. Chapter 4 describes Sedj, the developed implementation of the SeDeUse model. In Chapter 5, we present the evaluation process of the implemented framework. Finally in Chapter 6 we present the conclusions of this dissertation and a list of tasks to achieve in future work.

2

Related Work

In this chapter we present the work that most closely relates to the scope of this thesis. Section 2.1 introduces the paradigm of service-oriented architectures (SOA) and an analysis to the technology that best implements the paradigm, Web Services, followed by its use in Java technology. Next, we present the notion of service composition, orchestration and choreography, focusing mainly in the use of BPEL technology in the business world.

Section 2.2 approaches the use of Web Services in dynamic environments, also the support and limitations of BPEL in this environment. Then, a comparison is performed regarding several approaches that try to cover BPEL limitations.

Finally, in Section 2.3 we introduce the notion of Web Services with attributes. TModels and Semantic Web have been used together to provide semantic description of Web Services, thus improving its discovery and selection process.

2.1 Service-Oriented Architecture

We begin this section by introducing the concept of Component-Based Architecture (CBA) [41], an architecture paradigm that has two main objectives: to simplify the design process of a system/application and to reduce software development time. This architecture is the result of an improvement on software quality, promoting loose-coupling and component reusability.

The architecture of CBA is composed by software components and interfaces. The former represents a system, that have well defined interfaces for communication with

other software components. The latter defines the role of software components and a mean of communication with external components. Examples of technologies based on CBA are: Common Object Request Broker Architecture (CORBA) [9], Java Beans [19], Java Remote Method Invocation (RMI) [35] and Component Object Model (COM) [8].

In recent years software architecture evolution has been driven by the necessity of improving business service and interoperability between enterprises. The notion of service led to the evolution of CBA to Service Oriented Architectures (SOA) [23], an architectural model that is defined by a set of interacting services.

In this architectural model, a system is defined by a set of interacting services, these are composed by software components. SOA aims to increase loose-coupling and service reusability, and, in that process, promote separation of concerns between interface and implementation. With this new architectural style, enterprises can develop, interconnect and maintain their applications and services on a cost efficient way.

A service-oriented architecture contains the following particular characteristics:

- Services are discoverable through a central registry and dynamically bound;
- Services are independent and modular;
- Services emphasize interoperability;
- Services are loosely-coupled;
- Services have a network-addressable interface;
- Services are location transparent;
- Services are able to be composite;
- SOA are able to recover independently from errors;

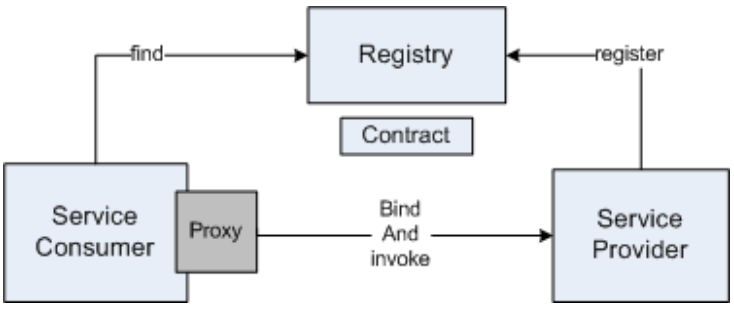

Figure 2.1: The "find-bind-execute" mechanism

The SOA architecture is illustrated in Figure 2.1, and is defined by the following six entities [23]:

Service Contract/Interface: the contract is a definition on how the service consumer communicates (request and response from service) with the service provider. It may also refer to the quality of service (QoS) levels, the non-functional aspects of the service.

Service Provider: is the service itself, accepts and executes requests from consumers, the service contact is published, by the provider, in the registry.

Service Registry: it is a service directory where services can be found. The directory keeps service contracts from service providers and those contracts are accessible to service consumers.

Service Consumer: can be a client, a service, or an application that needs a service. The consumer begins by searching services in the registry, then binds to a selected one and executes it by sending a request according to the contract.

Service Proxy: this entity is not required, but can improve performance by keeping a cache of remote references and other data information. The proxy is a service that is supplied by the service provider and is a reference to the remote object, the consumer executes the remote object functions by calling them on its service proxy.

Service Lease: a lease is granted to the service consumer for a specific time, during that time the contract is valid, when the time runs out a new lease needs to be requested by the consumer. This is necessary when services need to keep state information about the service provider and consumer.

Although SOA paradigm presents itself as a reliable approach for service integration, researchers continue working on improving SOA abstraction, that led to Service Component Architecture (SCA) [3]. The motivation for SCA is to abstract the middleware programming model dependencies from the business logic, reducing complexity for developers when using middleware APIs. This approach is not relevant for the work but is worth mentioning. SCA supports several technologies, service interfaces can be defined using a Web Service Description Language or Java. Service components can be built with Enterprise Java Beans, Spring Beans, CORBA components or with programming languages such as, Java, C++ and PHP. Allows adaptation of existing applications and data using a SOA abstraction. One important feature of SCA is the freedom of using the most suitable implementation. The implementation is servant of the business process and not the opposite.

2.1.1 Service Oriented Computing

Service Oriented Computing (SOC) [28] is a programming paradigm that uses services as the main support to develop distributed application in a rapid, low cost and easy composition way, even in heterogeneous environments.

SOC uses the concept of service to describe an autonomous and platform-independent entity. Services can be described, published and discovered. Any developed code or a system component can be reused by converting it to a service and making it available through the network.

One great advantage of adopting SOC technology is the ability of developing compound applications that previously worked in a isolated way. Enterprise Resource Planning (ERP), Customer Relationship Management (CRM), Supply Chain Management (SCM) and other legacy systems can now be transformed to service architectures, therefore improving productivity on applications whose information resides on different systems.

The promising vision of SOC is to reach a condition where services can cooperate with each other, resulting on less effort for application development, thus creating flexible processes and agile applications that can help to expand and create more organizations.

Web Services are the promising technology based on the SOC concept, they provide the base for developing and executing processes that are distributed over the network and available through interfaces. Legacy systems and other applications previously developed are now accessible through the Internet, making Web Services so appealing nowadays.

2.1.2 Web Services

Web Services are one of the technologies available today that allow the implementation of SOA. There are several definitions for Web Services, this is taken from the W3C [1] website:

> "A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards."

[1] W3C is a consortium that works to develop standards for the web, www.w3c.org

In loosely-coupled systems any change in components rarely makes an impact on the whole system. The use of web technologies allows interoperability between enterprises working in different frameworks. Web Services are accessible everywhere with a connection to the web, they reduce heterogeneity and the process of application integration is easier. In regard to the six SOA concepts mentioned in the beginning of this section, Web Services do not support the contract lease.

Web Services try to solve some of the problems found in former technologies that were used to develop CBA/SOA applications (RMI, CORBA, COM/DCOM), namely:

Interoperability: Each enterprise uses their own message format. By using XML as a standard, interoperability can be achieved.

Firewall: Former technologies suffer of communication problems, ports not allowed were used. Web Services use HTTP through port 80.

The problem with firewall permissions can be solved with HTTP tunneling technique [17], this approach is used to encapsulate network protocols using HTTP protocol. The motivation is to overcome some limitations that can appear in locations where connectivity is limited or restrict, but it has disadvantages. HTTP tunneling use is inefficient, significant bandwidth waste on use. This technique has been used on RMI to overcome firewalls.

Web Services Architecture

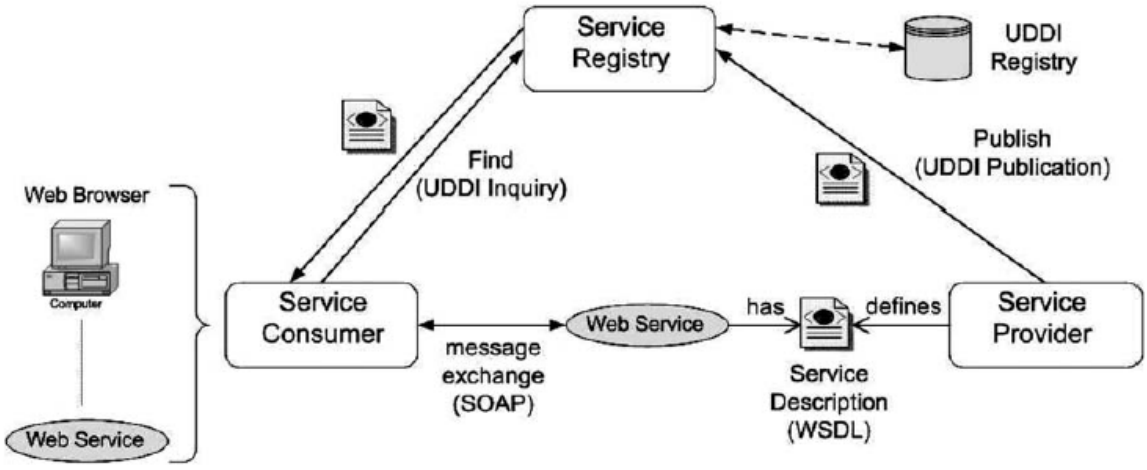

Figure 2.2: Web Services Architecture

This architecture is supported by three specifications: WSDL [48] for service description, UDDI [39] for service discovery and SOAP [36] for transport. It also has

three roles: service *consumer*, *provider* and *registry*. The service consumer inquires the registry (UDDI), it evaluates the request and responds with an interface (WSDL), containing information about the service provider location and operations. The consumer can now interact with service provider through SOAP messages, the architecture is illustrated on Figure 2.2 [2].

Hypertext Transfer Protocol (HTTP) [16] and eXtensible Markup Language (XML) [50] and are essential in Web Services execution. XML is a markup language designed to provide a simple message format to categorize data. Due to its simplicity, XML is the standard message format for Web Services communication. HTTP is an application-level protocol that allows browsing the Web.

1. Web Service Description Language (WSDL), a XML document describing the Web Service interface, the service methods and data types.

2. Universal Description Discovery and Integration (UDDI), is a platform-independent Web Service registry. The consumers use UDDI to search services and a service interface, WSDL, is provided.

3. Simple Object Access Protocol (SOAP), is a standard communication protocol based on XML over HTTP.

Interoperability and Internet deployment has highly increased Web Services popularity. Currently several standards have been defined: security (WS-Security) [43], messaging (WS-ReliableMessaging) [42], transaction (WS-Transaction) [44], management (WS-Management) [49], and composition (WS-BPEL and WS-CDL) [46] [47].

Web Services in Java

There are several Java technology APIs, the following two are amongst the most used: Java API for XML Web Services (JAX-WS) [3] and Apache Axis2 [4].

Java API for XML Web Services (JAX-WS) is a Java technology to develop Web Services. Currently in version 2.0, it provides a simple and standard model to develop Web Service applications and clients. The process of developing a Web Service using JAX-WS involve the following steps:

1. Annotate in the class file the @WebService tag to expose the application as a Web Service.

[2] Figure taken from from Business-to-business interactions: issues and enabling technologies, The VLDB Journal (2003)

[3] https://jax-ws.dev.java.net/

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

2. Annotate the methods that will be exposed as Web Service operations with @WebMethod tag.

3. Generate server stubs with `wsgen` tool and deploy the Web Service.

4. Generate the client stubs with `wsimport` tool from an executing instance of the Web Service.

Apache Axis2 is an open source framework for developing Web Services in Java. The process of developing a Web Service involves the following steps:

1. Generate the WSDL interface from the Java interface with `java2wsdl` tool.

2. Generate the server stub with `wsdl2java` tool and edit the code.

3. Deploy the Web Service in Axis2 server.

4. Generate the client stub with `wsdl2java` tool and create a client for the service.

2.1.3 Service Composition

Web Service composition is the process of developing a Web Service that gathers resources from other services. For example, a flight reservation service can use other services, a car rental or a hotel accommodation service. The idea of application integration is not new but previous attempts were too complex, difficult to manage and had limited success [1].

The use of composition in business processes is very important, Business-to-business (B2B) interactions at a process level with Web Services requires the definition of a workflow, that describes interactions between services. To achieve a scenario where online applications cooperate with people and organizations efficiently, some characteristics need to be assured:

- All involved parties must be able to communicate through different platforms/frameworks.

- Provide security on personal information and business operations.

- Modularity and interoperability

- Dynamic recomposition of services to face the current/future changes.

To define business activities and workflow there are two approaches, orchestration and choreography, the former based on a central entity that controls all the execution of services and message passing, the latter based on a scheme where all components involved have their role defined and there is no need of a central manager [28].

A business process is a set of of service invocations and activities that produce a business result, within or across organizations.

Orchestration [5]

Orchestration in Web Services is like the maestro conducting an orchestra, there is a central coordinator that defines the execution of the involved Web Services. Web Services in that process have no knowledge of other services, only the coordinator has that information.

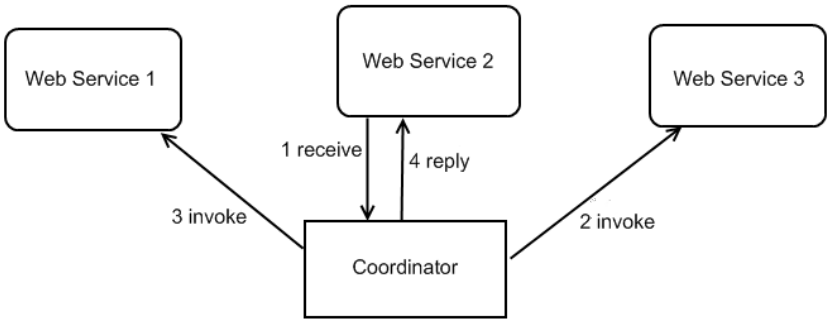

Figure 2.3: Orchestration

Business Process Execution Languages for Web Services (BPEL [6]) [20], proposed on a joint effort from IBM and Microsoft. BPEL is a language for specifying business processes through service composition. BPEL uses partner link types to describe interaction between a BPEL process and involved parties (Web Services and client). This framework main goal is to become the standard in Web Services automation.

Inside organizations BPEL is used to achieve the integration of systems that previously worked isolated. Between organizations BPEL allows a better and easier service integration with other partners. To better understand the orchestration concept a workflow is illustrated in Figure 2.3. The central coordinator is the process (can be another Web Service) responsible for controlling the orchestration, it defines which services and operations are executed. Web Services involved in the composition are not aware of others and do not know that belong in a composition, only the coordinator has that

[5] This section and the next are based on [20]
[6] Also known as BPEL4WS or WSBPEL

information.

Below are some of the most important features of BPEL:

- Describe the logic of business processes through composition of services.
- Handle synchronous and asynchronous operations invocations on services, and manage callbacks that occur at later times.
- Invoke service operations in sequence or parallel.
- Maintain multiple long-running transactional activities, which are also interruptible.
- Structure business processes into several scopes.

Choreography

Choreography in Web Services, unlike orchestration, does not have a coordinator. Each involved web service knows when and with whom execute their operations. It is a collaborative effort that emphasizes the public message exchange between public business processes.

Web Service Choreography Description Language (WS-CDL) [47] is a language that describes the interaction between the participants in collaborative applications. With this approach a global vision of the business process is achieved. In this language it is possible to describe sets of rules that define how services work together. WS-CDL specifications are useful to verify message exchange execution in runtime between business partners.

The following sentence [47] explains in a formal way the Web Services choreography specification:

> "The Web Services Choreography specification is aimed at the composition of interoperable collaborations between any type of party regardless of the supporting platform or programming model used by the implementation of the hosting environment."

WS-CDL authors see it as a complementary language to BPEL, as it concentrates on the flow and behavior of a specific business process (internal behavior), WS-CDL defines the message flow of involved parties. The choreography concept is illustrated in Figure 2.4. In this situation there is no coordinator, every service involved is aware of the composition and knows when to interact and what operations need to be executed.

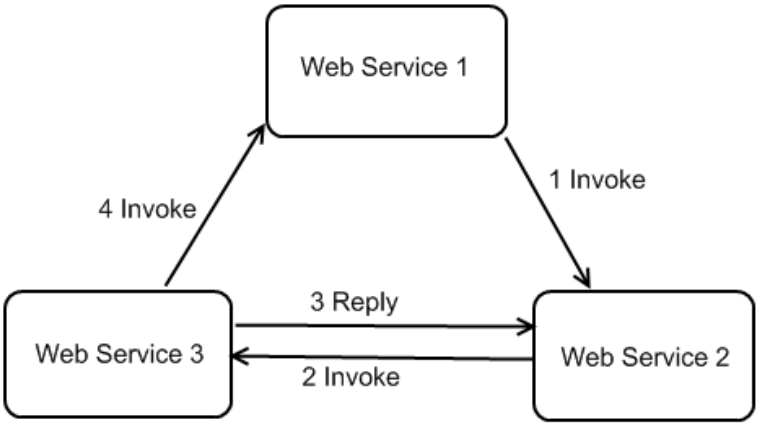

Figure 2.4: Choreography

BPEL, as seen, has support for orchestration (executable business process) but also supports choreography through abstract business processes. Executable business processes combine existing services, define their activities and input/output messages. This process that is created diminishes the gap between developed code and business process specification. Abstract business processes only describe the public message exchange between involved parties.

Although BPEL relies on a central manager to coordinate the composition is more flexible than choreography: the entity responsible for the process execution is always known, Web Services can be added even if they are not aware that belong to a business process and alternative scenarios can be defined as support for any failure. evolution and changes.

Web Service Composition Frameworks

Nowadays Web Services composition and interoperability are very important, not only at research level but also in enterprise collaboration. Several frameworks have been developed trying to offer the best approach in associating services from different locations. Here are some of the developed frameworks that support Web Services composition:

BPELJ [6], is the combination of BPEL with Java to create composition of business process applications. BPELJ allows developers to define portions of Java code, Java snippets, in BPEL process definitions. By integrating Java code in BPEL processes it is possible to execute intermediate actions such as, calculate values for flow control, build XML documents, execute other code without creating another Web Service.

BPELJ defines other type of partner links, Java partner links, making it possible to define Java interfaces instead of only WSDL. This new type of partner links allows a BPEL process to use Java components with Web Services, having the ability of passing serializable Java objects as operation parameter or return value.

JOpera [32], is a visual composition language for service composition that provides tools for modeling, execution, monitoring and debugging of Web Services in workflows. Services are composed in processes, they are defined in a visual notation based in data and control flow graphs. A control flow graph defines the order of task execution. A data flow graph defines from where the information comes to each service.

JOpera has a threading model that decouples process instances from the threads that execute them. This decision in the JOpera engine enables the architecture to move from a centralized to a distributed configuration. To support this process execution is divided in two components: *navigator* and *dispatcher*. The navigator runs processes, it uses a graph traversal algorithm to set the tasks for execution on the **task queue**. The dispatcher retrieves them from the task queue and executes the Web Service invocation, after the invocation is complete the result is put into the **event queue**. The navigator collects the results and updates the process.

The description of service composition is kept independent from the protocols used to invoke services. This abstraction improves JOpera's scope beyond composition of coarse-grained Web Services, allowing inclusion of Java modules in the composition without needing to define them as new Web Services [30].

The composition model is incrementally verified for consistency, for instance, data flow connections of linking services. Also to ensure efficiency on execution, the language is compiled to Java and once more compiled to bytecode, the result is loaded into the kernel of JOpera.

JOpera is a tool for Eclipse IDE, started at Information and Communication Systems Research Group at the Department of Computer Science of ETH Zurich and now actively developed by the Faculty of Informatics of the University of Lugano, Switzerland.

SELF-SERV [4], is a language for Web Service composition based on statecharts. In this framework there are three types of services: *elementary*, *composite and communities*. Elementary service is an application accessible through the Web. A composite service is a composition of Web Services. A service community provides means to compose a large number of Web Services in a flexible way, they offer description of services without mentioning any provider. A community works like a container of substitute

services and, during runtime, is responsible selecting the service that best fits a user profile in that situation.

The orchestration model for SELF-SERV is peer-to-peer and the responsibility is distributed by coordinators attached to each composition state. Coordinators are software components responsible for initiating, controlling and monitoring the attributed state, and collaborate with other coordinators. In runtime the knowledge required by the coordinators is extracted from the statechart, containing preconditions and post-processing actions. The composition and execution environment is implemented in Java and service communication is done with XML documents.

ServiceCom [26], is a Java based tool for modular Web Service composition. This tool is based on *service components*, they are a packaging mechanism for developing distributed applications using already published Web Services. This mechanism is supported by Service Composition Specification Language (SCSL). This language uses constructs such as *activity*, *binding*, *condition* and **composition type** to define the Web Service composition. For instance, the **binding** construct is used to link the **activity** construct with service providers, and can define properties to be used for searching service providers at runtime.

ServiceCom uses a four step approach to service composition: description, planning, building, invocation. The description phase allows the user to specify a composition by defining SCSL constructs in a visual development environment. In the planning phase ServiceCom provides a library with Web Service providers to help the user define a service provider for an activity in the composition. The library contains service providers that are locally known, and services can be added and removed of the library. If the user does not specify a service provider, then at runtime a service provider is selected automatically using the properties defined in the binding construct. The building phase is responsible for linking the composition specification and the actual invocation of services. The following files are generated:

1. For each activity in the composition Java source code is generated.

2. Java source code for the composition.

3. A WSDL definition for the composition

Finally the Invocation phase is the execution of the composition.

2.1.4 Challenges

The most important challenge in service composition is to achieve the automatic composition of business processes, all in a reduced cost, efficiently, reliable, secure and dynamic way. There are no business processes able to react to the evolution and changes. All service interactions are known from the beginning thus being hard to define service composition that function in a proper way in *all* situations, especially dynamic environments.

2.2 Using Web Services in Dynamic Environments

Nowadays Web Services are being used in dynamic contexts, new services are created, others stop working, partners and business rules change. All this motivates for a composition progressing from static to a dynamic and more flexible way. This section introduces the limitations of BPEL, specially in dynamic environments, and several extensions that were developed to overcome these limitations.

2.2.1 BPEL in Dynamic Environments

In BPEL, partner links describe links to Web Services or to clients that invoke BPEL processes, but these links are defined at design time, precluding BPEL use in dynamic environments due to its static logic of composition [20]. Next some limitations of BPEL are presented:

Data manipulation: In BPEL it is not possible to create variables dynamically, each used variable must be declared at design time and at process level. BPEL cannot recognize if two variable are composed of the same primitive type, this reduces BPEL flexibility [13].

Fixed participants: BPEL assumes that the number of participants in the process is known at design time and cannot be modified during runtime. This limits BPEL use on dynamic environments where participants enter and leave at any time [15].

Dynamic binding: In BPEL the service requester must know the correct endpoint address of a partner service, this results in tightly-coupled systems between service providers and requesters. In a situation where the partner service address changes the process fails [12].

Failure recovery: BPEL presents two statements for handling faults: `catch` and `catchAll`. The `catch` handles faults defined in WSDL. The `catchAll` handles

undefined faults. When a Web Service fails and a process tries to invoke that service the fault is catched with `catchAll` statement, but since the fault type cannot be determined this limits BPEL QoS [13]

It is not possible for the developers/designers to have an absolute domain knowledge and account all recovery strategies for BPEL processes. BPEL statements for recovery are simple and do not allow to define complex recovery plans, for instance, rollbacks, alternative execution models or Web Service replacement [24].

Dynamic environment: BPEL is not designed for dynamic environments, businesses evolve in time and new services need to be integrated quickly. Addition of new services should not require significant work in changing the process workflow, a substitute service can have a different interface so a mapping is needed [40].

To address these limitations of BPEL on Web Services composition, dynamic binding[7] technique is used.

Actions that are performed/used several times are associated with a name or symbol, this is known as binding. There are two types of binding, static and dynamic. Static binding is when the binding of a call to a method is performed during compilation time and dynamic binding is the process of binding one call to a method in runtime [11]. The latter is used in situations where there is more than one method for a call, preventing the developer from writing conditional statements. Although this technique slows the application there are some advantages:

- **Polymorphism**, more than one method can be defined for one operation, the correct method is selected at runtime.
- **Extensible**, new classes can be created to receive messages without modifying or recompiling the class where the message came from.
- **Complexity**, reduces complexity by eliminating switch instructions and substituting them with functions calls.

2.2.2 Dynamic Binding support in BPEL

Although BPEL has limitations regarding dynamic binding there are some approaches developed to overcome its limitations in dynamic binding, namely WS-Addressing [45], a standard providing a mechanism, *endpoint reference*, that represents a Web Service endpoint and enables the user to select services from available services or specify new

[7] Sometimes called Late binding

services at runtime [1]. However this approach does not solve all problems, intervening services are still defined at design time. To overcome some BPEL limitations some extensions have been developed and are now presented.

Using proxies

Proxy is a technique used by several frameworks to support dynamic binding in Web Services, it has the responsibility of monitoring service behavior so it can detect any failure and reconfigure in an autonomous way. The proxy concept is used on MASC [12], WS-Binder [10] and TRAP/BPEL [14], all extensions to BPEL.

Manageable and Adaptable Service Compositions (MASC) is a framework for Web Service composition in dynamic environments, providing policy-based mechanisms to achieve a more tolerant composition. To achieve this goal, business process interactions are made to a dynamic proxy, that is responsible for monitoring failures and service replacement.

The policy mechanism in MASC is called WS-Policy4MASC and is an extension to WS-Policy. WS-Policy4MASC is used to define policies for: finding the most suitable endpoint for a given interaction, and to control binding at runtime. The policies used for service selection specify service preferences such as, performance, reliability and cost. These policies are attached to process activities allowing flexible service selection based on the instance executing the process. At runtime if a service replacement is executed there are dependencies that need to ensured between the process activity and service. MASC defines a parameter for this type of situations, *ServiceBindingScope*. By assigning the parameter to `SingleInvocation` then a different service provider can be used for each process operation, this is only possible if operations are different at every execution of the process, for example, in a web search.

MASC introduces the concept of Virtual Endpoint (VEP), where several equivalent services act like a recovery block, configured to provide backup when necessary. VEP supports MASC middleware in dynamic binding, for each service added to a VEP block it is necessary to define a protocol for message exchange between the proxy and service. The proxy is responsible for heterogeneity using conversion rules that describe how input/output messages from equivalent services need to be transformed.

MASC middleware does not use UDDI registry so the authors developed their own data model for managing and searching metadata on registered services.

WS-Binder is a Web Service composition framework with support for dynamic

binding, the framework operates at two levels of composition: before and during its execution.

In pre-execution binding, the framework is responsible for setting the bindings that best satisfy the QoS of the composition. WS-Binder accepts a BPEL workflow as input and a set of properties, the workflow goes through a series of steps, service discovery, selection, binding and proxy instantiation, the last is responsible for initiating the service and all interactions go through it. In a failure situation or when bindings are not defined a slice of the workflow is selected for rebinding, the discovery and selection process is executed again. After rebinding is complete the control is returned to the proxy.

At runtime after a service is replaced, the invocations performed by the proxy may need conversion because the new service may represent the same operation differently, so the proxy call an adapter that applies a protocol to resolve mapping differences. To achieve this, services need a richer interface than WSDL. The framework introduces the concept of *facets* that describe specific properties of the service, for instance, a behavioral facet describes the behavior of a service in terms of a state machine.The mapping is done by components known as *adapters*, they apply the protocol looking for concepts that match.

WS-Binder is being developed within the SeCSE (Service-Centric System Engineering) European IP Project, and includes components for service specification, discovery, composition, publication, testing and monitoring.

TRAP/BPEL is a framework for Web Service composition supporting dynamic adaptation, providing BPEL with autonomic behavior by using a generic proxy. BPEL processes are adapted to gain self-management behavior, *hooks* are incorporated in the process, they catch and forward the interaction to the generic proxy. Hooks are normally included in `invoke` operations.

Specific proxies have an interface that is an aggregation of all service interfaces that are being monitored. This can be troublesome if a high number of services are being used. TRAP/BPEL introduces a generic proxy that has a standard interface for all monitored services invoked by adapted BPEL processes. So the process of replacing a service invocation to a proxy invocation involves identifying all the messages necessary to create the input message for the proxy, contents are serialized in the input variable, and it is also necessary to create a sequence of activities that deserialize the output message from the proxy.

The proxy upon receiving an invocation checks previous loaded policies to find a match for that invocation. If a policy is found then the proxy can invoke the service

recommended in the policy, find a new service, retry service invocation, if no policy is found then default behavior is to search the registry for a service that has the same port type.

Using Aspects

Aspect-Oriented Programming [7] is a paradigm that increases modularity of crosscutting concerns [8]. Concerns like auditing and logging are very important in business processes, they are spread in several process definitions leading to code duplication. This way applications are more complex and difficult to manage.

To address this AOP introduced a new modularity unit, the *aspect*, composed by three features: *join points*, *pointcuts* and *advices*. A join point is a point defined in the program execution, including method call, field access, exception handler. A pointcut is a element that extracts the context of join points, it specifies where the aspect is integrated. An advice is auxiliary code that is executed when the specified join point is reached, this code can be set for execution before, after or instead of the original join point.

Aspect-Oriented for BPEL (AO4BPEL) is an aspect-oriented extension for BPEL to make Web Service composition more flexible and modular. AO4BPEL enables dynamic integration of aspects in processes at runtime, and they can be activated and deactivated during execution allowing the application to adapt its behavior dynamically. Dynamic binding in AO4BPEL is achieved by introducing, at runtime, aspects with new functionalities to a composition. For example, a scenario where there is a composition of a flight travel with hotel accommodation service and the developer defines an aspect with car rental service. The developer submits to BPEL engine the WSDL and port address of the new Web Service and the aspect is registered. This new aspect can be activated dynamically while the process is running. If business rules change in the future the aspect can be deactivated.

The focus of AO4BPEL is mainly Web Services composition and not management. Web Service Management Layer (WSML) [40] is a management layer that handles management problems. This framework is an alternative to BPEL.

The management layer is placed between the application and Web Services and allows dynamic selection and integration of services, service management on the client side and supports rules that restrain selection, integration and composition. WSML functionality is implemented by JAsCo, a dynamic aspect-oriented language with the following properties:

[8] Crosscutting concerns are concerns that cross several entities in a business process.

- Aspects are highly reusable, they are described independently from context.
- JAsCo allows easy application and removal of aspects during runtime.
- JAsCo supports specification for aspect combinations.

The JAsCo language has two concepts that are important: *aspect beans* and *connectors*. An aspect bean is an extension of Java Bean component that specifies crosscutting behavior. Connectors are responsible for specifying where the behavior is deployed.

WSML uses an Abstract Service Interface (ASI) between concrete services and application to hide syntactic differences between semantically equivalent services. The application makes requests to ASI, and WSML is responsible for making the transformation to a concrete service. To achieve this it is necessary to describe the mapping of service or service composition into ASI, these descriptions can be provided by the service owner or specified by the developer. Also it is necessary to describe the mapping between the ASI and concrete service interfaces. Both mappings are done with sequence diagrams.

To realize this abstraction, JAsCo defines aspects that are responsible for redirection of generic requests to services. This aspect catches requests from the application and replaces them with a concrete Web Service invocation. Connectors are responsible for deploying the redirection aspect, and they contain the mapping information of generic requests from the application and also how to make that request to a specific service. So there will be as many connectors as different requests from the application.

Connectors are created dynamically providing integration of new services dynamically. This feature allows new services to be integrated by defining a new connector at runtime that maps the new service. In the WSML architecture the **Selection Module** is responsible for selecting the correct connector. This module works together with **Monitoring Module** that searches services in the UDDI registry. For example, when a service is not available or takes too much time to respond, the Selection Module can replace the service by deactivating its connector and activating another connector.

Responding to Failure Recovery

Nowadays it is very important to increase self management behavior in Web Service compositions, services today work in dynamic environments and their becomes essential that services can configure autonomously.

SH-BPEL [24] is a plugin for BPEL engine that promotes self healing. The architecture is composed by standard BPEL engine and a **Process Manager**, the main module

is responsible for executing management actions and is itself composed by other modules, **Message Monitor** and **Management Engine**. For example, when a service fails the Message Monitor sends a message to the Management Engine, this module works in two modes: *active* or *passive*. In passive mode the module ignores the message and waits for an external action. In the active mode the management action attached to the message is executed. Here are some of the supported modules for SH-BPEL:

- **Web Service Invoker:** dynamic invocation of Web Services, the invocation is done independently from WSDL.
- **Substitution Manager:** functionalities for Web Service replacement.
- **Web Service Retriever:** operations that enable dynamic Web Service search.
- **Mediation Service:** when a Web Service is replaced the WSDL interface may be different and so the communication with BPEL is not possible. To minimize this problem the Mediation Service transforms the exchanged messages between services and BPEL.

Using Reflection

JOpera [31] uses a visual composition language to define service composition, its framework provides a development environment with the following features: conditional execution, failure handling, type-safety, nesting, recursion, late binding with reflection.

Reflection is the ability of a system to represent and modify by itself internal information. JOpera uses reflection to access metadata information about the static process structure, its execution state and runtime environment. These characteristics are available through *system parameters* and *system services*. With reflection the developer has access to bindings and registry services during runtime and is able to control them through the composition language making the process more flexible.

Each task defined in JOpera's visual editor for composition has associated a set of system parameters and properties, these contain metadata about process execution and are updated by the runtime environment. System services expose information of JOpera runtime environment and allow interaction with the process. This information includes: program library API, process control API and resource management API. The process control API is used for controlling process execution from the process itself, allowing for instance, to cancel a process execution after reaching a condition, to suspend automatically a process when coming to a stage.

In a scenario where Web Services from different organizations are used in a composition some problems may appear, organizations may represent data differently. So to overcome this limitation it is necessary to adapt messages from different services. JOpera uses Stylesheet Transformations (XSLT) or XML Path query language.

Using groups

To overcome BPEL limitations on the fixed number of participants Sliver [15] middleware has been developed. This framework is an extension to BPEL with the goal for deployment in mobile devices such as, mobile phones, pda's, laptops. Sliver only depends on two lightweight external libraries increasing its use by a wide range of devices, and supports several communication protocols, from HTTP to Bluetooth. To find services Sliver uses Bluetooth service discovery mechanism. As new service providers are discovered then BPEL can update its service bindings.

In regard the problem of fixed participants, the Sliver middleware implements extensions for *partner groups* and *partner link* reuse. A partner group is a unbounded list of partner links, they can be assigned to any number of service endpoints at the same time while partner links do not, and can be controlled by the process at runtime. With this it is not necessary to define the number of participants at design time, initially partner groups have no connection to endpoints, then with `add` and `remove` BPEL activities it is possible to add an endpoint, from a partner link, to a partner group.

In Sliver middleware the **BPEL Server Layer** is responsible for mapping incoming partner links to the kinds of messages that they accept as input and map outgoing partner links to concrete service endpoints. The applications that use Sliver can define policies to use already defined mappings or runtime mapping of partner links.

In the studied frameworks dynamic binding and failure recovery definitions may appear similar, dynamic binding refers to the ability of runtime service discovery, and failure recovery is related to a Web Service replacement, substitute services may already be defined before runtime. To summarize, all the frameworks presented in this section are shown in Table 2.1 in regard to BPEL limitations presented on Subsection 2.2.1. Not all limitations presented are in the table, only the ones encountered on the studied frameworks.

	Fixed participants	Failure recovery	Dynamic binding	Dynamic environments	BPEL based
MASC	No	Yes	Yes	Yes	Yes
WS-Binder	No	Yes	Yes	Yes	Yes
TRAP/BPEL	No	Yes	Yes	No	Yes
AO4BPEL	No	Yes	No	No	Yes
WSML	No	Yes	Yes	Yes	No
SH-BPEL	No	Yes	Yes	Yes	Yes
JOpera	No	Yes	Yes	Yes	No
Sliver	Yes	Yes	Yes	Yes	Yes

Table 2.1: Frameworks addressing BPEL limitations

2.2.3 Dynamic Composition of Web Services

Business services are frequently changing new ideas are always coming, research in this topic is still emergent there is a need to make Web Services composition suitable in dynamic environments. Dynamic composition of Web Services can detect service failures and present alternatives to the user, allow the user to search services that satisfy functional and non-functional requirements at runtime, ensure more flexibility. The ultimate goal is a state where everything is autonomous, systems developed with the ability of self-management, self-configuration, self-healing and self-protection.

To achieve this state of the art new approaches have been developed but they are mainly extensions to BPEL, so the problem needs to be addressed from another perspective.

2.3 Web Services with attributes

The UDDI registry shows wide support from some of the software companies that use the notion os service property on Web Service discovery and use, it has become the standard for storing interface descriptions of Web Services. In UDDI services are described in XML with physical attributes such as name, address and provided services. However, XML is a machine-readable language but not a machine-understandable language, so UDDI does not support semantic information on Web Services. For example, two different syntactic queries to UDDI may represent semantically equivalent services. Also UDDI does not support discovery by service capabilities and other properties, thus inhibits to explore its abilities [34].

The use of semantic information together with UDDI registry to provide a better Web Service description is a work in progress. To overcome this limitation and enhance UDDI use, some work has been developed: *tModels* [38] and Web Ontology Language for Web Services (OWL-S) [27].

2.3.1 TModels

A Web Service to be registered in UDDI must implement an interface, and it has to be previously registered in UDDI. For example, airlines may work together to define an interface and publish it in UDDI for querying ticket prices on a specific date, time and departure/arrival cities. This situation presents several Web Service implementations, one for each airline, but only one interface in UDDI. This interface is a tModel, an important data structure in UDDI registry that can be used to represent interfaces, classification information and can be used as namespaces to give more meaning in UDDI. The tModel is composed by:

- **name:** a Web Service name.
- **description:** a description of the service.
- **overviewURL:** a link pointing to the document that describes the Web Service, for example, the WSDL interface.
- **categoryBag:** used to categorize information, helps to determine which category the business belongs to. The categoryBag element can hold several **keyedReferences** elements.
- **Universally Unique Identifier (UUID):** this is generated automatically and used to reference the tModel.

To register the developed Web Service in UDDI it is necessary to create a data model, this document contains the following elements: *businessEntity*, *businessService* and *bindingTemplate*. The businessService describes a service and has a bindingTemplate element that points to the tModel. The *businessEntity* provides a Web Service that implements the interface (tModel) referenced by bindingTemplate.

2.3.2 Semantic Web Services

Web Services and Semantic Web are two areas of the World Wide Web that had great evolution in past years, the intersection of these two promotes semantic in Web Services [22]. Semantic Web focuses on having software agents that can use resources of

the web intelligently, making the web understand and satisfy requests from people or other entities. The goal of using together these two technologies is to achieve a better service description, leading to a greater automation in service composition, selection and invocation.

3

SeDeUse

Nowadays one of the top reasons in distributed systems research is the interaction of networks composed of mobile and pervasive devices with the Internet, this is a priority in service-based applications available in the business market.

Services mainly abstract business processes but they can also abstract common publicly available resources, such as a network printers, sensors, displays, and any type of resource that can be made available through the network. However the state-of-the-art in this field field targets mainly business-to-business interaction as service directories store business specific interfaces instead of general abstract interfaces. Besides the existing coordination models, such as service orchestration and choreography, were designed to operate generally with previously known locations and tightly coupled resource awareness and usage.

Several solutions have been proposed, discussed in Chapter 2, to overcome these limitations but, a new approach is required for this problem. SeDeUse presents itself as a new approach, thinking the problem from scratch, were resource awareness can be abstracted from its use. SeDeUse [29] is an initial proposal for a model that tries to overcome the mentioned limitations, the model is described thoroughly in this chapter.

3.1 SeDeUse Model

SeDeUse features a two layer approach to separate the service use from awareness and a intermediate layer between the application and services. The latter provides dynamic discovery, management and acquisition of services. The two layers are:

- **Service Awareness Layer (SAL)**, defines the *kind* of services to be used on the application, i.e., services that are discovered in the network. Also service attributes are defined to constrain the search space.

- **Service Use Layer (SUL)**, defines a simple coordination model for service use.

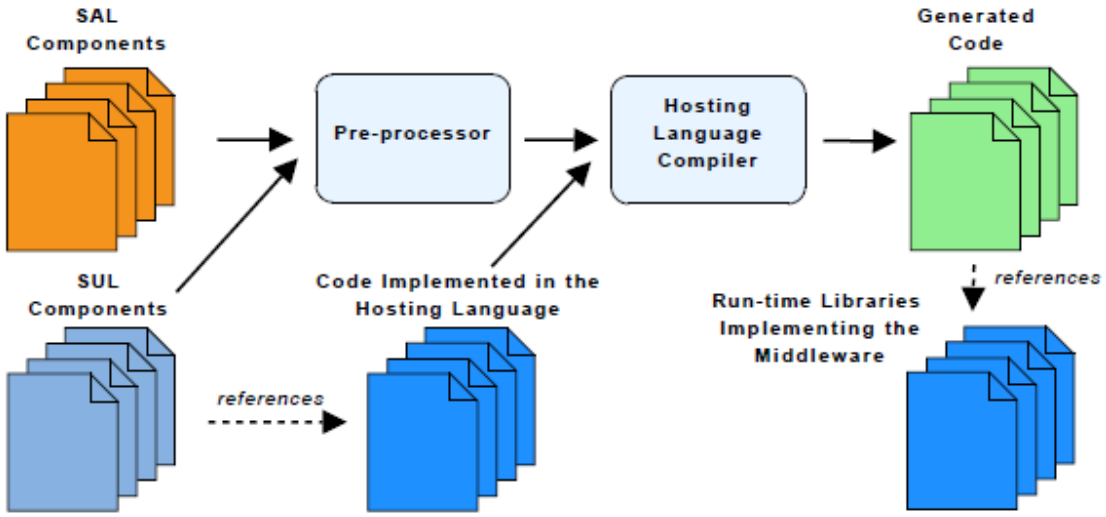

Figure 3.1: The compilation process

This separation of service use and awareness was designed with the service end-user in mind and the model offers intuitive abstractions for its use, with this model the user can define a simple coordination language for the functional components.

The SeDeUse model does not define by itself a complete language so this model will be mapped into a hosting language and a preprocessing stage is required to generate code of the hosting language, as illustrated in Figure 3.1. This intermediate step is responsible for processing SAL and SUL definitions and generate code that will interact with the middleware for service discovery, binding, reconfiguration and failure recovery. The middleware also supports migration of computations, devices with low computational resources, mobile phones, pda's, can send their computation to more capable devices.

3.1.1 Service Layers

The syntax for service awareness and service use components is based on identifiers and values defined in Table 3.1. A sequence of zero or more elements of a given category γ by $\tilde{\gamma}$, an empty sequence by ϵ and an optional symbol or production by the [ ] notation.

s, r	Service identifier
o	Service operation identifier
a	Variable identifier
t	Type identifiers of the hosting language
x	Exception identifiers of the hosting language
v	Values of the hosting language

Table 3.1: Values and identifiers

Service Awareness Layer

This layer defines a syntax for service declaration and definition of attributes that is shown in table 3.2, the syntax is composed by a sequence of service kind declarations. A kind is service interface identifier and a set of specific properties. The identifier defines the key for discovering the service in available repositories, and attributes will restrict the search space. It is also possible to declare restrictions as **hard** (= operator), soft (**in** operator) or as preferences using **pref** keyword.

D	::=	$D\ D$	Sequence of declarations
		$\mid s\ \{\ \tilde{A}\ \}$	Service kind declaration
		$\mid s\ \{\ \tilde{A}\ \}$ **alias** r	Service kind declaration with alias
A	::=	[**pref**] $a\ =\ v$	Attribute constraint
		$\mid$ [**pref**] a **in** $\{\ v_1, v_2, \ldots, v_n\ \}$	Attribute soft constraint

Table 3.2: Syntax of declarative components

Listing 3.1 defines a possible SAL representation for a printer service. The **colors**, **paper** and **type** attributes limit the search space imposing restrictions on service attributes.

Listing 3.1: A printer

```
Printer {
  colors = "blackandwhite",
  paper = "letter"
  type = "laser"
}
```

Listing 3.2 defines a possible document converter service, where the **input** attribute can be "doc" or "docx" and the **output** is a document in "pdf" format.

Listing 3.2: Document converter

```
Convert {
  input in {"doc" , "docx"},
  output = "pdf"
}
```

Several definitions of a service kind may exist so to avoid name conflicts the syntax resorts to *aliases*. These names are service identifiers and can be used in SAL and SUL definitions. Listing 3.3 presents the use of aliases to distinguish between different *kinds* of services, the *ColorPrinter* defines a *Printer* kind with designated attributes. *ColorLaserPrinter* constrains the *ColorPrinter* definition by setting the **type** attribute to be laser or ink jet.

Listing 3.3: More printer type definitions

```
Printer {
  colors = "color",
  paper = "a4"
} alias ColorPrinter

ColorPrinter {
    type in {"laser", "ink jet"}
} alias ColorLaserPrinter
```

These service declarations and attributes do not define concrete services, they define service kinds. The use of SAL together with SUL definitions will produce different results, according to each SUL definition.

Service Usage Layer

SUL provides programming abstractions and a coordination model in service use to manage computations in network services. The syntax for SUL is described in Table 3.3

Using Services: The **use** construct abstracts the computation into a set of parameters, similar to a class definition in Object-Oriented languages. To create instances of such abstraction the **new** construct is used, it is responsible for binding the defined parameters to values provided as arguments. Service operations are invoked as methods on objects.

The innovation in this constructs is that the **use** keyword allows code to be represented as service identifiers and these identifiers are bound in a dynamic and transparent way every time an instance is created. After the identifiers are connected to instances it is possible to invoke operations through them.

P	::=	**uses** $\tilde{S}$ **in** c**(**$\tilde{a}$**)** P $\tilde{X}$	Service use abstraction
		$\mid P \mid P$	Parallel composition
		$\mid P$ **;** P	Sequential composition
		$\mid$ **{** P **}**	Grouping
		$\mid [a$ **=**$]$ E	Assignment
		$\mid$ **retry in** e	Restart a transaction
		$\mid \tilde{i}$	Hosting language process
E	::=	**new** c**(**$\tilde{e}$**)**	An instance of an use abstraction
		$\mid s.o$**(**$\tilde{E}$**)** $\mid$ s**[**e**].**o**(**$\tilde{E}$**)**	Method invocation
		$\mid e$	Hosting language expression
S	::=	[**volatile**] e s $\mid$ [**volatile**] **all** s	Service allocation
X	::=	**catch (**x a**) {** P **}**	Exception handling

Table 3.3: Syntax of functional components

Listing 3.4: Using a service

```
use Printer in MyPrinter(String doc) {
    Printer.print(doc)
}
new MyPrinter(``myDocument'')
```

Listing 3.4 presents an example of **use** and **new** construct, the parameter "doc" is passed in constructor, while the binding for Printer service is obtained transparently with the intermediate layer when the instance is created. Another property of **use** is its ability to bind several different services of the same kind. This feature is useful to manage several service requests or to synchronize data, an example is illustrated in listing 3.5. The services are accessed just like an array, and to avoid exceptions like array index out-of-bounds the indexes are converted to values between 0 and the number of services available. This feature is achieved by defining an upper bound number of services, thus conforming to dynamic environments, and permits multiple service instances to be used transparently.

Listing 3.5: Using several instances of a service

```
use 2 SearchEngine in MySearch(String query) {
    SearchEngine[0].search(query) |
    SearchEngine[1].search(query)
}
```

The user may omit the the index and invoke the operation, this is possible because *SearchEngine.search(query)* stands for *SearchEngine[i++].search(query)*, where *i* is an integer initialized with 0.

Defining Computations: Computations are performed by processes(sequences of instructions) of the hosting language, and they can be in a parallel or sequential way. Instructions can be defined as parallel (| operator) or sequential (**;** operator). If parallel composition is used then it is necessary to create a thread for each defined instruction. The **;** operator defines a point where all threads created from the current flow are terminated. The use of parallel and sequential composition is illustrated in Figure 3.2.

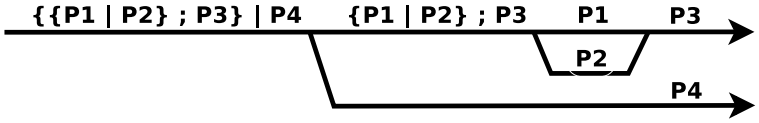

Figure 3.2: Execution flow for $\{\{\ P1 \mid P2\ \}\ ;\ P3\ \} \mid P4$

Failure Recovery: This layer allows to define failure recovery with two constructs: **retry** and **volatile**. The model uses an exception handling mechanism syntactically similar to Java. Protection is made for the code inside a **use** against a binding failure, if discovery or invocation of the service fails then an exception is raised. The difference from the usual exception mechanisms is that here the scope of protection is for service use, and not the delimited code. Listing 3.6 illustrates an example of this mechanism, the code delimited by **use** is seen as a transaction. If an exception is raised then the transaction can be re-executed using **retry**. It performs a new discovery process, eliminating the previous binding, and re-executing the service.

Listing 3.6: Handling exceptions

```
use Printer in VirtualPrinter(VirtualPC vpc) {
   vpc.setPrinter(Printer);
}
catch (ServiceException e) {
  vpc.unsetPrinter();
  retry in 0;
}
```

SUL definitions are regarded as stateful, (i.e., services have a representation of state), but when working in dynamic environments services may become unavailable and bindings are lost. Thus, the **volatile** keyword can be used for independent service invocations (stateless). Rigorously, the use of this keyword totally agrees with the dynamic aspect of this kind of environments. And, in this sense, if a service binding is lost a new one will be established by the middleware and no exception is raised.

Listing 3.7: Using the volatile keyword

```
use volatile Service in Abs() {
    Service.op1();
    Service.op2()
}
new Abs()
```

Software Mobility: The model does not explicitly refer to mobility where the program visits hosts, but a service location can be defined in service definitions. A special attribute (@) allows the user to define if a given resource is local or remote to the computation and the device itself. This attribute can take the following values:

- **local:** to guarantee that the resource is local to the device.
- **remote:** the opposite of local.
- **coupled:** the resource is local to the computation.
- **closest:** the resource and the computation are as close as possible.
- **performance:** the system chooses the instance that provides a better performance.

Code migration to a target host will only happen if he is willing to accept the code for execution. This feature is not part of this project, but it is being developed in another project and will be integrated with SeDeUse.

The model resorts to software layer to handle the characteristics of dynamic environments. As illustrated on Figure 3.3 the layer is responsible to answer requests from the application, generated in the compilation process, namely, obtain service bindings, discovery, matching, and replacement.

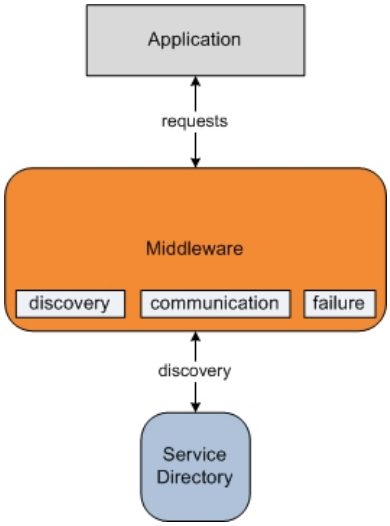

Figure 3.3: Middleware process

4

SedJ

In this chapter we present the concretization of the SeDeUse model that was introduced in Chapter 3. We have chosen Java programming language as our hosting language for the model due to its widespread use and rapid prototyping. Our language with the new abstractions added to Java language is called Sedj.

The SeDeUse architecture is structured according to two main modules, **compiler** and **middleware**. The compiler is responsible for processing SAL and SUL files in order to generate the executable Java code. The middleware is placed between the application and the underlying network of services, being responsible for dynamic service binding, discovery and recovery. These interactions between the application and network are executed transparently, hiding the dynamic behavior from the user.

This chapter starts by presenting the concrete syntax for both SAL and SUL layers, which includes the modifications required to adapt them to the Java language. Then follows a description of each module, which includes the technologies used and a description of the implementation.

4.1 Concrete Syntax

The SeDeUse framework relies on two layers, as mentioned in the previous chapter, one for service awareness (SAL) and other for service use (SUL). While the SUL defines a simple coordination model for service interaction, SAL defines *kinds* of services to be used in the application, where the service *kind* specifies a class of services. SAL components are defined by a set of properties and a given interface.

The model, introduced in Subsection 3.1.1, presents an abstract syntax for both layers. However, some modifications are required to integrate the new abstractions with the Java language specification. The concrete syntax for SAL and SUL layers is explained in Subsections 4.1.1 and 4.1.2, respectively.

4.1.1 Service Awareness Layer Syntax

The concrete syntax for a SAL component is specified in Table 4.1. A SAL definition is composed by a sequence of service kind declarations. Each kind is identified by a name, an interface, and a list of properties.

The reserved words for the SAL syntax are highlighted in bold in the grammar, and cannot be used as identifiers. The **[ ]** identifier stands for optional value.

$Program$	::=	$Service$	Program
	|	$Service\ Program$	Sequence of declarations
$Service$	::=	**service** $ServiceDef$ { $AttList$ }	Service kind declaration
$ServiceDef$	::=	$Identifier\ InheritanceType\ Identifier$	
$InheritanceType$	::=	**implements**	Inheritance Type
	|	**extends**	
$AttList$	::=	**[pref]** Att **[,** $AttList$ **]**	Attribute preference
Att	::=	$Identifier$ **=** $ValueList$	Attribute constraint
	|	$Identifier$ **in** { $ValueList$ }	Attribute soft constraint
$ValueList$	::=	$Value$ **[,** $ValueList$ **]**	
$Value$	::=	(IntegerLiteral | StringLiteral)+	
$Identifier$	::=	StringLiteral	

Table 4.1: Concrete syntax of SAL components

In order to seamlessly incorporate these definitions in Java, we decided not to infer the interface of the service *kind* from its use on the SAL components, but rather to free the programmer to explicitly specify it, as is usual in Java programming. The use of an interface at this stage ensures the absence of invocation errors.

For a better comprehension of the SAL syntax, we now present some examples in Listing 4.1. Listing 4.2 illustrates the correspondent interfaces. **BWPrinter** specifies a printer service with three properties (**color**, **type** and **paper**) and the interface (*Printer*) that the service must match. In this situation the properties correspond to a printer with "black" color, that uses "A4" paper, and of "laser" type.

The **Display** example presents a definition of a document display service with two properties: **screen** and **files**. For these properties the values are "wide" for screen and "documents" for the type of file to be visualized. This service definition implements the *Monitor* interface.

Listing 4.1: Service kind definitions

```
service BWPrinter implements Printer {
    color = "black",
    type = "laser",
    paper = "a4"
}

service Display implements Monitor {
    screen = "wide",
    files = "documents"
}

service DocumentConverter implements
    Converter {
    input in {"doc", "docx"},
    output = "pdf"
}

service DocumentTranslator implements
    Translator {
    format = "doc",
    language in
        {"english", "french", "german"},
    output = "portuguese"
}

service ColorPrinter extends BWPrinter {
    color = "colors"
}
```

Listing 4.2: Interfaces

```
public interface Printer {

    void print(byte[] document);
}

public interface Monitor {

    void show(byte[] document);
}

public interface Converter {

    byte[] convert(byte[] document, String
        extension);
}

public interface Translator {

    byte[] translate(byte[] doc, String
        input_lang, String output_lang);
}
```

In another example in Listing 4.1 (**DocumentConverter**), is specified a possible service that converts documents in one format to another. The property **input** refers to a soft constraint (**in**), and in this case the service accepts documents in "doc" or "docx" format, and the **output** property expresses the resulting document format, "pdf". The interface for this SAL definition is *Converter*, that specifies that the service accepts a document in an array of bytes, a string symbolizing the extension, and returns an array of bytes of the converted document.

The **DocumentTranslator** defines a possible service that translates documents to a given language. The **format** property refers to the document format, in this case only "doc" format documents are accepted. The **language** property expresses the possible languages that the service can accept, and the **output** specifies the language in which the document is translated to. This service definition implements the *Translator* interface.

Finally, the **ColorPrinter** service extends the previous **BWPrinter** definition and overrides the **color** property value to "colors". This feature (**extends**) discards the orig-

inal use of **alias** in order to distinguish equal services of different *kinds*.

4.1.2 Service Use Layer Syntax

In Table 4.2 we can see the concrete syntax grammar for SUL, it is an extension to the original Java programming language specification. Once more the reserved words for the SUL syntax are highlighted in bold in the grammar, and cannot be used as identifiers. The ϵ identifier stands for empty value. This syntax displays several modifications regarding the abstract syntax presented in Chapter 3 on subsection 3.1.1. These changes are performed to seamlessly integrate the Sedj syntax in Java language, and are now described:

use $\Rightarrow$ uses: is defined at class level, just like **implements** or **extends**. It allows to extend the set of free variables in a class body (methods and constructors) with service identifiers. These identifiers are bound, transparently, whenever an instance is created.

new: the syntax remains unaltered , the semantics however now express a service bindings request.

catch: service binding exception resorts to the Java catch mechanism syntax, and is defined for service use.

retry in -> retry Java expr in Java expr: this construct is used for service restart, it is defined the number of attempts and time to wait (i.e. **retry** 2 **in** 3;) before restarting the process.

We now present some examples that demonstrate the SUL concrete syntax.

Document Converter service

In Listing 4.3 we have an example of a document converting service. The **uses** construct abstracts the service and its computation into the service identifier (**DocumentConverter**). The identifier refers to a SAL definition where properties and an interface were declared. Line 9 shows that service invocation is performed upon the service identifier. The exception mechanism (**catch**) in line 13 protects the service use of broken service bindings, and in line 14 the **retry** construct triggers service re-binding for, at most, two attempts and waits 4 seconds before calling the service again.

$Program$	::=	*JavaPackageDecls JavaImportDecls* $TypeDecls$	Program
$TypeDecls$	::=	$TypeDecl$ [$TypeDecls$]	Type declarations
$TypeDecl$	::=	$ClassDecl$	Class declaration
	|	*JavaInterfaceDecl*	Java interface
$ClassDecl$	::=	*JavaModifiers* **class** *JavaIdentifier* **uses**	Service use
		Srv_Aloc *JavaIdentifier JavaInterfaces* $Body$ $Catch$	
Srv_Aloc	::=	**[volatile]** [Srv_Field]	Service allocation
Srv_Field	::=	*JavaIntegerLiteral+*	Java number
	|	**all**	All services
$Body$	::=	{ $BodyMembers$ }	Class body
$BodyMembers$	::=	$BodyMember$ [$BodyMembers$]	Class body members
$BodyMember$	::=	$ClassMember$	Class member
	|	*JavaStaticInitializer*	Java static initializer
	|	$JavaConstructorDecl$	Java constructor
$ClassMember$	::=	*JavaField*	Java field declaration
	|	$Method$	Method declaration
$Method$	::=	*JavaMethodHeader*	Java method
	|	[$MethodBody$]	Method body declaration
$MethodBody$	::=	*JavaLocalDecl*	Java local declaration
	|	*JavaStatement*	Java statement
	|	$SedjExpr$	Sedj expression
$SedjExpr$	::=	*JavaExpr ; JavaExpr*	Sequential composition
	|	$SedjExpr$ | $SedjExpr$	Parallel composition
	|	*JavaIdentifier.JavaMethodCall*	Method invocation
	|	**new** *JavaConstrutorCall*	Instance of uses abstraction
	|	**retry** *JavaExpr* **in** *JavaExpr*	Restart a transaction
	|	*JavaExpr*	Java Expression
$Catch$	::=	**catch** { **retry** *JavaExpr* **in** *JavaExpr* }	Exception handling

Table 4.2: Concrete syntax of SUL components

Listing 4.3: Document converter service example

```
1  public class ServicePDFTest uses DocumentConverter {
2
3      byte[] res = null;
4
5      public ServicePDFTest() {
6      }
7
8      public byte[] call(byte[] doc) {
9          res = DocumentConverter.convert(doc, "pdf");
10         return res;
11     }
12 }
13  catch(Exception e) {
14     retry 2 in 4;
15 }
```

Printer service

Listing 4.4 illustrates a printer service example, it presents parallel composition, an implicit exception handling mechanism, and uses, if available, two instances of the **ColorPrinter** service. Line 11 shows the parallel composition of service invocations using the | construct. The use of **volatile** construct in line 1 guarantees the automatic rebinding whenever a failure occurs and some other instance is available. Service replacement is only performed once, if no match is found in service registry then the application ends execution and a message is shown that no service was found.

Listing 4.4: Printer service example

```
1  public class ServicePrinterTest uses volatile 2 ColorPrinter {
2
3      byte[] document = null;
4
5      public ServicePrinterTest() {
6      }
7
8      public void call(byte[] doc) {
9          this.document = doc;
10
11         { ColorPrinter.print(document); } | { ColorPrinter.print(document); }
12     }
13 }
```

Translation service

Listing 4.5 illustrates a document translation service. This example uses the **all** construct instead of specifying the number of different instances to be used. All available

services that match **DocumentTranslation** definition are used. For instance, if only two **different** services that match the given definition are found, then the service invocations are performed in a circular way, as illustrated in Figure 4.1.

This example presents the usage, at most, of three different translation services, with three potential different translations. The results are shown in order to establish which translation is more accurate.

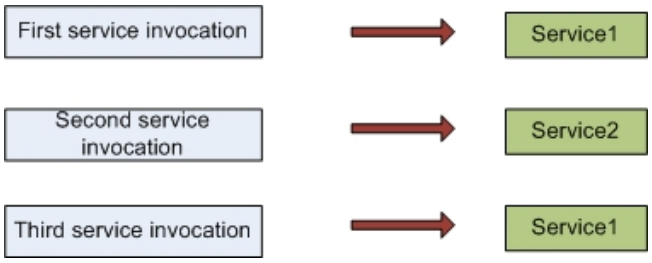

Figure 4.1: Circular service use

Listing 4.5: Translation service example

```
public class ServiceTranslationTest uses all DocumentTranslator {

    String text;
    String lang_in;
    String lang_out;

    public ServiceTranslationTest() {
        this.text = text;
        this.lang_in = lang_in;
        this.lang_out = lang_out;
    }

    public void call(String text, String lang_in, String lang_out) {
        String res1, res2, res3;

        res1 = DocumentTranslator.translate(text, lang_in, lang_out);
        res2 = DocumentTranslator.translate(text, lang_in, lang_out);
        res3 = DocumentTranslator.translate(text, lang_in, lang_out);

        System.out.println("Translated text:\n" + res1 + "\n" + res2 + "\n" + res3);
    }
}
```

4.2 Compiler

The compiler is responsible for processing SAL and SUL components and generate the correspondent Java code ready to be interpreted with the remainder of the application. There are three main steps in this module, SAL processing, SUL processing, and code transformation to generate an application.

To process SAL components a grammar was defined in JavaCC according to the rules shown in Table 4.1. Regarding SUL components, an extension to Polyglot5 was developed to apply a transformation process in order to generate the Java code.

We will begin by describing the technologies used to implement the compiler, followed by a general overview, and how SAL and SUL components are processed.

4.2.1 Technologies

The following technologies were used to develop the compiler module: Java Compiler Compiler (JavaCC) [21], Polyglot5 [33]. This Subsection now describes each of these technologies in detail.

JavaCC

JavaCC is a compiler generator written in Java that works with any VM from version 1.2, it accepts language specifications in BNF-like format as input. The compiler reads a specification that matches a pattern defined in the grammar and produces pure Java code. This tool has the following features: top-down parsers, tree building preprocessor, customizable, document generation, debug, special tokens, and is used by large community.

Top-down parsers allow the use of general grammars that are easier to debug. JavaCC comes with JJTree, an extremely powerful tree building preprocessor. It also offers several options to customize its behavior and the behavior of the generated parsers, such as, the kinds of Unicode processing to perform on the input stream. Extensive debug is available with options like **DEBUG_PARSER**, **DEBUG_LOOKAHEAD** and **DEBUG_TOKEN_MANAGER**, allowing an in-depth analysis.

The user may define tokens as special tokens in the lexical specification and these are ignored during parsing, an example is the processing of comments. JavaCC includes a tool called JJDoc that converts grammar files to documentation files, there is a good suppor of JavaCC, it comes with several examples and documentation that are a great way to get started with.

Parsers generated by JavaCC can clearly point the location of parse errors with complete diagnostic information. JavaCC is the most popular parser generator to develop compiler in Java.

Polyglot5

Polyglot5 is an extension of Polyglot [25], a source-to-source compiler that adds support to the Java 5 programming language. Polyglot is an extensible compiler framework for Java programming language, as default it is simply a semantic checker for Java 1.4, but a developer can extend it and create a new compiler. The framework is implemented using design patterns to promote extensibility, hence new language extensions may be developed without duplicating code from the framework itself.

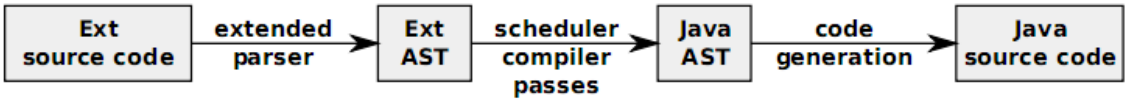

Figure 4.2: Polyglot compilation process

In Polyglot the *base language* is the default supported language (Java 1.4), and the new language is called *language extension*. The language extension accepts source code of a program written in that extension and outputs a new program written in Java 1.4 source code. The steps in compiling a new extension in polyglot are illustrated in Figure 4.2, the name **Ext** stands for language extension. The first step consists in parsing the input source code to produce an abstract syntax tree (AST), Polyglot has an extensible parser generator (PPG) that allows the programmer to define a new syntax as changes to the base language grammar. The programmer can add, modify, or remove productions and symbols of the base grammar. PPG is developed as a preprocessor for CUP LALR parser generator [18]. The resultant AST may have new nodes that represent the new syntax.

The second step lies on a series of compilation passes applied to the AST. Both semantic analysis and translation to Java may include several of such passes. The extension defines the order of these passes to be applied over a single source file. At runtime these are selected by a scheduler. Each compilation pass, if successful, rewrites the AST and is input to the next pass. When all passes are executed the result is a Java AST.

Polyglot5, a language extension for Polyglot, is a semantic checker for the Java 5 programming language. Extensions developed in Polyglot can also be used as base languages for a new extension, we will them use Polyglot5 as our base language to create a Java5 compliant extension.

4.2.2 General Overview

A general overview of this module is illustrated in Figure 4.3. The module performs actions in order to process SAL and SUL components, and generates Java code. The sequence of actions in order to obtain Java code is as follows: first SAL components are processed by a grammar defined in JavaCC, which generates a list of service definitions; secondly the SUL components are processed in three steps:

1. SUL file is parsed for service information such as, service identifier (this associates to a SAL definition), and number of services.

2. The framework interface provides an option allowing to perform, at this stage, a service search. This approach offers the possibility to have a mechanism similar to early/late binding. In early binding the service search is performed before applying the transformation process to the SUL component, and when Java code is generated the services are in cache and ready to be used. With late binding approach, services are discovered only during application execution. This approach is useful for instance, to enable disconnected compilation and to allow to use the application at another time, for example, several days later.

3. Finally, the SUL component is processed by the SedJ module, an extension to Polyglot5 developed to apply a series of transformations in order to generate the final Java code.

As illustrated in Figure 4.3 the processing of the SAL components generates information (service information), that is used during SUL component processing for the purpose of retrieving service properties.

SUL component processing will perform, if selected, a service search and store information about the interface, number of services, properties, and services founds to be used during the transformation process and application execution.

We now present the Java package structure for the compiler module in Figure 4.4.

gui: includes the classes that perform interactions with the framework. The package contains a subpackage called **user**, it includes classes for a graphic interface to aid in the selection and process of SAL and SUL components.

manager: this package is responsible for managing the framework. It dispatches requests from the interface to the correspondent components.

compiler: is responsible for processing SAL and SUL components. SAL components are processed by a JavaCC grammar and generate a list of service definitions.

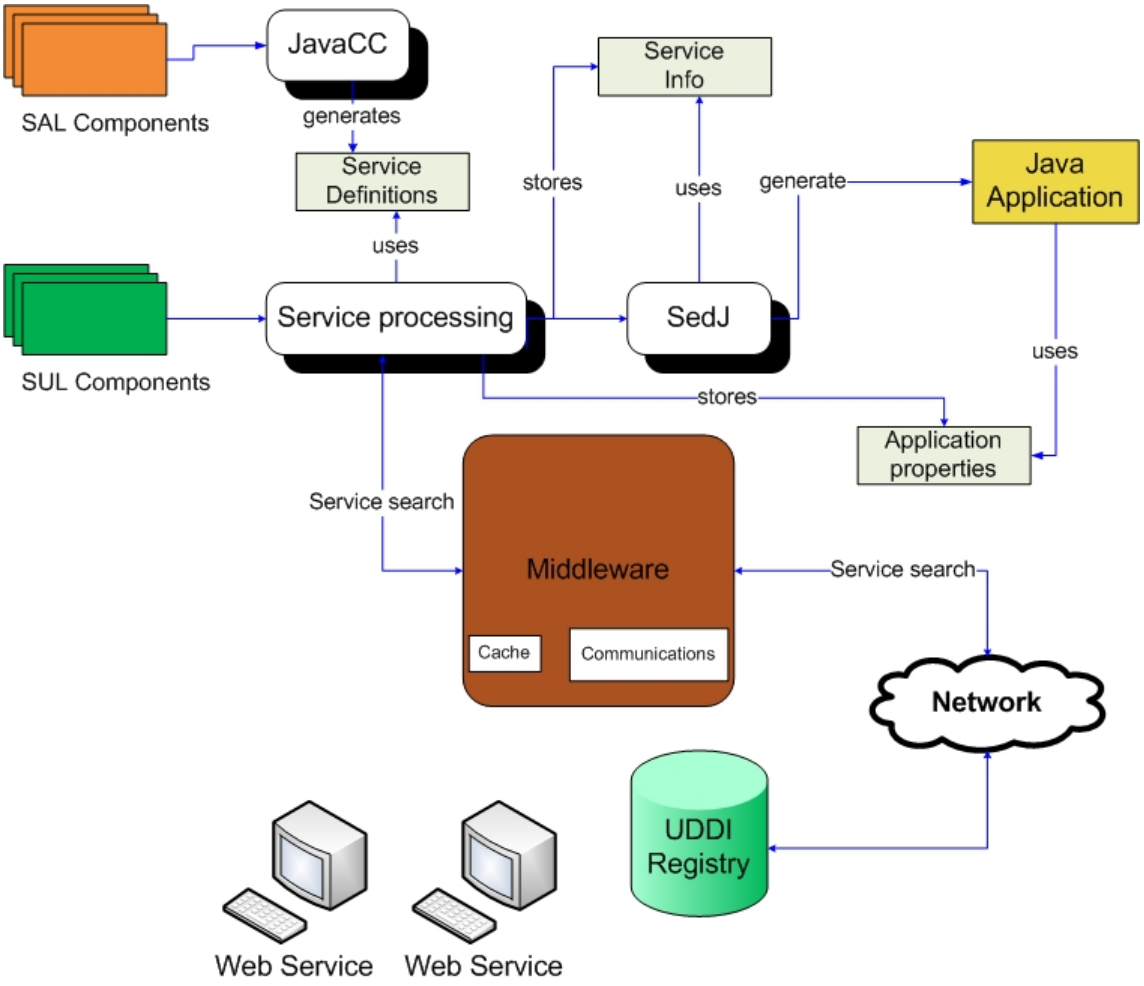

Figure 4.3: Compiler module architecture

SUL components go through two steps in order to generate Java code. This package contains two subpackages, one for each type of component.

compiler.sal: includes classes that perform SAL component processing and generate a list of *Service*. The package includes a subpackage (grammar), containing the rules defined for the JavaCC grammar.

compiler.sal.grammar: includes the classes that parse a grammar defined in JavaCC.

compiler.sul: includes classes to process SUL components in two steps: first, the component is analyzed for service information (identifier and number of services) and performs a service search in a registry, only if that option is selected. Information regarding the interface name, number of existing services and classname are kept in *SedjInfo* structure. Secondly, the SUL component goes through a series of passes, in **polyglot.ext.sedj** package, and uses information stored in *SedjInfo* to generate Java code.

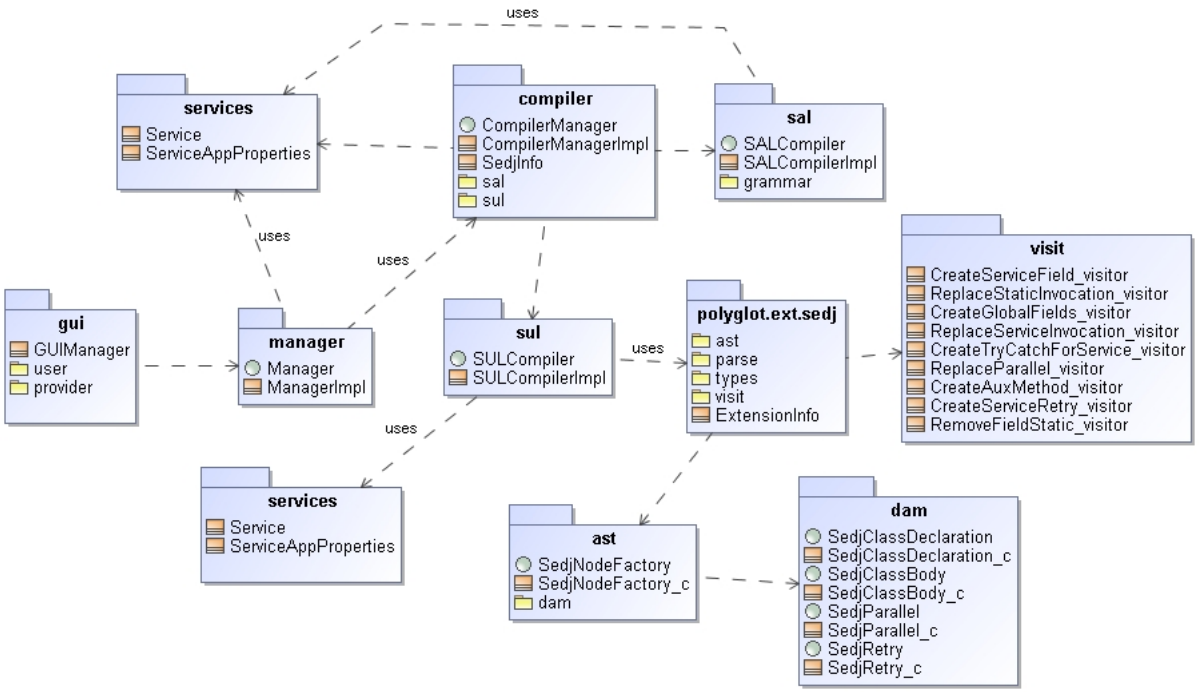

Figure 4.4: Compiler package diagram

polyglot.ext.sedj: this package performs transformation to SUL components and generates Java code. The SUL component is parsed by a grammar extension of a Java grammar specification, then a series of passes are applied, as visitors, to the AST. These passes carry through transformations to generate a file with Java code. The package contains four subpackages: **ast**, **parse**, **types**, and **visit**.

polyglot.ext.sedj.dam: contains classes that correspond to the new nodes created or modified.

polyglot.ext.sedj.visit: includes the classes that correspond to the visitors that perform transforms to the SUL component.

services: includes classes that contain information about services and are used both in the compiler and middleware module.

The compiler is written in the Java programming language and uses the Java 6 library. JavaCC and an extension to Polyglot5 are used to process SAL and SUL components. Our language (SedJ), is implemented as an extension to Polyglot5.

4.2.3 SAL Component Processing

As illustrated in Figure 4.3, SAL components are processed according to a grammar specification defined in JavaCC, and the result is a list of *Service*. The data structure

stores the SAL identifier, the interface identifier, and the service properties. It is used later during SUL component processing to retrieve service properties.

The *Service* class is listed in Appendix A.3, it is used to store information of SAL components, such as, SAL name, interface name, and properties. The structure is used during SUL component processing in order to retrieve service properties.

An example of a SAL component is illustrated in Listing 4.6.

Listing 4.6: SAL example

```
service BWPrinter implements Printer {
    color = "black",
    type = "laser",
    paper = "a4"
}
```

4.2.4 SUL Component Processing

As illustrated in Figure 4.3, the SUL component goes through two phases. In the first phase, the file is parsed for the number of services and the service identifier. The latter is used to match a SAL definition, thereby obtaining its interface, and list of attributes. The interface is loaded into the execution environment using the Java classloader mechanism.

The framework provides the possibility to perform a service search at this stage, a connection is made to the middleware in order to retrieve the services. Service search is realized with base on interface name, and the results are filtered according to properties defined in SAL definition. This approach is useful for places where several services of the same type are available. The discovery process is described in SubSection 4.3.3.

Listing 4.7: SUL example

```
public class ServicePDFTest uses DocumentConverter {

    byte[] res = null;

    public ServicePDFTest() {
    }

    public byte[] call(byte[] doc) {
        res = DocumentConverter.convert(doc, "pdf");
        return res;
    }
}
 catch(Exception e) {
    retry 2 in 4;
}
```

Information regarding the interface, and number of service instances is stored in a data structure called *SedjInfo*. This class is listed in Appendix A.4, it is used to store

information of interface name, number of services, and classname. This information is used when applying transformations to the SUL component. Finally, as illustrated in Listing 4.7, service invocations are performed upon service identifiers, but they are, now, replaced for the interface name, as illustrated in Listing 4.8. This procedure is required, before submitting the SUL component to the Sedj module, in order to be parsed correctly by the developed Java 5 compiler extension.

Listing 4.8: SUL example before applying transformations

```
public class ServicePDFTest uses DocumentConverter {

    byte[] res = null;

    public ServicePDFTest() {
    }

    public byte[] call(byte[] doc) {
        res = Converter.convert(doc, "pdf");
        return res;
    }
}
 catch(Exception e) {
    retry 2 in 4;
}
```

In the second phase, the SUL component is subject to a transformation process by the Sedj module. First, to integrate the new constructs defined in the SUL concrete syntax some modifications were required in Java grammar specification, namely:

Class header: Extension of the Java class header specification with service use (**uses**) and service allocation abstractions (**volatile**, number of services, and **all**). These abstractions are defined at the same level as **implements** and **extends** keywords.

Class body: Extension of Java class body to add an exception handling mechanism that catches the whole service execution inside **uses**. The **catch** abstraction can now be used at the end of a class definition.

Block: Extension of Java block to allow parallel composition (|) of statements and expressions.

Statements: Extension of Java statements to add a **retry** statement for service re-execution. However, this can only be used inside the new catch mechanism.

After modifying the Java grammar and adding new nodes, a compilation process takes place to generate the final Java code. The core of this process is a series of compilation passes applied to the abstract syntax tree. New passes were created to perform transformations on the AST. The *pass scheduler* selects the passes to run over the AST,

in an order defined by the language extension, guaranteeing that dependencies are not violated. A pass consists on a execution of a visitor that realizes transformations on nodes of the AST, these transformations are executed either when entering or when leaving a node. The implemented visitors only perform transformations when leaving nodes, this approach has the benefit of when leaving a node the compiler will not visit it again.

Next, we present a description of the visitors created to apply the desired transformations:

***CreateServiceField*:** creates a global field to represent the interface for the service being used.

***ReplaceStaticInvocation*:** visits each service invocation and replaces the invocation performed with the interface name with the identifier created in the previous visitor.

***CreateGlobalFields*:** creates the global fields required by the application and instantiates them with the appropriate value. For instance, a list of service instances, a list of threads required for parallel execution.

***ReplaceServiceInvocation*:** replaces the current service invocation with a list of the service interface type.

***CreateTryCatchForService*:** creates a **try catch** mechanism for each service invocation, and the decisions to perform if it fails.

***ReplaceParallel*:** replaces, for each defined sequence of operations, the parallel construct with threads. This visitor only performs transformation if parallel composition is found in the input file.

***CreateAuxMethod*:** generates the code to start thread execution regarding parallel composition. As in the previous visitor, this visitor only performs transformations if parallel composition is found.

***CreateServiceRetry*:** creates the service re-execution operation, it only performs transformations if a **retry** statement is found or if the service use is declared as **volatile**.

***RemoveFieldsStatic*:** This visitor removes the fields created by *CreateServiceField* visitor.

After all visitors and remaining passes are executed the result is a Java AST with all transformations applied. The last operation executed by SedJ is to call the Java compiler .

We now describe the transformations executed by each visitor on a SUL component. Listing 4.9 and 4.10 illustrate, respectively, the SAL definition and the correspondent interface for a printer service. Given the SUL example in Listing 4.11, Listing 4.12 presents the interface of the service being used, it is generated by the *CreateServiceField* visitor. Listing 4.13 illustrates the transformation performed by *ReplaceStaticInvocation* visitor. Service invocations that were performed with the interface name are now replaced by a valid identifier, service operations are not static and so they need to be substituted to obtain a correct Java file.

Listing 4.9: SAL ColorPrinter

```
service ColorPrinter implements Printer {
    color = "color",
    type = "laser",
    paper = "a4"
}
```

Listing 4.10: Printer interface

```
public interface Printer {
    void print(byte[] document);
}
```

Listing 4.11: SUL example

```
1  public class ServicePrinterTest uses 2 ColorPrinter {
2
3      public byte[] document = null;
4
5      public ServicePrinterTest() {
6      }
7
8      public void call(byte[] doc) {
9          this.document = doc;
10
11         Printer.print(document);
12
13         //parallel composition
14         { Printer.print(document); } | { Printer.print(document); }
15     }
16 }
17  catch(Exception e) {
18     //restart service execution
19     retry 1 in 3;
20 }
```

Listing 4.12: Service interface service field

```
private Printer auxiliar_interface;
```

Listing 4.13: Service invocations

```
1  public void call(byte[] doc) {
2      this.document = doc;
3
4      auxiliar_interface.print(document);
5      {
6          {
7              auxiliar_interface.print(document);
8          }
9          {
10             auxiliar_interface.print(document);
11         }
12     }
13 }
```

In Listing 4.14 the *CreateGlobalFields* visitor creates the global fields and instantiates them in the class constructor. In line 3, **fail** represents if a service invocation failed. The **srvs** field represents a list of the service instances previously discovered and that are available for service invocation, the **failed_srvs** field is used to store the index of the service instance from **srvs** that fail causing the exception to be triggered. The application will communicate with the middleware through the **app_manager** field. The **passed**, and **threads** are used for parallel composition.

The constructor instantiates the fields and establishes a connection with the middleware to retrieve the service bindings.

Listing 4.14: Global fields and constructor

```
1  public class ServicePrinterTest {
2
3      //automatically generated fields
4      public boolean fail = false;
5      private java.util.List<Printer> srvs;
6      private java.util.List<Integer> failed_srvs;
7      private middleware.application.ApplicationManager app_manager;
8      public boolean passed = false;
9      private java.util.List<Thread> threads;
10     private Printer auxiliar_interface;
11
12     //class attributes
13     byte[] document = null;
14
15     public ServicePrinterTest() {
16         super();
17         failed_srvs = new java.util.LinkedList<Integer>();
18         app_manager = new middleware.application.ApplicationManagerImpl("ServicePrinterTest");
19         threads = new java.util.LinkedList<Thread>();
20         srvs = app_manager.getServiceBindings();
21     }
```

Listing 4.15 illustrates the service invocation being replaced by **srvs** field. Lines 3-10 represent parallel service invocation.

Listing 4.15: Replace service invocation

```
public void call(byte[] doc) {
    this.document = doc;

    srvs.get(0).print(document);
    {
    //begin parallel
        {
            srvs.get(1).print(document);
        }
        {
            srvs.get(0).print(document);
        }
    }
    //end parallel
}
```

Listing 4.16 shows the transformation, generated by *CreateTryCatchForService*, of the new catch mechanism. Each service invocation is protected, if an error situation occurs, the index corresponding the failed service is stored, it will be used for service replacement.

Listing 4.16: Try-catch for service invocation

```
public void call(byte[] doc) {
    this.document = doc;

    try {
        srvs.get(0).print(document);
    } catch (Exception e) {
        fail = true;
        if (!failed_srvs.contains(0)) {
            failed_srvs.add(0);
        }
    }
    //begin parallel
    {
        {
            try {
                srvs.get(1).print(document);
            } catch (Exception e) {
                fail = true;
                if (!failed_srvs.contains(1)) {
                    failed_srvs.add(1);
                }
            }
        }
        {
            try {
                srvs.get(0).print(document);
            } catch (Exception e) {
                fail = true;
                if (!failed_srvs.contains(0)) {
                    failed_srvs.add(0);
                }
            }
        }
```

```
    }
    //end parallel
}
```

Listing 4.17 illustrates the transformations executed for processing parallel composition. Threads are created by the *ReplaceParallel* visitor. *CreateAuxMethod* visitor creates the code for threads to **start** and **join**. An auxiliary method is created for thread generation.

Listing 4.17: Parallel composition transformation

```
public void call(byte[] doc) {
    this.document = doc;

    if (!passed)
        threadGen();
    try {
        srvs.get(0).print(document);
    } catch (Exception e) {
        fail = true;
        if (!failed_srvs.contains(0)) {
            failed_srvs.add(0);
        }
    }

    for (int i = 0; i < threads.size(); i++) {
        threads.get(i).start();
    }

    for (int i = 0; i < threads.size(); i++) {
        try {
            threads.get(i).join();
        } catch (InterruptedException e) { }
    }
}

private void threadGen() {
    passed = true;
    Thread t0 = new Thread(new Runnable() {
        public void run() {
            try {
                srvs.get(1).print(document);
            } catch (Exception e) {
                fail = true;
                if (!failed_srvs.contains(1)) {
                    failed_srvs.add(1);
                }
            }
        }
    });
    threads.add(t0);
    Thread t1 = new Thread(new Runnable() {
        public void run() {
            try {
                srvs.get(0).print(document);
            } catch (Exception e) {
```

```
                fail = true;
                if (!failed_srvs.contains(0)) {
                    failed_srvs.add(0);
                }
            }
        }
    });
    threads.add(t1);
}
```

Listing 4.18 illustrates the code generated by *CreateServiceRetry* visitor. This is created for the retry expression present in line 18 of Listing 4.11. A new method is created to perform service invocation. The global field representing the interface *Printer* is removed by the *RemoveFieldsStatic* visitor.

Listing 4.18: Service retry

```
private void _call(int attempts, int time, byte[] doc) {
    ...

    if (fail && attempts > 0) {
        fail = false;
        for (int i = 0; i < failed_srvs.size(); i++) {
            Object obj = app_manager.replaceServiceBinding(failed_srvs.get(i));
            srvs.set(failed_srvs.get(i), (Printer) obj);
        }

        failed_srvs = new java.util.LinkedList<Integer>();
        try {
            Thread.sleep(time);
        } catch (InterruptedException e) { }
        _call(attempts - 1, time);
    }
}

public void call(byte[] doc) {
    _call(1, 3000, doc);
}
```

The entire transformation for the example in Listing 4.11 is listed on Listing 4.20.

Listing 4.19: Generated code

```
public class ServicePrinterTest {

    public boolean fail = false;
    private java.util.List<Printer> srvs;
    private java.util.List<Integer> failed_srvs;
    private middleware.application.ApplicationManager app_manager;
    public boolean passed = false;
    private java.util.List<Thread> threads;
    byte[] document = null;

    public ServicePrinterTest() {
        super();
        failed_srvs = new java.util.LinkedList<Integer>();
```

```
        app_manager = new middleware.application.ApplicationManagerImpl("ServicePrinterTest");
        threads = new java.util.LinkedList<Thread>();
        srvs = app_manager.getServiceBindings();
    }

    private void _call(int attempts, int time, byte[] doc) {
        this.document = doc;
        if (!passed)
            threadGen();
        try {
            srvs.get(0).print(document);
        } catch (Exception e) {
            fail = true;
            if (!failed_srvs.contains(0)) {
                failed_srvs.add(0);
            }
        }

        for (int i = 0; i < threads.size(); i++) {
            threads.get(i).start();
        }

        for (int i = 0; i < threads.size(); i++) {
            try {
                threads.get(i).join();
            } catch (InterruptedException e) { }
        }
        passed = false;
        threads = new java.util.LinkedList<Thread>();

        if (fail && attempts > 0) {
            fail = false;
            for (int i = 0; i < failed_srvs.size(); i++) {
                Object obj = app_manager.replaceServiceBinding(failed_srvs.get(i));
                srvs.set(failed_srvs.get(i), (Printer) obj);
            }
            failed_srvs = new java.util.LinkedList<Integer>();
            try {
                Thread.sleep(time);
            } catch (InterruptedException e) { }
            _call(attempts - 1, time, document);
        }
    }

    private void threadGen() {
        passed = true;
        Thread t0 = new Thread(new Runnable() {
            public void run() {
                try {
                    srvs.get(1).print(document);
                } catch (Exception e) {
                    fail = true;
                    if (!failed_srvs.contains(1)) {
                        failed_srvs.add(1);
                    }
                }
            }
```

```
        });
        threads.add(t0);
        Thread t1 = new Thread(new Runnable() {
            public void run() {
                try {
                    srvs.get(0).print(document);
                } catch (Exception e) {
                    fail = true;
                    if (!failed_srvs.contains(0)) {
                        failed_srvs.add(0);
                    }
                }
            }
        });
        threads.add(t1);
    }

    public void call(byte[] doc) {
        _call(1, 3000, doc);
    }
}
```

We now present what part of the generated code correspond to the new abstractions integrated in the Java grammar specification:

uses: In Listing 4.20, line is the result of using this abstraction, a list of the given identifier is created.

retry X in Y: In Listing 4.21, line 9-12 is the result of using service re-execution. The application waits for Y time and afterwards a new attempt on the service is performed.

volatile: allows an implicit retry mechanism. If this abstraction is used, then lines 9-11 in Listing 4.21 are not generated, only a invocation to the method (call()) would appear instead.

|(parallel): Listing 4.22 illustrates the generated code of using the parallel composition abstraction. For each sequence of operations in parallel a new thread is created.

catch: In Listing 4.21 the lines 6-13, and 19-26 are the result of using the catch abstraction, a try-catch mechanism is created for each service invocation.

Listing 4.20: Generated Java code example - Service use

```
1 public class ServicePrinterTest {
2
3     public boolean fail = false;
4     private java.util.List<Printer> srvs;
5     private java.util.List<Integer> failed_srvs;
```

```
    private middleware.application.ApplicationManager app_manager;
    public boolean passed = false;
    private java.util.List<Thread> threads;
    byte[] document = null;
    ...
}
```

Listing 4.21: Generated Java code example - Service retry

```
...
if (fail && attempts > 0) {
    fail = false;
    for (int i = 0; i < failed_srvs.size(); i++) {
        Object obj = app_manager.replaceServiceBinding(failed_srvs.get(i));
        srvs.set(failed_srvs.get(i), (Printer) obj);
    }
    failed_srvs = new java.util.LinkedList<Integer>();
    try {
        Thread.sleep(time);
    } catch (InterruptedException e) { }
    _call(attempts - 1, time, doc);
}
...
```

Listing 4.22: Generated Java code example - Parallel composition

```
...
private void threadGen() {
    passed = true;
    Thread t0 = new Thread(new Runnable() {
        public void run() {
            try {
                srvs.get(1).print(document);
            } catch (Exception e) {
                fail = true;
                if (!failed_srvs.contains(1)) {
                    failed_srvs.add(1);
                }
            }
        }
    });
    threads.add(t0);
    Thread t1 = new Thread(new Runnable() {
        public void run() {
            try {
                srvs.get(0).print(document);
            } catch (Exception e) {
                fail = true;
                if (!failed_srvs.contains(0)) {
                    failed_srvs.add(0);
                }
            }
        }
    });
    threads.add(t1);
}
```

4.3 Middleware

The middleware is a software layer that provides an interface, which the code generated by the compiler will use, to make requests, such as, service binding, discovery and recovery. Also, the compiler module interacts with the middleware in order to request a service discovery process during the compiler phase.

We will begin by describing a general overview of this module, followed by a characterization of the service interface mapping mechanism used in service discovery, and a detailed description of the operations available in the middleware.

4.3.1 General Overview

A general overview of this module is illustrated in Figure 4.5, the middleware provides an API for communication with the application and is responsible for replying to: service binding, and replacement requests. The middleware, besides the API, provides a service discovery process used during the compiler phase in order to retrieve the number of services available.

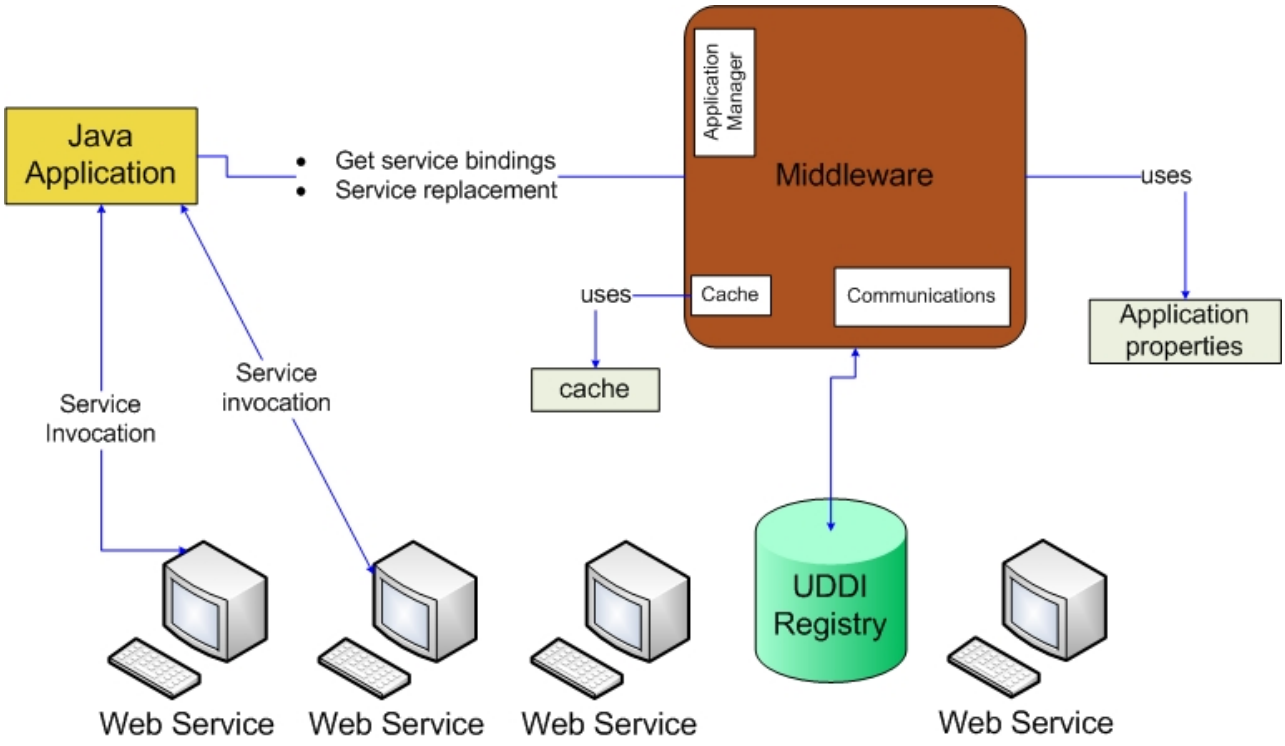

Figure 4.5: Middleware module architecture

When the application is set for execution the first operation, and compulsory, is to request service bindings to the middleware. Secondly, whenever a failure occurs, the

application requests a service binding replacement. In this scenario the middleware triggers a new service discovery process in order to provide an appropriate service replacement according to its properties.

Services are maintained in a local cache and used during the discovery process, the cache is updated whenever a service fails or a new binding is established. This approach ensures less communications with the registry. Information regarding interface, properties and services to be used were stored in *Application properties* during the compiler phase and are now used during application execution to obtain service bindings and replacement.

Services are registered in a jUDDI registry, and communication between the application and services is realized with support on JAX-WS [1]. Service discovery between the middleware and jUDDI are performed resorting to UDDI4J [2] library. A java library that provides an API that performs operations on jUDDI registry.

We now present the Java package structure for the middleware module in Figure 4.6.

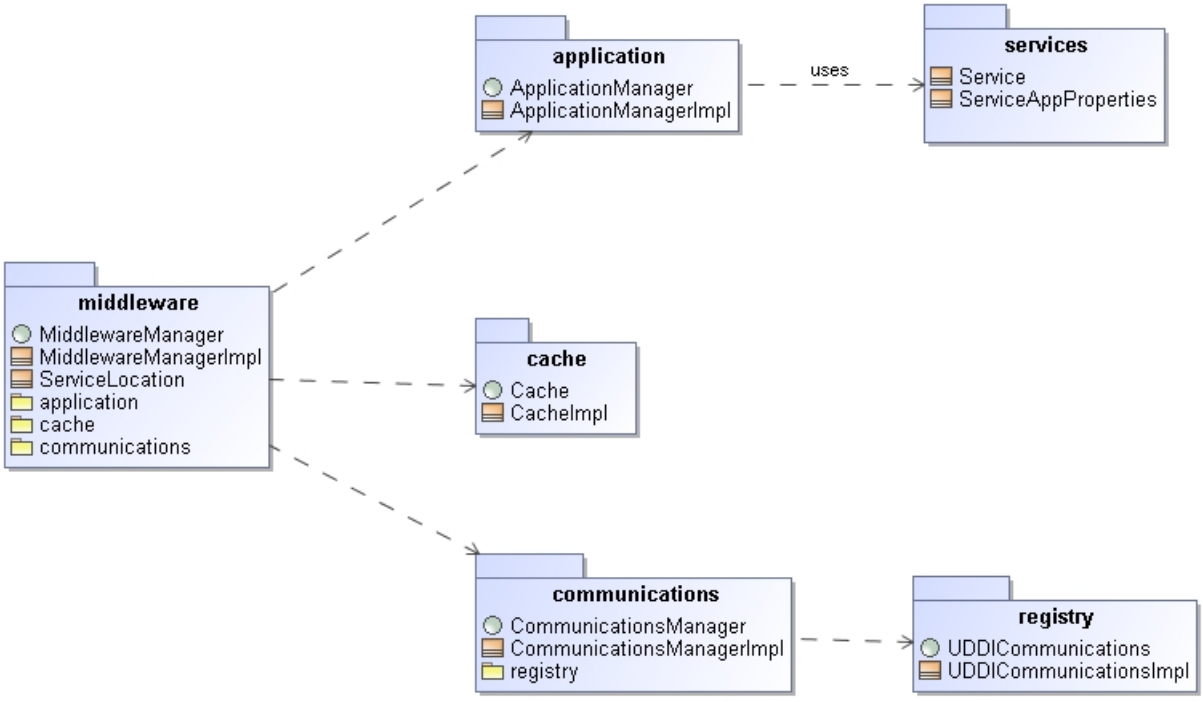

Figure 4.6: Middleware package diagram

application: is responsible for interaction with the generated application, namely, to retrieve service bindings when application is set for execution, and for binding

[1] https://jax-ws.dev.java.net/
[2] Library available at http://uddi4j.sourceforge.net/

replacement in service failure scenarios. *ApplicationManagerImpl* class uses *ServiceAppProperties* structure to retrieve, from local storage, the service interface, properties and list of services to be used.

cache: cache mechanism for services retrieved from the registry. Services are maintained in a structure (*Cache*), this is composed by the interface name and a list of services, being used, that implement that interface. In service failure or service discovery situations the cache is updated.

communications: handles communications with service registries, currently only communication with UDDI registry is supported. This package contains a subpackage, **registry**.

registry: performs communication with jUDDI registry, namely, registry connection, service storage and discovery.

The *ServiceLocation* class is listed on Appendix A.2, it is used to store the WSDL service address and the service properties location.

The *CacheImpl* class is listed on Appendix A.5, it is used to store the interface name and the list of services that implement that interface.

To store service locations the middleware uses a Hashtable<String, ServiceLocation>, the String is the service name and ServiceLocation contains the information about its location. To store service instances in cache the middleware uses a Hashtable<String, Cache>, the String represents the interface name, and Cache contains the information about what services implement that interface.

4.3.2 Interface Mapping

jUDDI registry only provides storage for service location, and discovery is only performed in a name basis. To overcome this limitation, services stored in jUDDI also include, besides the WSDL address, the location of a XML file containing service properties, as illustrated in Listing 4.23.

Listing 4.23: Service properties

```
<service>ColorPrinter
    <properties>
        <type>laser</type>
        <brand>hp</brand>
        <paper>a4;a5;letter</paper>
    </properties>
</service>
```

A service discovery process in the framework will return a structure (*ServiceLocation*) with information regarding the WSDL and service properties location, both in XML. Given the SAL example in Listing 4.24, the discovery process would compare the properties defined in SAL definition against the properties in the XML of Listing 4.23. If the given properties match the ones presented in the XML document then the service is a match. In this scenario, the *ColorPrinter* service is a match for the properties defined in SAL example.

Listing 4.24: Service properties

```
service BWPrinter implements Printer {
    type = "laser",
    paper = "a4"
}
```

This approach provides our framework with a more advanced discovery mechanism where service interface name is used to search for services and properties to limit the result space.

4.3.3 Service Discovery

The service discovery process consists on the following steps:

1. A connection is established with the jUDDI registry through an invocation of the *searchServiceUDDI* method from the **communications** package. The interface name and number of services are passed as method arguments and a list of *ServiceLocation* is returned as a result of the search.

2. From the returned list, the *ServiceLocation* structure has the information regarding WSDL address and service properties location. Now a comparison is performed between the given properties and the service properties described in the XML file. This interface mapping between services is explained in SubSection 4.3.2.

3. If the properties are matched, then a four step process is performed:

   - service stubs are generated from the service WSDL address.

   - the interface correspondent to the service is extended with **extends** <service interface name>. Given the example in Listing 4.25, if stubs are generated for a ColorPrinter service and the given interface during SAL processing is Printer, then the generated interface (ColorPrinter) is transformed in the code illustrated in Listing 4.26.

   - compile the generated classes.

- create a jar archive of the generated service classes. This approach of creating a jar file benefits from only loading one file to the execution environment, instead of loading each generated class.

4. The services location structure (Hashtable<String, ServiceLocation>), is updated with a new service name and its service location. The cache structure is updated for the given interface with a new service.

5. The steps from 2 to 4 repeat until all services returned from the jUDDI registry are processed.

6. Finally, the cache and service location are stored locally, and a list of service names is returned.

Listing 4.25: Inteface extension example

```
public interface ColorPrinter {

...

}
```

Listing 4.26: Interface extension example

```
public interface ColorPrinter extends
    Printer {

...

}
```

4.3.4 Service Bindings

The first operation executed by the application when in execution is to request its service bindings to the middleware. Listing 4.27 illustrates the constructor of the generated code of a given SUL component.

Listing 4.27: Class constructor

```
1 ...
2 public ServicePrinterTest() {
3     super();
4     failed_srvs = new java.util.LinkedList<Integer>();
5     app_manager = new middleware.application.ApplicationManagerImpl("ServicePrinterTest");
6     threads = new java.util.LinkedList<Thread>();
7     srvs = app_manager.getServiceBindings();
8 }
9 ...
```

In line 4, the application establishes a connection with the middleware and passes the class identifier as an argument. The identifier ("ServicePrinterTest") is used by the middleware to load the *ServiceAppProperties* structure. The *ServiceAppProperties* class is listed on Appendix A.1, it is used to store, locally, information regarding interface name, classname, number of services, properties, and a list of services stubs to be used. This information is used afterwards when the generated Java code is executing.

In line 7, the application requests its service bindings invoking the *getServiceBindings* operation on the middleware. The *ServiceAppProperties* structure is loaded when a connection is performed (Line 6 in Listing 4.27) to the middleware by the application, and information regarding this service execution is now available. This method consists on the following steps:

1. The *ServiceAppProperties* contains the information regarding the list of "available" services that match the *kind* required. First the middleware checks the size of that list in order to know if a service search is required.

2. If the list size is bigger than zero, then the service stubs are loaded by the Java classloader mechanism into the execution environment and new instances are created for each service stub.

3. If the list contains no elements, then a service discovery process is required:

   - First, the cache and services location mechanisms are loaded to memory.

   - A lookup is performed in the cache mechanism to retrieve the cache structure of the given interface name. If no match is found, then that interface was not yet registered. If a match is found, then we retrieve the service list that implements the given interface.

   - For each service a comparison is performed between the given properties (present in ServiceAppProperties) and the properties of the service obtained from the cache mechanism (service properties are defined in a XML file whose location is defined in services location mechanism). If a match occurs, then the service name is stored in a list.

   - If the number of services is not fulfilled (obtained from ServiceAppProperties), a discovery is performed in jUDDI registry. The discovery process is described in SubSection 4.3.3. The returned services are added to the list of service names.

   - Finally, if the number of services still is not fulfilled, then existing services are copied and added until the desired number is met. If no matching services are found, then the application stops its execution.

4.3.5 Service Replacement

In scenarios where bindings are broken, the application requests the middleware a new service binding. Listing 4.28 shows sections, of the generated code for a SUL component, related to service replacement.

Listing 4.28: Service replacement mechanism

```
1  ...
2  try {
3      srvs.get(0).print(document);
4  } catch (Exception e) {
5          fail = true;
6          if (!failed_srvs.contains(0)) {
7              failed_srvs.add(0);
8          }
9      }
10 ...
11 ...
12 if (fail && attempts > 0) {
13     fail = false;
14     for (int i = 0; i < failed_srvs.size(); i++) {
15         Object obj = app_manager.replaceServiceBinding(failed_srvs.get(i));
16         srvs.set(failed_srvs.get(i), (Printer)obj);
17     }
18
19     failed_srvs = new java.util.LinkedList<Integer>();
20     try {
21         Thread.sleep(time);
22     } catch (InterruptedException e) { }
23
24     _call(attempts - 1, time, doc);
25 }
26 ...
```

For instance, in line 3 a service with the **print** operation is invoked. If the connection with the service is lost, then a new binding must be established (line 15). The service replacement process consists on the following steps:

1. The middleware retrieves information regarding the interface name and properties from *ServiceAppProperties* structure.

2. A service discovery process takes place with the interface name, properties, and only one service is requested. The service discovery process is described in Sub-Section 4.3.3.

3. The Java classloader mechanism loads the returned service stub and creates a new service instance.

4. Finally, the cache mechanism updates its information, by replacing in the given interface the service name that failed, and in the *ServiceAppProperties* structure, the service that failed is replaced by the new service name in the position provided by the application.

To summarize, in this chapter we presented the implementation of SeDeUse framework. Firstly, it was introduced the concrete syntax for both, SAL and SUL components. Secondly, a description of the compiler regarding the technologies used and the

process in order to generate Java code (in more detail the transformation applied to the SUL component). Finally, a description of the middleware layer with detailed information on how syntactically different services are semantically equivalent, and the operations provided by the middleware. In the next chapter, we present three scenarios of possible use for the Sedj language.

5

Applications/Case Studies

This chapter reports the results of three case studies aimed at evaluating the Sedj framework. These studies assess the framework's programming abstractions supported by a middleware layer that hides the nature of using service-oriented computing in dynamic environments.

The first scenario consists in a document manipulation service, where a document is converted from its original digital format into a specified format, then is translated to another language, finally the document is sent for a printing service. The second scenario involves service operations available in a highway road. The third scenario demonstrates services that are used in airport terminals.

The chapter begins with a description of the experimental settings for the framework's language evaluation, followed by the presentation of three scenario examples. Finally, a conclusion is presented regarding the framework's evaluation.

5.1 Setup/Experimental Settings

The experimental evaluation was executed on the Computer Science Department at Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa.

Three applications were developed to demonstrate the framework's utility, regarding the programming abstractions for service awareness and use, sustained by a middleware layer for support in dynamic environment.

The settings for our experimental evaluation is as follows: one laptop to execute the Sedj framework; one host computer with three jUDDI registries deployed in the

Apache Tomcat framework; and a variable number of host computers executing Web Services that establish connection to services. JUDDI uses a MySQL database to maintain registry data.

5.2 A Document Manipulation Service

In this case study, the scenario is a composition of services to manipulate a text document. Firstly, the contents of the text document contents are translated to a specified language by a document translation service. Then, the document is given to another service in order to be converted to a format suitable for the available printers. Finally, the resulting document is sent to the available printer services to be printed.

The goal of this scenario is to demonstrate the use of service abstraction (**uses**), parallel composition (|), service allocation (**all**), exception handling (**catch**), and service re-execution (**retry**).

We now present the settings for this case study, also the SAL and SUL components used, and the main class file with the composition for this demonstration.

5.2.1 Scenario Settings

The execution environment setting for this scenario is as follows. One jUDDI registry that contains three available printer services, one document translation service, and two document converter services. At runtime, after invoking all services, the bindings for the translation service and the printers are broken. The application is expected to recover and create new bindings to other equivalent services registered in jUDDI.

In this subsection, we present the SAL and SUL components for this scenario, and the main file coordinating the services. Listing 5.1 illustrates a possible SAL definition for a document translation service. The SAL definition implements the *Translation* interface, presented in Listing 5.2.

Listing 5.1: SAL - TextTranslation

```
service TextTranslation implements Translation {
    type = "doc",
    input in {"english", "french", "german"},
    output in {"portuguese"}
}
```

Listing 5.2: Translation interface

```
public interface Translation {
    byte[] translate(byte[] doc, String in_lang, String out_lang);
}
```

The SUL file to use the document translation service is presented on Listing 5.3, and the correspondent generated code for this component is illustrated on Appendix A.6.

Listing 5.3: SUL - TestTranslator

```
import interfaces.Translation;

public class TestTranslator uses Translation {

    public byte[] result;

    public TestTranslator() {
    }

    public byte[] call(byte[] document, String input_lang, String output_lang) {
        result = Translation.translate(document, input_lang, output_lang);
        return result;
    }
}
catch(Exception e) {
    retry 1 in 3;
}
```

Listing 5.4 illustrates a SAL definition for a possible document converter service, it implements the *Converter* interface shown in Listing 5.5.

Listing 5.4: SAL - DocumentConverter

```
service DocumentConverter implements Converter {
    input in {"doc", "docx"},
    output = "pdf"
}
```

Listing 5.5: Converter interface

```
public interface Converter {
    byte[] convert(byte[] doc, String extension);
}
```

Listing 5.6 illustrates the SUL file for a document converter application. The generated code for this SUL component is listed on Appendix A.7.

Listing 5.6: SUL - TestConverter

```
import interfaces.Converter;

public class TestConverter uses volatile Converter {

    public byte[] res;

    public TestConverter() {
    }

    public byte[] call(byte[] doc, String extension) {
        res = Converter.convert(doc, extension);
        return res;
    }
}
```

Listing 5.7 illustrates a SAL definition for a possible document printer service. The SAL definition implements the *Printer* interface presented in Listing 5.8.

Listing 5.7: SAL - ColorPrinter

```
service ColorPrinter implements Printer {
    color = "colors",
    type = "laser"
}
```

Listing 5.8: Printer interface

```
public interface Printer {
    void print(byte[] document);
}
```

Listing 5.9 illustrates the SUL file for a printer application. The correspondent generated code for this component is illustrated on Appendix A.8.

Listing 5.9: SUL - TestPrinter

```
import interfaces.Printer;

public class TestPrinter uses all Printer {
    public byte[] doc;

    public TestPrinter() {
    }

    public void call(byte[] doc) {
        this.doc = doc;

        //parallel composition
        {Printer.print(doc);} | {Printer.print(doc);}
    }
}
catch(Exception e) {
    //re-execute service, waits 4 seconds then re-executes once
    retry 1 in 4;
}
```

5.2.2 Execution

Listing 5.10 illustrates the main file containing the composition of services and sequence of actions that are used in this scenario.

Listing 5.10: DocumentManipulationScenario class

```
import java.io.File;

public class DocumentManipulationScenario {

    public void run(String filename) {
        File f = new File(filename);
```

```
7
8           TestTranslator translator = null;
9           TestConverter converter = null;
10          TestPrinter printer = null;
11
12          //get text from document
13          byte[] doc = Utils.getBytesFromFile(f);
14
15          //translate to portuguese
16          translator = new TestTranslator();
17          doc = translator.call(doc, "en", "pt");
18
19          //convert to pdf
20          converter = new TestConverter();
21          doc = converter.call(doc, "pdf");
22
23          //send to printer
24          printer = new TestPrinter();
25          printer.call(doc);
26
27          //breaking the binding for translate and printer
28          //calling the services again
29
30          doc = translator.call(doc, "en", "pt");
31          printer.call(doc);
32      }
33
34      public static void main(String[] args) {
35          new DocumentManipulationScenario().run(args[0]);
36      }
37  }
```

The *DocumentScenarioManipulation* class defines a possible configuration for using the three generated classes of services. First, the translation service is executed, it translates the given document, in byte array, from English to Portuguese language, returning also a byte array. In line 16 the **new** construct creates a new instance of *TestTranslator* class and obtains service bindings from the middleware. Then in line 17 the translation service is executed, returning the document in the desired language.

Secondly, the document is sent to a document converter service, lines 20-21, in order to convert the document from **doc** to **pdf** format. Finally, the document is sent to a printing service. The *TestPrinter* class demonstrates the use of parallel composition (|), the document is sent to print in two different printers at the same time.

Service Discovery at Runtime

In this case, the *TestPrinter* did not perform a service search during the compilation phase. So, when a new instance of this class is created the **getServiceBindings()** operation in line 18 of Appendix A.8 involves a service discovery process to retrieve

available service instances. Two different printers are found that match given properties.

Handling Broken Bindings

After executing all three services the bindings for the translation and printer services are broken, the Web Services currently executing translation and printing operations are terminated. Invoking these services again will result in failure, lines 21-26 on Appendix A.6 of *TestTranslator* class show that when a service invocation failure occurs, the index of service instances list is stored and a boolean variable is set to **true**. The former is used in the middleware during service replacement to replace the failing instance, in the *ServiceAppProperties* class, with the new replacement. The latter is used in line 27 to check if a failure occurred, in order to trigger the service binding replacement process (lines 30-31). A new binding is established for the translation service and the service is executed as if no error occurred for the user.

The same process is realized in the *TestPrinter* class, lines 40-41 on Appendix A.8 show the service replacement. Regarding the printer service, the registry returns only one service different from the services that failed. This new instance is used in both invocations of the parallel composition. The document is sent twice for printing in the same printer. Only during document printing we became aware that just one printer service is available.

The case study demonstrated the use of service abstraction, parallel composition, service allocation, exception handling and service re-execution.

5.3 Road services

The case study aims at evaluating the use of services possibly available through sensors that are spread in highways, invoking operations such as, traffic report, get travel assistance, how much far is the next gas station.

The goal in this scenario is to demonstrate the use of service abstraction (**uses**), and implicit exception handling mechanism (**volatile**).

We now present the settings for this case study, also the SAL and SUL components used, and the main class file with the composition for this demonstration.

5.3.1 Scenario Settings

For this scenario the settings involve two jUDDI registries, each one containing a service that represent different road sections. During runtime after executing several op-

erations from one service we simulate a road change, from one highway to another road, by replacing the current registry address. The application is expected to fail when changing and recover from that failure.

In this Subsection, we present the SAL and SUL components for this scenario, and the main file coordinating the services. Listing 5.11 illustrates a possible SAL definition for a service regarding sensors in a highway. The SAL definition implements the *Highway* interface, presented in Listing 5.12.

Listing 5.11: SAL - RoadServices

```
service RoadServices implements Highway {
    direction = "north"
}
```

Listing 5.12: Highway interface

```
public interface Highway {

    String trafficReport(int kms_ahead);
    void travelAssistance(int km);
    String vehiclesInOppositeWay();
    String nearestGasStation();
}
```

The SUL file to use the highway sensors service is presented on Listing 5.13, and the correspondent generated code for this component is illustrated on Appendix A.9.

Listing 5.13: SUL - TestHighway

```
import interfaces.Highway;

public class TestHighway uses volatile Highway {
    public TestHighway() { }

    public void call() {

        String output = "";
        output = Highway.trafficReport(15);
        System.out.println(output);

        Highway.travelAssistance(145);

        output = Highway.vehiclesInOppositeWay();
        System.out.println(output);

        output = Highway.nearestGasStation();
        System.out.println(output);
    }
}
```

5.3.2 Execution

Listing 5.14 illustrates the main file containing the composition of services and sequence of actions that are used in this scenario.

Listing 5.14: HighwayServicesScenario class

```
1  public class HighwayServicesScenario {
2
3      public void run() {
4
5          TestHighway road_services = new TestHighway();
6
7          road_services.call();
8
9          //changing from one road to another changes the service provider
10         //so here a new connection to a different registry is required
11         //and previous bindings are lost
12
13         road_services.call();
14     }
15
16     public static void main(String[] args) {
17         new HighwayServicesScenario().run();
18     }
19 }
```

In line 5 of Listing 5.14 a new instance of *TestHighway* class is created in order to obtain service bindings from the middleware. Line 7 shows the invocation of highway services, as illustrated on Appendix A.9 on lines 20, 29, 37, and 46. After obtaining the results of this invocation we simulate a change from one highway to another, this results in changing the registry address, current service bindings are lost. In line 13 another service invocation is realized and now every invocation fails and the replacement process is performed, lines 55-63 on Appendix A.9. A new binding is established and service invocation is performed once again, the results now are relative to another highway service.

This case study demonstrated the use of service abstraction (**uses**), and **volatile**. The latter used to replace service binding. If no match is found for that service, then the application stops executing.

5.4 Airport Services

This case study aims at evaluating the use of the services available at terminals in an airport. The operations available in this kind of services include, checking the flight list, reserve a ticket, check a flight status, and alert problem in the airport.

The goal in this scenario is similar to the previous scenario, demonstration the use of service abstraction (**uses**), implicit exception handling mechanism (**volatile**), and what happens when services are not available.

We now present the settings for this case study, also the SAL and SUL components used, and the main class file with the composition for this demonstration.

5.4.1 Scenario Settings

For this scenario the settings involve three jUDDI registries, two contain one service that represent different terminals in the same airport. During runtime after executing several operations from one service we simulate a change from one airport terminal to another by replacing the current registry address. The application is expected to fail when changing from one terminal to another, recover from that failure and when changing to another terminal the application will fail.

In this Subsection, we present the SAL and SUL components for this scenario, and the main file coordinating the services. Listing 5.15 illustrates a possible SAL definition for a service regarding sensors in a highway. The SAL definition implements the *Airport* interface, presented in Listing 5.16.

Listing 5.15: SAL - RoadServices

```
service AirportServices implements Airport {
    location = "LisbonAirport"
}
```

Listing 5.16: Airport interface

```
import java.util.List;

public interface Airport {
    List<String> checkFlightList();
    String checkForRestaurants();
    String whereIsFlight(String flight_id);
    void reportProblem(String problem);
    String carRental(String car_brand);
}
```

The SUL file to use the services in airport terminals is presented on Listing 5.17, and the correspondent generated code for this component is illustrated on Appendix A.10.

Listing 5.17: SUL - TestAirport

```
import interfaces.Airport;
import java.util.List;
import java.util.LinkedList;

public class TestAirport uses volatile Airport {

    public TestAirport() { }

    public void call() {

        List<String> res = new LinkedList<String>();
        String flight = "";

        res = Airport.checkFlightList();
        System.out.println(res);
```

```
16
17          /*user enters a flight_id*/
18          /*ex: TP301 - TAP flight number 301*/
19          flight = Airport.whereIsFlight("TP301");
20          System.out.println(flight);
21
22          /*alert to a problem in the airport*/
23          Airport.reportProblem("The elevators are not working.");
24      }
25  }
```

5.4.2 Execution

Listing 5.18 illustrates the main file containing the composition of services and sequence of actions that are used in this scenario.

Listing 5.18: AirportServicesScenario class

```
1   public class AirportServicesScenario {
2
3       public void run() {
4
5           TestAirport airport_srvs = new TestAirport();
6
7           airport_srvs.call();
8
9           //change to another terminal, service bindings are lost
10          airport_srvs.call();
11
12          //change to terminal 3, service bindings are lost
13          airport_srvs.call();
14      }
15
16      public static void main(String[] args) {
17          new AirportServicesScenario().run();
18      }
19  }
```

In Line 5, the application creates a new instance of *TestAirport* class in order to obtain service bindings from the middleware. Line 7 shows the first invocation of the service, at this stage we are located in terminal 1 of the airport. After obtaining all results we change to another terminal in the same airport, terminal 2. This results in changing the registry address and losing service bindings. Performing another request of airport services results in a failure, and since we have declared the service use as volatile, a new service binding is obtained.

Moving to another terminal in the same airport, terminal 3, and invoking airport services in line 13 results in failure again, but in this situation since the registry has no service registered no service is obtained from the middleware. The service use is declared as volatile, meaning that the middleware performs one search for services

and if no matching service is found, then the application ends. A different use for this scenario could be moving from different airports and using services on each one.

The case study demonstrated the use of service abstraction (**uses**), volatile exception handling and failure.

5.5 Final Remarks

The case studies presented here demonstrated the usage of the Sedj language. On the first scenario, we presented the use of three different services using parallel composition (|), service allocation (**all**), exception handling (**catch**), and service re-execution(**retry**). In the second scenario we focused on losing bindings changing from one location to another and recover from them with **volatile** service definition. Finally, in the third scenario we shown recovery from losing bindings, also with volatile service definition, and what happens if no service is available. The evaluation of these results have been the expected.

One benefit that is most notorious from using our framework is to compare the input SUL component with the correspondent generated code. We can agree that the equivalent generated code is more complex and bigger in regard the code presented in SUL components using the new programming abstractions.

Sedj provides a simpler and easier way for service composition concerning SAL and SUL components. The framework is supported by a middleware that handles aspects of the dynamic environment.

6

Conclusions

The objective of this dissertation was to study and develop an initial implementation of the SeDeUse model. It features new programming abstractions added to Java language specification and is sustained by a middleware layer to cope with the characteristics of using service-oriented computing in dynamic environments. The model is divided in two phases, the compiler and middleware modules. The former is responsible for separating the functional from the non-functional requirements on a two-layer approach, SAL and SUL. The latter represents a software layer that provides an API for interaction with the generated code from the compiler phase, and handles the dynamic nature of the execution environment.

To achieve this goal we first designed a model that can be easily integrated in high-level programming languages such as, Java. In this model we also studied from scratch a support mechanism for dynamic environments. Secondly, the model is instantiated in a general use language in order to serve as a tool to assess the language expressiveness.

This thesis provided three main contributions. First the concrete instantiation of the SeDeUse model in the Java language, which we named Sedj. It expresses the concrete syntax for the new constructs in Java. Secondly, development of a prototype that includes, translation of SAL and SUL components to Java language, a middleware layer, also in Java, for support to the application in dynamic environments, and an interface mapping mechanism between services.

Finally, we tested our functional prototype in order to assess the language expressiveness. Three case studies were performed, and from our experiments we have con-

cluded that our implementation of SeDeUse presents good results. There is a separation between service use and definition, and the middleware handles the dynamic nature of the execution environment.

6.1 Future Work

During the development of our work, some aspects that can be improved in the future were identified. This section highlights the most important.

Semantic web: in order to achieve a full automation of the Web Services life-cycle.

Several services in one SUL component: the possibility of declaring different services in SUL components.

Interface mapping: other type of interface mapping support can be added to the framework.

Dynamic properties: the possibility of modifying service properties at runtime.

Integration: with another project being developed by another student in order to achieve code mobility.

The use of Semantic Web has a key concept: the **ontology**. It is a formal representation of the knowledge within a certain domain. An ontology is a data structure hierarchized with information regarding entities and the relations between them in a certain domain. With an ontology for Web Services, the whole process of searching, selection, and invoking services can be autonomous.

At the moment, only one type of service can be defined in each SUL component. The possibility of declaring more than one service diminishes the number of SUL components, and increases the language expressiveness. Listing 6.1 illustrates an example of a possible SUL component with several services.

The mapping mechanism can be a replaceable component in the framework. In Chapter 2 in subsection 2.2.2 regarding the frameworks that use aspects, the Web Service Management Layer (WSML) [40] uses an abstract service interface that hides syntactic differences between semantically equivalent services. In this case, the abstract service interface performs the required transformations in order to match the requests to the equivalent services. Other types of mapping can be added to our framework, for instance, a similar mechanism to WSML or service methods mapping.

In Chapter 3 on subsection 3.1.1 regarding the software mobility topic, we present the possibility of specifying a code mobility mechanism on service properties (SAL).

Another thesis regarding this topic is being developed and is a feature to be later integrated in Sedj.

Listing 6.1: SUL example several services

```
public class DocumentManipulationScenario uses volatile Translation, Converter, Printer {

    public DocumentManipulationScenario() {
    }

    public byte[] callTranslation(byte[] document, String input_lang, String output_lang) {
      byte[] result = null;
        result = Translation.translate(document, input_lang, output_lang);
        return result;
    }

    public byte[] callConverter(byte[] document, String extension) {
        byte[] res = null;
        res = Converter.convert(doc, extension);
        return res;
    }

    public void callPrinter(byte[] doc) {
        this.doc = doc;

        //parallel composition
        {Printer.print(doc);} | {Printer.print(doc);}
    }
}
```

Appendix

A.1 ServiceAppProperties class

```
1  package services;
2  import java.io.Serializable;
3  import java.util.HashMap;
4  import java.util.List;
5
6  public class ServiceAppProperties implements Serializable {
7
8      private static final long serialVersionUID = 3L;
9      private String interface_name;
10     private List<String> service_stubs;
11     private String classname;
12     private int nos;
13     private HashMap<String, List<String>> properties;
14     private boolean isVolatile;
15
16     public ServiceAppProperties(String interface_name,
17             List<String> service_stubs, String classname, int nos,
18             HashMap<String, List<String>> properties, boolean isVolatile) {
19         this.interface_name = interface_name;
20         this.service_stubs = service_stubs;
21         this.classname = classname;
22         this.nos = nos;
23         this.properties = properties;
24         this.isVolatile = isVolatile;
25     }
26
27     public String getInterfaceName() {
28         return this.interface_name;
29     }
30
```

```
31      public List<String> getServiceStubs() {
32          return this.service_stubs;
33      }
34
35      public String getClassname() {
36          return this.classname;
37      }
38
39      public int getNos() {
40          return this.nos;
41      }
42
43      public HashMap<String, List<String>> getProperties() {
44          return this.properties;
45      }
46
47      public void addStubs(List<String> stubs) {
48          this.service_stubs.addAll(stubs);
49      }
50
51      public void replaceServiceStub(int index, String stub) {
52          this.service_stubs.set(index, stub);
53      }
54
55      public boolean isVolatile(){
56          return this.isVolatile;
57      }
58  }
```

A.2 ServiceLocation class

```
1  package middleware;
2
3  import java.io.Serializable;
4
5  public class ServiceLocation implements Serializable {
6
7      private static final long serialVersionUID = 1L;
8      private String wsdl_location;
9      private String prop_location;
10
11     public ServiceLocation(String wsdl, String prop) {
12         this.wsdl_location = wsdl;
13         this.prop_location = prop;
14     }
15
16     public String getServiceLocation() {
17         return this.wsdl_location;
18     }
19
20     public String getPropertiesLocation() {
21         return this.prop_location;
22     }
23 }
```

A.3 Service class

```
1  package services;
2
3  import java.io.Serializable;
4  import java.util.HashMap;
5  import java.util.List;
6
7  public class Service implements Serializable {
8
9      private static final long serialVersionUID = 1L;
10     private String name;
11     private String service_extends;
12     private String service_implements;
13     private HashMap<String, List<String>> properties;
14
15
16     public Service(String name, String interface_name, String extends_name, HashMap<String,
           List<String>> properties) {
17         this.name = name;
18         this.service_implements = interface_name;
19         this.service_extends = extends_name;
20         this.properties = properties;
21     }
22
23     public String getName() {
24         return this.name;
25     }
26
27     public String getInterface() {
28         return this.service_implements;
29     }
30
31     public String getExtends() {
32         return this.service_extends;
33     }
34
35     public HashMap<String, List<String>> getProperties() {
36         return this.properties;
37     }
38 }
```

A.4 SedjInfo class

```
1  package compiler;
2
3  import java.io.Serializable;
4
5  public class SedjInfo implements Serializable {
6
7      private static final long serialVersionUID = 1L;
8
9      private String interface_name;
10     private int nos;
11     private String classname;
12
13     public SedjInfo(String interface_name, int nos, String classname) {
14         this.interface_name = interface_name;
15         this.nos = nos;
16         this.classname = classname;
17     }
18
19     public String getInterfaceName() {
20         return this.interface_name;
21     }
22
23     public int getNumberOfServices() {
24         return this.nos;
25     }
26
27     public String getClassname() {
28         return this.classname;
29     }
30 }
```

A.5 Cache class

```java
package middleware.cache;

import java.util.List;

public class CacheImpl implements Cache {

    private String interface_name;
    private List<String> service_stubs;

    public CacheImpl(String interface_name) {
        this.interface_name = interface_name;
    }

    public void addServiceStub(String name) {
        service_stubs.add(name);
    }

    public List<String> getServiceStubs() {
        return this.service_stubs;
    }

    public boolean removeServiceStub(String stub_name) {
        return service_stubs.remove(stub_name);
    }
}
```

A.6 TestTranslator generated code

```
import interfaces.Translation;

public class TestTranslator {

    public boolean fail = false;
    private java.util.List<Translation> srvs;
    private java.util.List<Integer> failed_srvs;
    private middleware.application.ApplicationManager app_manager;
    public byte[] result = null;

    public TestTranslator() {
        super();
        failed_srvs = new java.util.LinkedList<Integer>();
        app_manager = new middleware.application.ApplicationManagerImpl("TestTranslator");
        srvs = app_manager.getServiceBindings();
    }

    private byte[] _call(int attempts, int time, byte[] document, String input_lang, String
        output_lang) {
        try {
            result = srvs.get(0).translate(document, input_lang, output_lang);
        } catch (Exception e) {
            fail = true;
            if (!failed_srvs.contains(0)) {
                failed_srvs.add(0);
            }
        }
        if (fail && attempts > 0) {
            fail = false;
            for (int i = 0; i < failed_srvs.size(); i++) {
                Object obj = app_manager.replaceServiceBinding(failed_srvs.get(i));
                srvs.set(failed_srvs.get(i), (Translation) obj);
            }
            failed_srvs = new java.util.LinkedList<Integer>();
            try {
                Thread.sleep(time);
            } catch (InterruptedException e) {
            }
            _call(attempts - 1, time, document, input_lang, output_lang);
        }
        return result;
    }

    public byte[] call(byte[] document, String input_lang, String output_lang) {
        return _call(1, 3000, document, input_lang, output_lang);
    }
}
```

A.7 TestConverter generated code

```
import interfaces.Converter;

public class TestConverter {

    public boolean fail = false;
    private java.util.List<Converter> srvs;
    private java.util.List<Integer> failed_srvs;
    private middleware.application.ApplicationManager app_manager;
    public byte[] doc;
    public byte[] res;
    public String extension;

    public TestConverter() {
        super();
        failed_srvs = new java.util.LinkedList<Integer>();
        app_manager = new middleware.application.ApplicationManagerImpl("TestConverter");
        srvs = app_manager.getServiceBindings();
    }

    private byte[] _call(byte[] doc, String extension) {
        try {
            res = srvs.get(0).convert(doc, extension);
        } catch (Exception e) {
            fail = true;
            if (!failed_srvs.contains(0)) {
                failed_srvs.add(0);
            }
        }
        if (fail) {
            fail = false;
            for (int i = 0; i < failed_srvs.size(); i++) {
                Object obj = app_manager.replaceServiceBinding(failed_srvs.get(i));
                srvs.set(failed_srvs.get(i), (Converter) obj);
            }
            failed_srvs = new java.util.LinkedList<Integer>();
            _call(byte[] doc, String extension);
        }
        return res;
    }

    public void call(byte[] doc, String extension) {
        return _call(doc, extension)
    }
}
```

A.8 TestPrinter generated code

```
1  import interfaces.Printer;
2
3  public class TestPrinter {
4
5      public boolean fail = false;
6      private java.util.List<Printer> srvs;
7      private java.util.List<Integer> failed_srvs;
8      private middleware.application.ApplicationManager app_manager;
9      public boolean passed = false;
10     private java.util.List<Thread> threads;
11     public byte[] doc;
12
13     public TestPrinter() {
14         super();
15         failed_srvs = new java.util.LinkedList<Integer>();
16         app_manager = new middleware.application.ApplicationManagerImpl("TestPrinter");
17         threads = new java.util.LinkedList<Thread>();
18         srvs = app_manager.getServiceBindings();
19     }
20
21     private void _call(int attempts, int time, byte[] doc) {
22         this.doc = doc;
23         if (!passed)
24             threadGen();
25         for (int i = 0; i < threads.size(); i++) {
26             threads.get(i).start();
27         }
28         for (int i = 0; i < threads.size(); i++) {
29             try {
30                 threads.get(i).join();
31             } catch (InterruptedException e) {
32             }
33         }
34         passed = false;
35         threads = new java.util.LinkedList<Thread>();
36
37         if (fail && attempts > 0) {
38             fail = false;
39             for (int i = 0; i < failed_srvs.size(); i++) {
40                 Object obj = app_manager.replaceServiceBinding(failed_srvs.get(i));
41                 srvs.set(failed_srvs.get(i), (Printer) obj);
42             }
43             failed_srvs = new java.util.LinkedList<Integer>();
44             try {
45                 Thread.sleep(time);
46             } catch (InterruptedException e) {
47             }
48             _call(attempts - 1, time, doc);
49         }
50     }
51
52     private void threadGen() {
53         passed = true;
54         Thread t0 = new Thread(new Runnable() {
```

```
55              public void run() {
56                  try {
57                      srvs.get(0).print(doc);
58                  } catch (Exception e) {
59                      fail = true;
60                      if (!failed_srvs.contains(0)) {
61                          failed_srvs.add(0);
62                      }
63                  }
64              }
65          });
66          threads.add(t0);
67          Thread t1 = new Thread(new Runnable() {
68              public void run() {
69                  try {
70                      srvs.get(1).print(doc);
71                  } catch (Exception e) {
72                      fail = true;
73                      if (!failed_srvs.contains(1)) {
74                          failed_srvs.add(1);
75                      }
76                  }
77              }
78          });
79          threads.add(t1);
80      }
81
82      public void call(byte[] doc) {
83          _call(1, 4000, doc);
84      }
85  }
```

A.9 TestHighway generated code

```
1  import interfaces.Highway;
2
3  public class TestHighway {
4
5      public boolean fail = false;
6      private java.util.List<Highway> srvs;
7      private java.util.List<Integer> failed_srvs;
8      private middleware.application.ApplicationManager app_manager;
9
10     public TestHighway() {
11         super();
12         failed_srvs = new java.util.LinkedList<Integer>();
13         app_manager = new middleware.application.ApplicationManagerImpl("TestHighway");
14         srvs = app_manager.getServiceBindings();
15     }
16
17     private void _call() {
18         String output = "";
19         try {
20             output = srvs.get(0).trafficReport(15);
21         } catch (Exception e) {
22             fail = true;
23             if (!failed_srvs.contains(0)) {
24                 failed_srvs.add(0);
25             }
26         }
27         System.out.println(output);
28         try {
29             srvs.get(0).travelAssistance(145);
30         } catch (Exception e) {
31             fail = true;
32             if (!failed_srvs.contains(0)) {
33                 failed_srvs.add(0);
34             }
35         }
36         try {
37             output = srvs.get(0).vehiclesInOppositeWay();
38         } catch (Exception e) {
39             fail = true;
40             if (!failed_srvs.contains(0)) {
41                 failed_srvs.add(0);
42             }
43         }
44         System.out.println(output);
45         try {
46             output = srvs.get(0).nearestGasStation();
47         } catch (Exception e) {
48             fail = true;
49             if (!failed_srvs.contains(0)) {
50                 failed_srvs.add(0);
51             }
52         }
53         System.out.println(output);
54
```

```
55          if (fail) {
56              fail = false;
57              for (int i = 0; i < failed_srvs.size(); i++) {
58                  Object obj = app_manager.replaceServiceBinding(failed_srvs.get(i));
59                  srvs.set(failed_srvs.get(i), (Highway) obj);
60              }
61              failed_srvs = new java.util.LinkedList<Integer>();
62              _call();
63          }
64      }
65
66      public void call() {
67          _call();
68      }
69  }
```

A.10 TestAirport generated code

```
1  import interfaces.Airport;
2  import java.util.List;
3  import java.util.LinkedList;
4
5  public class TestAirport {
6
7      public boolean fail = false;
8      private List<Airport> srvs;
9      private List<Integer> failed_srvs;
10     private middleware.application.ApplicationManager app_manager;
11
12     public TestAirport() {
13         super();
14         failed_srvs = new LinkedList<Integer>();
15         app_manager = new middleware.application.ApplicationManagerImpl("TestAirport");
16         srvs = app_manager.getServiceBindings();
17     }
18
19     private void _call() {
20         List<String> res = new LinkedList<String>();
21         String flight = "";
22         try {
23             res = srvs.get(0).checkFlightList();
24         } catch (Exception e) {
25             fail = true;
26             if (!failed_srvs.contains(0)) {
27                 failed_srvs.add(0);
28             }
29         }
30         System.out.println(res);
31         try {
32             flight = srvs.get(0).whereIsFlight("TP301");
33         } catch (Exception e) {
34             fail = true;
35             if (!failed_srvs.contains(0)) {
36                 failed_srvs.add(0);
37             }
38         }
39         System.out.println(flight);
40         try {
41             srvs.get(0).alertProblem("The elevators are not working.");
42         } catch (Exception e) {
43             fail = true;
44             if (!failed_srvs.contains(0)) {
45                 failed_srvs.add(0);
46             }
47         }
48
49         if (fail) {
50             fail = false;
51             for (int i = 0; i < failed_srvs.size(); i++) {
52                 Object obj = app_manager.replaceServiceBinding(failed_srvs.get(i));
53                 srvs.set(failed_srvs.get(i), (Airport) obj);
54             }
```

```
55              failed_srvs = new LinkedList<Integer>();
56              _call();
57          }
58      }
59
60      public void call() {
61          _call();
62      }
63  }
```

Bibliography

[1] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services: Concepts, Architecture and Applications*. Springer Verlag, 2004.

[2] Guruduth Banavar and Abraham Bernstein. Software infrastructure and design challenges for ubiquitous computing applications. *Commun. ACM*, 45(12):92–96, 2002.

[3] Michael. Beisiegel and al. Service Component Architecture - Building Systems using a Service Oriented Architecture. *A Joint Whitepaper by BEA, IBM, Interface21, IONA, SAP, Siebel, Sybase*, Nov 2005.

[4] Boualem Benatallah, Marlon Dumas, and Quan Z. Sheng. Facilitating the Rapid Development and Scalable Orchestration of Composite Web Services. In *Distributed and Parallel Databases*, 2005.

[5] Joseph Bih. Service oriented architecture (SOA) a new paradigm to implement dynamic e-business solutions. *Ubiquity*, 7(30):1–1, 2006.

[6] Michael Blow, Yaron Goland, Matthias Kloppmann, Frank Leymann, Gerhard Pfau, Dieter Roller, and Michael Rowley. BPELJ: BPEL for Java. BEA and IBM, March 2004.

[7] Anis Charfi and Mira Mezini. Aspect-Oriented Web Service Composition with AO4BPEL. In *European Conference on Web Services*, pages 168–182, 2004.

[8] COM/DCOM: `http://www.microsoft.com/com/default.mspx`.

[9] CORBA: `http://www.omg.org`.

[10] Massimiliano Di Penta, Raffaele Esposito, Maria Luisa Villani, Roberto Codato, Massimiliano Colombo, and Elisabetta Di Nitto. WS Binder: a framework to

enable dynamic binding of composite web services. In *SOSE '06: Proceedings of the 2006 international workshop on Service-oriented software engineering*, pages 74–80, New York, NY, USA, 2006. ACM.

[11] Elilabs: `http://www.elilabs.com/rj/dreams/node18.html`.

[12] Abdelkarim Erradi and Piyush Maheshwari. Dynamic Binding Framework for Adaptive Web Services. In *ICIW '08: Proceedings of the 2008 Third International Conference on Internet and Web Applications and Services*, pages 162–167, Washington, DC, USA, 2008. IEEE Computer Society.

[13] Oneyka Ezenwoye and S. Masoud Sadjadi. Composing aggregate web services in BPEL. In *ACM-SE 44: Proceedings of the 44th annual Southeast regional conference*, pages 458–463, New York, NY, USA, 2006. ACM.

[14] Onyeka Ezenwoye and S. Masoud Sadjadi. TRAP/BPEL: A framework for dynamic adaptation of composite services. 2008.

[15] Gregory Hackmann, Christopher Gill, and Gruia-Catalin Roman. Extending BPEL for Interoperable Pervasive Computing. *Pervasive Services, IEEE International Conference on*, pages 204–213, July 2007.

[16] HTTP: `http://www.w3.org/Protocols/rfc2616/rfc2616.html`.

[17] HTTP tunneling:
`http://java.sun.com/developer/onlineTraining/rmi/RMI.html`.

[18] Scott E. Hudson, Frank Flannery, C. Scott Ananian, Dan Wang, and Andrew Appel. CUP LALR parser generator for Java. http://www.cs.princeton.edu/ appel/modern/java/CUP/, 1996.

[19] Java Beans: `http://java.sun.com/javase/technologies/desktop/javabeans/index.jsp`.

[20] Matjaz B. Juric. *Business Process Execution Language for Web Services BPEL and BPEL4WS 2nd Edition*. Packt Publishing, 2006.

[21] Viswanathan Kodaganallur. Incorporating language processing into java applications: A javacc tutorial. *IEEE Software*, 21:70–77, 2004.

[22] David Martin, John Domingue, Michael L. Brodie, and Frank Leymann. Semantic Web Services, Part 1. *IEEE Intelligent Systems*, 22(5):12–17, 2007.

[23] James McGovern, Sameer Tyagi, and Sunil Mathew Michael Stevens. *Java Web Services Architecture*. Elsevier Science and Technology, 2003.

[24] Stefano Modafferi, Enrico Mussi, and Barbara Pernici. SH-BPEL: a self-healing plug-in for Ws-BPEL engines. In *MW4SOC '06: Proceedings of the 1st workshop on Middleware for Service Oriented Computing (MW4SOC 2006)*, pages 48–53, New York, NY, USA, 2006.

[25] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for java. In *In 12th International Conference on Compiler Construction*, pages 138–152. Springer-Verlag, 2003.

[26] Bart Orriëns, Jian Yang, and Mike P. Papazoglou. ServiceCom: A Tool for Service Composition Reuse and Specialization. In *WISE '03: Proceedings of the Fourth International Conference on Web Information Systems Engineering*, page 355, Washington, DC, USA, 2003. IEEE Computer Society.

[27] OWL-S: `http://www.w3.org/Submission/OWL-S/`, 2004.

[28] Michael P. Papazoglou, Paolo Traverso, Schahram Dustdar, Frank Leymann, and Bernd J. Krämer. 05462 Service-Oriented Computing: A Research Roadmap. In Francisco Cubera, Bernd J. Krämer, and Michael P. Papazoglou, editors, *Service Oriented Computing (SOC)*, number 05462 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[29] Hervé Paulino and Carlos Tavares. SeDeUse: A Model for Service-oriented Computing in Dynamic Environments . In *Second International Conference on MOBILe Wireless MiddleWARE, Operating Systems, and Applications (Mobilware 2009)*, number 7 in Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, pages 157–170. Springer-Verlag, 04 2009.

[30] Cesare Pautasso. Rapid composition of web services with jopera for eclipse. In *Submission for the Research-Industry Exchange at EclipseCon 2005*, 2005.

[31] Cesare Pautasso and Gustavo Alonso. "The JOpera visual composition language". *Journal of Visual Languages and Computing*, 16(1-2):119 – 152, 2005. 2003 IEEE Symposium on Human Centric Computing Languages and Environments.

[32] Cesare Pautasso, Thomas Heinis, and Gustavo Alonso. JOpera: Autonomic Service Orchestration. *IEEE Data Engineering Bulletin*, 29, September 2006 2006.

[33] Polyglot5:
http://www.cs.ucla.edu/~milanst/projects/polyglot5.

[34] Tian Qiu, Lei Li, and Pin Lin. Web Service Discovery with UDDI Based on Semantic Similarity of Service Properties. In *SKG '07: Proceedings of the Third International Conference on Semantics, Knowledge and Grid*, pages 454–457, Washington, DC, USA, 2007. IEEE Computer Society.

[35] RMI: http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp.

[36] SOAP: http://www.w3.org/TR/soap/, 2007.

[37] Patrik Spiess, Stamatis Karnouskos, Dominique Guinard, Domnic Savio, Oliver Baecker, Luciana Moreira Sá de Souza, and Vlad Trifa. SOA-Based Integration of the Internet of Things in Enterprise Services. In *IEEE International Conference on Web Services, ICWS 2009 , Los Angeles, CA, USA*, pages 968–975, July 6–10, 2009.

[38] TModel:
http://www.codeproject.com/KB/XML/understandingTModels.aspx.

[39] UDDI: http://uddi.xml.org/.

[40] Bart Verheecke, María A. Cibrán, and Viviane Jonckers. AOP for Dynamic Configuration and Management of Web Services. pages 137–151.

[41] Guijun Wang and Casey K. Fung. Architecture Paradigms and Their Influences and Impacts on Component-Based Software Systems. In *HICSS '04: Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 9*, page 90272.1, Washington, DC, USA, 2004. IEEE Computer Society.

[42] Web Services Reliable Messaging:
http://www.ibm.com/developerworks/library/specification/ws-rm/.

[43] Web Services Security:
http://www.ibm.com/developerworks/library/specification/ws-secure/.

[44] Web Services Transaction:
http://www.ibm.com/developerworks/library/specification/ws-tx/.

[45] WS Addressing:
http://www.w3.org/Submission/ws-addressing/, 2004.

[46] WS-BPEL: http://www.oasis-open.org/committees/download.php/14616/wsbpel-specification-draft.htm.

[47] WS-CDL: http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217/, 2004.

[48] WSDL: http://www.w3.org/TR/wsdl20/, 2007.

[49] WS Management Standard: http://www.dmtf.org/standards/wsman/.

[50] XML: http://www.w3.org/TR/REC-xml, 2008.