Académie de Montpellier
Université Montpellier II
Sciences et **T**echniques du **L**anguedoc

Thèse

pour obtenir le grade de docteur

Spécialité	:	**Informatique**
Formation Doctorale	:	**Informatique**
École Doctorale	:	**Information, Structures, Systèmes**

présentée et soutenue publiquement par

Zeina Azmeh

A Web Service Selection Framework for an Assisted SOA

Soutenue le septembte 2011, devant le jury composé de :

Rapporteurs
Yamine Aït-Ameur (Prof.) ..... LISI, École Nationale Supérieure de Mécanique et d'Aérotechnique
Philippe Lalanda (Prof.) ..... LIG, Université Joseph Fourier

Examinateurs
Stefano A. Cerri (Prof.) ..... LIRMM, Université de Montpellier II
Michel Dao (Ingénieur Expert) ..... Orange Labs
Laurence Duchien (Prof.) ..... LIFL, Université de Lille 1

Directrice de thèse
Marianne Huchard (Prof.) ..... LIRMM, Université de Montpellier II

Co-Directeurs de thèse
Chouki Tibermacine (MCF) ..... LIRMM, Université de Montpellier II
Christelle Urtado (MCF) ..... LGI2P, École des Mines d'Alès

Acknowledgments

Contents

Chapter 1

Introduction

Contents

1 Context

Web services move us beyond information exchange, towards the interoperation between different applications on a network [1]. The interoperability of software applications enables us of defining new software systems by the integration of existing applications, which are accessible through well-defined services. The idea of creating new software systems by loosely coupling services over a network is the heart of the Service-Oriented Architectures (SOA) paradigm [2, 3, 4]. Although SOA is not necessarily based on Web services, they represent an important realization of these architectures. Web services are also important because they are based upon open internet standards for description and invocation.

Dealing with Web services recognizes three fundamental actions, each of which is related to a standard: *Web service description*, the action using which, a service can be represented(exposed and advertised) to the external world. A functionality of a service is represented to the external world by the means of an abstract interface described using the standard Web Service Description Language (WSDL) [5]; *Web service discovery*, which is associated to registries and repositories, where one can search for a needed service. A registry must provide a fairly precise search capability to render the discovery action more efficient and less time-consuming. Initially, Web service registries were supposed to be built upon a standard called UDDI [6], but discovery mechanisms are not limited to this standard, as we shall in more details see later on; *Web service invocation*, which describes a client-server communication and is embodied in the messages exchanged between a service provider and a service consumer. These messages are described by a standard called SOAP [7], and are transmitted over an HTTP connection.

Performing the three previous actions is done by three actors, a *provider*, a *registry*, and a *consumer*, as illustrated in Figure 1.1. This figure represents what we call the SOA triangle [3]. It illustrates the interaction between the three actors, as well as the used standards.

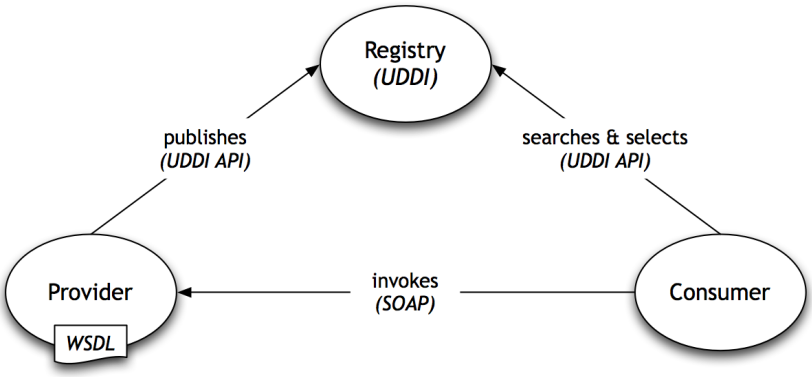


Figure 1.1: The SOA triangle.

A service provider publishes his service at a service UDDI registry using a publishing interface. The public interfaces and binding information of the registered services are clearly defined in the WSDL standard language. A registry organizes the published services and provides a query interface that enables a service consumer to search for a needed service, and obtain its provider's location information. A service consumer then interacts with a service provider through the SOAP protocol.

In our days, Web service discovery is not limited to UDDI-based registries. Service discovery can be carried out through either Web service portals (service search engines) or through websites of certain service providers. The previous SOA triangle can be rather represented as in Figure 1.2.

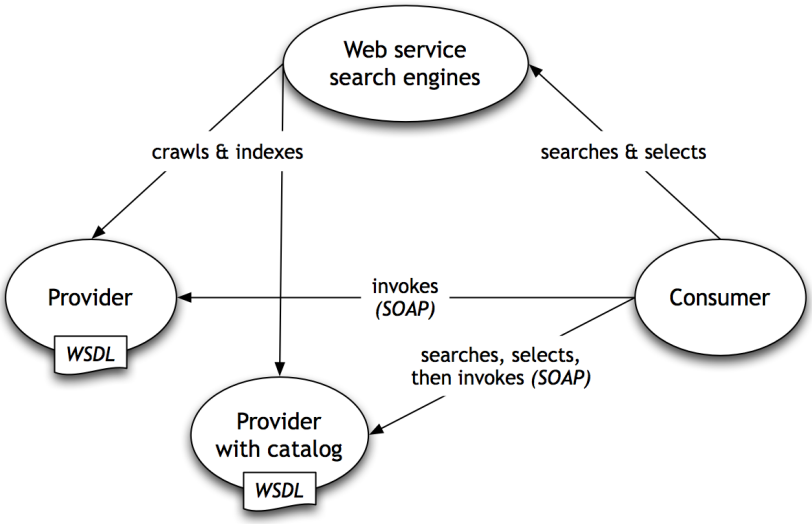


Figure 1.2: The SOA triangle nowadays.

Web portals are basically search engines that use focused crawlers as well as manual reg-

istration to gather WSDL interfaces and index them. Web services can be used following the cycle in Figure 1.3, which involves several actions, as follows:

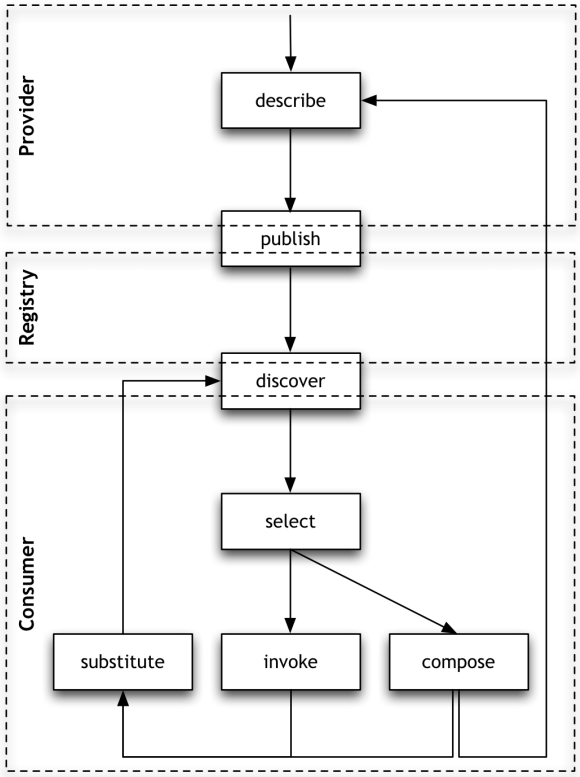


Figure 1.3: The Web service utilization cycle and its activities.

A Web service utilization cycle starts by *describing* a service before exposing it to the external world. A service is described using a standard language called WSDL. Exposing a service to the external world is done by *publishing* it to a public registry built using a standard called UDDI. Using such a public registry, a service can be *discovered* among many others, then be selected if it meets the required functionality. Once a service is *selected*, it can be either *invoked* inside an application using a standard called SOAP, or can be *composed* with other services to form a composite service, which itself can be considered as a new service that can follow the utilization cycle. Sometimes, a service that is selected to be used inside an application or a composition, may become no longer functional for some reason. In such a case, we can *substitute* this service by another discovered and selected one.

In the next section, we discuss the issues and difficulties surrounding the consumer's part of this Web service utilization cycle.

2 Problem Statement - Web Service Selection

In this thesis, we deal with the consumer related part for Web service utilization, which is illustrated previously in Figure 1.3.

The main question to answer is:
How can a consumer select a suitable service to use, according to some desired requirements?

We define the service selection problem as a four-levels problem, as illustrated in Figure 1.4.

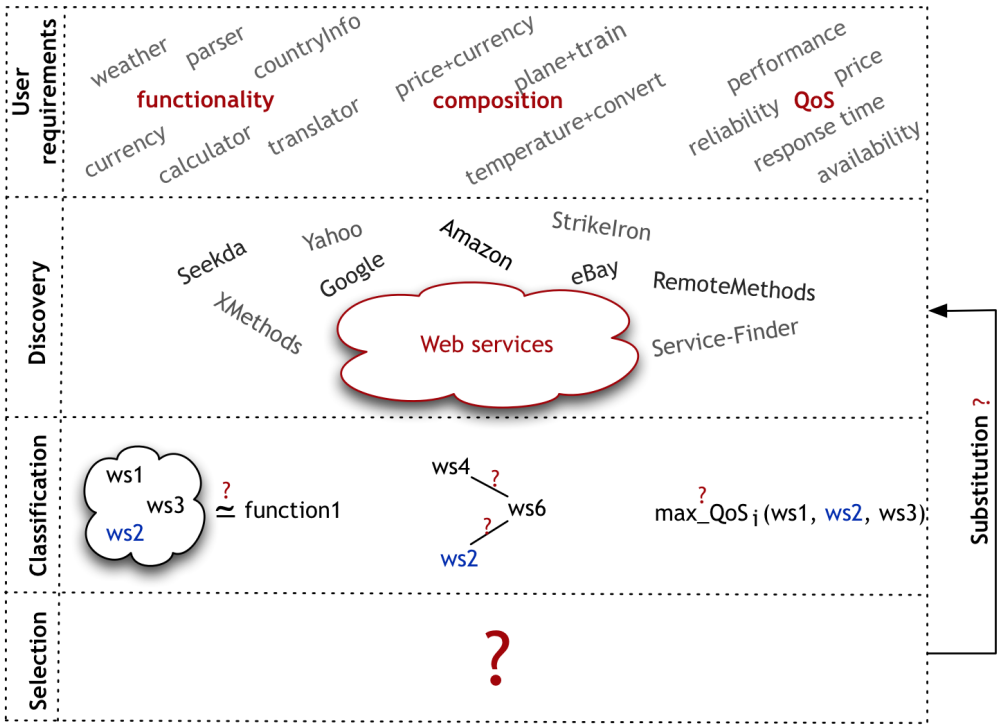


Figure 1.4: Overview of the problematic of service selection.

2.1 Describing User Requirements

Current discovery mechanisms limit the user expression capability to only keywords. This disables the user from specifying his needs for a certain functionality (one or more operations), for an accepted level of QoS per each service, and for composable services. Thus, there is a lack of a requirement description language that enables the user of specifying all of the previous requirements.

2.2 Web Service Discovery

There is a large number of Web services nowadays that is increasing everyday (Seekda Web service search engine [8] indexes more than 28,000 services). They are described using WSDL language, inside which, the description is restricted to syntactic information. The only available semantic information is the name of each parameter, which may not always be clear [9].

Current discovery mechanisms are embodied in Web service search engines, especially after the deficiency of UDDI registries: "UDDI did not achieve its goal of becoming the registry for

all Web Services metadata and did not become useful in a majority of Web Services interactions over the Web" [10]. The Web service search engines index services by the keywords found in their WSDL interfaces. This is not efficient nor adequate, because of the insufficient amount of words that can be extracted from the WSDL interfaces, especially when they are not documented. Such search engines return a large list of services that are not all related to the used keywords, nor to the searched functionality, and who might have invalid WSDL syntax. Thus, this raises a need for validity and functional filtration for verifying each returned service.

2.3 Web Service Classification

Before selecting a service, there is a need for services organization and browsing techniques, according to which, services can be ranked. This is an effort- and time- consuming task for the user to do by hand, especially when having a large number of returned services that are not necessarily documented. There is also a need for tools to determine whether two services provide similar functionality, and to decide which service offers a better compromise of QoS levels. The last important thing to consider is deciding which services can be composed with others.

2.4 Web Service Selection

The selection of a proper Web service to use is not straightforward, according to the previous problems of expressing user requirements, service discovery and classification techniques. It even becomes a real challenge when building a service composition.

Web services have various characteristics of QoS due to several factors, like the dynamic nature of the internet and the various service providers (individuals and professionals). Services are remotely accessed, and sometimes they can be provided by non-professional individuals for free. Therefore, their continuous execution might not be guaranteed nor their QoS levels.

If a user succeeds in discovering, selecting, and composing all of the services he needs, his service composition might be threatened by the fail of one or more services at any moment. In such a case, there is a need for a mechanism of selecting service substitutes (backups) to recover the missing functionality and maintain the composition's continuity.

Thus, there is a need to determine which services to select, based on identifying which ones provide the needed functionality, offer the best compromise of QoS levels, and are composable with others.

3 Contributions

We propose a framework that assists users during design-time, in the selection of a needed Web service either for individual invocation, or for composition. In addition to considering possible service substitutions to maintain a continuous functionality.

The framework takes into consideration several criteria for performing the selection. These criteria represent the user requirements for functionality, QoS, and composition. Thus, the

framework tackles the consumer related part of Web service utilization, including Web service discovery, selection for independent or composition utilization, and substitution. Therefore, each layer of our framework corresponds to a level of the described problem in the previous section, and they are as follows:

- **User requirements layer:** in which, we propose to extend the WSDL standard to define an abstract WSDL description of the needed services. In such description file, we can specify several needed services by their functionality, along with their accepted QoS levels. We also propose to use the BPEL standard to define an abstract process, in which, a user can use the services that he described in the abstract WSDL file.
- **Discovery layer:** in which, we propose to have several components for the retrieval of the needed services. We propose to have an analyzer for user requirements specified in the abstract WSDL and abstract BPEL files; a service retriever that searches for the needed services using a Web service resource; and a WSDL parser that extracts the information inside each retrieved WSDL file, and checks the service's validity.
- **Classification layer:** in which, we adopt two classification techniques based on concept lattices. The first technique is called Formal Concept Analysis (FCA) and the second is Relational Concept Analysis (RCA). We classify services using these techniques into lattices to facilitate the selection.
- **Selection layer:** in which, we propose methods for lattice navigation, in order to identify and select the services that best match user requirements, together with their potential substitutes (backups).

4 Thesis Outline

This thesis contains a background and state of the art study, a description of our proposed framework with several use cases, and an overview of the obtained experimental results. The following chapters are organized as follows:

- **Chapter** 2 gives a background about Web services along with their related standards.
- **Chapter** 3 studies the related work, lists the different used technologies, and puts the light on the current issues.
- **Chapter** 4 gives a background about the Formal and Relational Concept Analysis (FCA, RCA), together with their formal definitions illustrated by examples. It also proposes an original algorithm for querying and navigating a concept lattice family [A].
- **Chapter** 5 presents an overview about our proposed framework that handles the problem of Web service selection, as well as a description of its main layers and components.
- **Chapter** 6 describes Web service selection by tags [B], the first use case of our framework. In this use case, we show the utility of our framework for the selection of one or more independent services, along with their possible substitutes. We use formal concept analysis (FCA) to classify Web services into concept lattices according to their automatically

extracted tags (the significant keywords appearing in their documentations) [C]. A service lattice reveals the invisible relations between the services. It is considered as a browsing mechanism that facilitates the selection of a needed service, and the identification of its candidate backups. This enables a continued functionality of a Web service, which becomes indispensable, especially when a service represents a part of a composite application.

- **Chapter** 7 describes Web service selection by functionality [D], the second use case of our framework. In this use case, we enable browsing Web services by their functionality, in order to facilitate the selection of a service offering needed operations, and the identification of its possible backups. We accomplish this by constructing Web service lattices using many-valued contexts of similarity values calculated for each pair of operations. This enables us of extracting groups of mutually similar operations. Each one of these groups represents a functionality, according to which, a new service lattice can be generated. The generated service lattices provide us with navigation capabilities.

- **Chapter** 8 describes Web service selection according to multi- user requirements [E], the third and final use case of our framework. In this use case, we aim at providing a facility for building business processes transparently, according to user requirements. This means that a user can model his business process in an abstract way, without being aware of the concrete services existing on the Web. This is realized by considering three levels of user requirements: the needed functionality, the accepted QoS levels, and the composition. Then, identifying the services that satisfy these requirements. We specialize our framework in order to facilitate Web service selection according to user requirements. We use the Relational Concept Analysis (RCA) to characterise the services by their QoS levels and to express the composition relations between them. The generated lattices help in identifying the services that match the specified requirements, with the help of RCA relational queries.

- **Chapter** 9 presents the set of experiments that were conducted using real Web services to validate our framework. We present each experiment on two parts: a methodology part, where we explain the steps that we followed for conducting the experiment; and a validation part, where we show and discuss the obtained results.

- **Chapter** 10 concludes with a summary about the proposed framework with its use cases. It also draws the future trends and perspectives.

Related Publications

[*A*] Zeina Azmeh, Mohamed Hacene Rouane, Marianne Huchard, Amedeo Napoli and Petko Valtchev: Lessons Learned in Querying Relational Concept Lattices to Locate Concepts of Interest. Submitted to the eighth International Conference on Concept Lattices and their Applications (CLA'11), Nancy, France, October 2011. (Notification on August 22, 2011).

[*B*] Zeina Azmeh, Marianne Huchard, Chouki Tibermacine, Christelle Urtado and Sylvain Vauttier: Using Concept Lattices to Support Web Service Compositions with Backup Services. In Proceedings of the 5th International Conference on Internet and Web Applications and Services (ICIW'10), pages 363-368. Barcelona, Spain, May 2010. IEEE Computer Society.

[*C*] Zeina Azmeh, Jean-Rémy Falleri, Marianne Huchard and Chouki Tibermacine: Automatic Web Service Tagging Using Machine Learning and WordNet Synsets. Revised selected papers from (WEBIST'10). Lecture Notes in Business Information Processing (LNBIP) 75, pages 46-59. June 2011. Springer-Verlag.

[*D*] Zeina Azmeh, Fady Hamoui, Marianne Huchard, Nizar Messai, Chouki Tibermacine, Christelle Urtado and Sylvain Vauttier: Backing Composite Web Services Using Formal Concept Analysis. In Proceedings of the 9th International Conference on Formal Concept Analysis (ICFCA'11), pages 26-41. Nicosia, Cyprus, May 2011. LNCS/LNAI, Springer-Verlag.

[*E*] Zeina Azmeh, Maha Driss, Fady Hamoui, Marianne Huchard, Naouel Moha and Chouki Tibermacine: Selection of Composable Web Services Driven by User Requirements. In the Proceedings of the 9th International Conference on Web Services (ICWS'11). Washington DC, USA, July 2011. IEEE Computer Society.

Part I

State of the Art

Chapter 2

Web Service Basics

Contents

1 Introduction

Service-Oriented Architecture (SOA) [3] is a design paradigm that suggests the construction of applications by the composition of existing software modules that are called services. In this vision, Web services are considered an important realization of this application design paradigm.

In this thesis, we are proposing a solution for Web service selection, for facilitating Web services utilization. This is especially important for the realization of Web Service-Oriented Architectures (WSOA).

In this chapter, we give the key basics and definitions surrounding the Web service technology. We also explain the principles of Web service composition for achieving a WSOA. We conclude by listing the problems that are related to Web services and their standards.

2 What are Web Services?

In the literature, we can find several definitions for Web services. We expose a definition from [1] that introduces a Web service as follows:

"A Web service is a platform-independent, loosely coupled, self-contained, programmable Web-enabled application that can be described, published, discovered, coordinated, and configured using XML artifacts (open standards) for the purpose of developing distributed interoperable applications."

Indeed, a Web Service is an application that is accessible through the Internet. It is self-contained, meaning that it functions independently without external support. It is self-describing through an XML-based public interface containing a URL for service binding, and a specification of the functionality. This public interface can be discovered by other software systems. Then, they may interact with the Web service in a manner prescribed by its definition, using XML-based messages conveyed by Internet protocols [11, 2].

Dealing with Web services involves three actors, a *provider*, a *registry*, and a *consumer*, as illustrated in Figure 2.1. This figure represents the SOA triangle [3]. It illustrates the interaction between the three actors, as well as the used standards. A service provider publishes his service description at a service UDDI registry using a publishing interface. The public interfaces and binding information of the registered services are clearly defined in the WSDL standard language. A registry organizes the published services and provides a query interface that enables a service consumer to search for a needed service, and obtain its provider's location information. A service consumer can then interact with a service provider through the SOAP protocol.

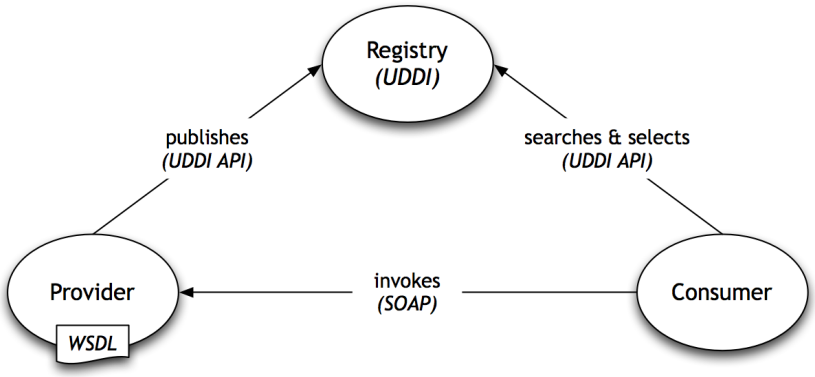


Figure 2.1: The SOA triangle.

The three previous actors perform three fundamental Web services actions, each of which is related to a standard.

2.1 Describing a Web Service

Describing a Web service is the action using which, a service can be represented (exposed and advertised) to the external world. A functionality of a service is represented to the external world by the means of an abstract interface described using the standard Web Service Description Language (WSDL) [5].

2.1.1 Functional Properties

WSDL is an XML-based language that contains several elements to describe a Web service's functionality. It can be regarded as two parts: an abstract part and a concrete one. The abstract part describes the service interface, i.e. the operations together with their parameters and data types. The data types are described as XML schema elements. The concrete part describes how to bind to and invoke the concerned service, by specifying the binding protocol together with the service's endpoint. For example, in Figure 2.2, we can see the WSDL interface for a math service, called MathService.

```
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions ... >
<!--The types section-->
  <wsdl:types>...</wsdl:types>
<!--The messages section-->
  <wsdl:message name="AddSoapIn">...</wsdl:message>
  <wsdl:message name="AddSoapOut">...</wsdl:message>
  <wsdl:message name="AddHttpGetIn">...</wsdl:message>
  <wsdl:message name="AddHttpGetOut">...</wsdl:message>
<!--The portTypes section-->
  <wsdl:portType name="MathServiceSoap">...</wsdl:portType>
  <wsdl:portType name="MathServiceHttpGet">...</wsdl:portType>
<!--The bindings section-->
  <wsdl:binding name="MathServiceSoap" type="tns:MathServiceSoap">...</wsdl:binding>
  <wsdl:binding name="MathServiceHttpGet" type="tns:MathServiceHttpGet">...</wsdl:binding>
<!--The service section-->
  <wsdl:service name="MathService">...</wsdl:service>
</wsdl:definitions>
```

Figure 2.2: The WSDL interface of the MathService.

We notice that it has a root definitions element <wsdl:definitions>, which is structured into five sections. We list them bottom-up, as follows:

– service section <wsdl:service>, which specifies the service name, together with its endpoint location for each of the provided interface (portType). For example, in Figure 2.3, we notice that the MathService has two interfaces "MathServiceSoap" and "MathServiceHttpGet", for which it specifies two distinct endpoints.

```
<!--The service section-->
  <wsdl:service name="MathService">
    <wsdl:port name="MathServiceSoap" binding="tns:MathServiceSoap">
      <soap:address location="..." />
    </wsdl:port>
    <wsdl:port name="MathServiceHttpGet" binding="tns:MathServiceHttpGet">
      <http:address location="..." />
    </wsdl:port>
  </wsdl:service>
```

Figure 2.3: The service part inside the MathService WSDL interface.

– bindings section <wsdl:binding>, which specifies the protocol for each binding, together with the provided operations. For example, in Figure 2.4, we see that the MathService

provides two bindings "MathServiceSoap" and "MathServiceHttpGet" corresponding to the two interfaces it provides. For the "MathServiceSoap" interface, the binding specifies that the used protocol is SOAP using <soap:binding>.

```
<!--The bindings section-->
  <wsdl:binding name="MathServiceSoap" type="tns:MathServiceSoap">
    <soap:binding transport="..." />
    <wsdl:operation name="Add">
      <soap:operation soapAction="..." />
      <wsdl:input>...</wsdl:input>
      <wsdl:output>...</wsdl:output>
    </wsdl:operation>
  </wsdl:binding>

  <wsdl:binding name="MathServiceHttpGet" type="tns:MathServiceHttpGet">...</wsdl:binding>
```

Figure 2.4: The binding part inside the MathService WSDL interface.

– portTypes section <wsdl:portType>, which specifies the interfaces (portTypes) that the service provides. For example, in Figure 2.5, we see that the MathService provides two portTypes "MathServiceSoap" and "MathServiceHttpGet". We see also that the "MathServiceSoap" provides an operation called "Add", which takes an input message called "AddSoapIn" and returns an output message "AddSoapOut".

```
<!--The portTypes section-->
  <wsdl:portType name="MathServiceSoap">
    <wsdl:operation name="Add">
      <wsdl:input message="tns:AddSoapIn" />
      <wsdl:output message="tns:AddSoapOut" />
    </wsdl:operation>
  </wsdl:portType>

  <wsdl:portType name="MathServiceHttpGet">...</wsdl:portType>
```

Figure 2.5: The portType part inside the MathService WSDL interface.

– messages section <message>, where the input/output parameters are specified for each operation. For example, in Figure 2.6, we see the input/output messages for the "Add" operation. The input message "AddSoapIn" has one parameter of type "AddRequest", and the output message "AddSoapOut" that has one parameter of type "AddResponse".

```
<!--The messages section-->
  <wsdl:message name="AddSoapIn">
    <wsdl:part name="parameters" element="tns:AddRequest" />
  </wsdl:message>
  <wsdl:message name="AddSoapOut">
    <wsdl:part name="parameters" element="tns:AddResponse" />
  </wsdl:message>
```

Figure 2.6: The messages defined for the "Add" operation inside the MathService.

– types sections <wsdl:types>, where the parameter types of each operation inside the WSDL file are specified as an XML schema element. For example, the MathService de-

fines a schema of two elements "AddRequest" and "AddResponse", as shown in Figure 2.7. They represent the parameters types for "AddSoapIn" and "AddSoapOut" messages, respectively. The "AddRequest" element is a sequence of two elements of type *float*, while the "AddResponse" has only one element of type *float*.

```
<!--The types section-->
  <wsdl:types>
    <s:schema ... >
      <s:element name="AddRequest">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="1" maxOccurs="1" name="A" type="s:float" />
            <s:element minOccurs="1" maxOccurs="1" name="B" type="s:float" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="AddResponse">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="1" maxOccurs="1" name="AddResult" type="s:float" />
          </s:sequence>
        </s:complexType>
      </s:element>
      ...
    </s:schema>
  </wsdl:types>
```

Figure 2.7: The WSDL types defined for the MathService.

2.1.2 Non-Functional Properties

Non-functional properties of a Web service are mainly used to model Quality of Service (QoS) attributes. QoS refers to the ability of the Web service to respond to expected invocations and to perform them at a level that corresponds to the mutual expectations of both its provider and its consumer [1]. The QoS is influenced by the Internet's dynamic and unpredictable nature. Therefore, delivering good QoS is a critical and significant challenge.

QoS attributes such as constant availability, connectivity, and high responsiveness are the key to keeping a service competitive and viable. They represent important criteria to determine the service usability and utility, which influence the Web service's popularity. Especially with the ever increasing number of functionally similar Web services being made available on the internet, there is a need to be able to distinguish between them using a set of well-defined QoS attributes and values. Thus, QoS has become an important part of service description, for a better service selection [12] and composition [13].

Some of the QoS attributes can be calculated by the consumer of a Web service (like: response time), while other attributes must be calculated and provided by the Web service's provider (like: scalability and security). Below, we list some of the main QoS attributes:

- **Availability:** which indicates the Web service's presence during a period of time, as a percentage.
- **Performance:** which is measured in terms of throughput and latency. Throughput rep-

resents the number of Web service requests served at a given time period. Latency is the time between sending a request and receiving the response.

- **Reliability:** Reliability is the quality aspect of a Web service that represents the degree of being capable of maintaining the service and service quality. The number of failures per month or year represents a measure of reliability of a Web service.
- **Response time:** which is the average time (in milliseconds) that is needed to obtain a response from a Web service.
- **Security:** which represents the provider's approaches and levels of providing security, by authenticating the parties involved, encrypting messages, and providing access control.

2.2 Discovering a Web Service

Web service discovering is associated to registries and repositories, where one can search for a needed service. A registry must provide a fairly precise search capability to render the discovery action more efficient and less time-consuming. Initially, Web service registries were supposed to be built upon a standard called UDDI [6], but discovery mechanisms are not limited to this standard.

Universal Description, Discovery and Integration (UDDI)

UDDI [6] is an XML-based standard that was proposed for Web service registries, for allowing providers to publish their services, so that they can be located afterwards by consumers.

A UDDI registry is structured into three layers, as illustrated in Figure 2.8:

- *businessEntity* or *white pages*, which list the providers and their related information;
- *businessService* or *yellow pages*, which provide a classification of services based on standard taxonomies;
- *bindingTemplate* or *green pages*, which provide information about the service bindings and their types *tModel*.

These layers permit searching a UDDI registry according to the three data types: *businessEntity*, *businessService*, and *tModel*. Thus, the possible searching queries are: *search for business* by (business name, identifier, category, discovery URL); *search for service* by (service name, category), and *search for service type* by (service type name, category).

In our days, Web service discovery is not limited to UDDI-based registries. Service discovery can be carried out through either Web service portals (service search engines) or through websites of certain service providers. The previous SOA triangle can be rather represented as in Figure 2.9.

Web Service Search Engines

Web service search engines use focused crawlers as well as manual registration to gather WSDL interfaces and index them. They index Web services by the keywords they extract from their

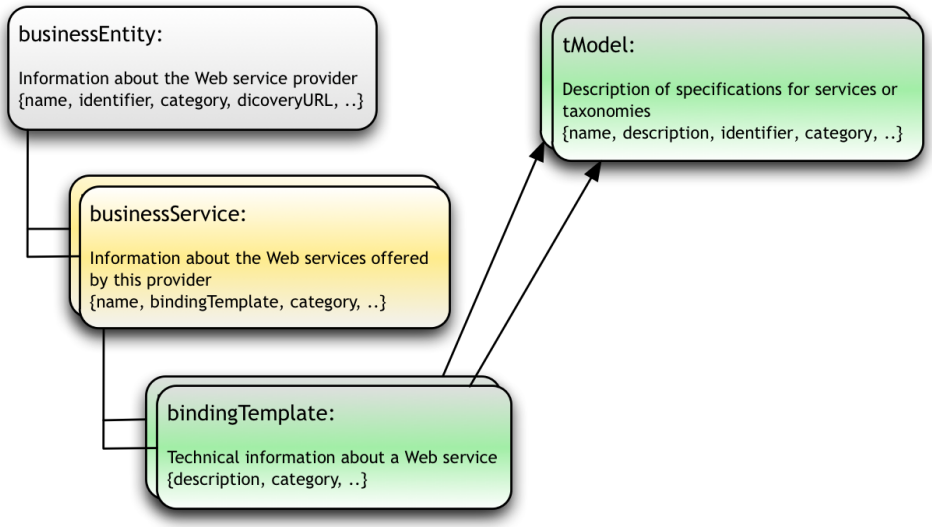


Figure 2.8: Overview of UDDI data structure.

WSDL interfaces. We mention two popular service search engines: Seekda [8] and Service-Finder [14]. These two search engines enable also some Web 2.0 features, like: user tagging, user textual description adding, and user rating. They also provide us with calculated values for two QoS attributes: availability and response time.

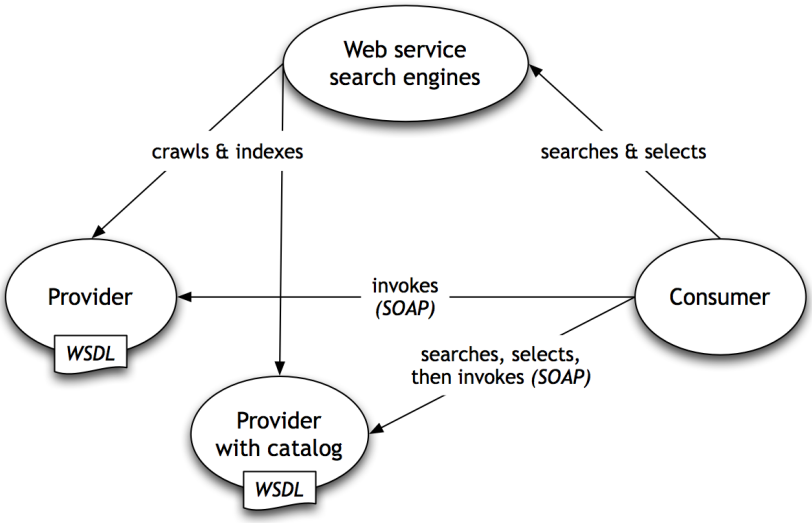


Figure 2.9: The SOA triangle nowadays.

2.3 Invoking a Web Service

Web service invocation describes a client-server communication and is embodied in the messages exchanged between a service provider and a service consumer. These messages are described by a standard called SOAP [7], and are transmitted over an HTTP connection.

Object Access Protocol (SOAP)

SOAP [7] is an XML-based standard for exchanging structured messages for invocation and result response. A SOAP message is an XML document consisting of a SOAP envelope, an optional SOAP header, and a SOAP body. The SOAP message header contains information that allows the message to be routed to its final destination. The envelope is the root element of the message. It defines how the message should be handled and by whom. The header can be used to add features to a SOAP message. The body is a container for mandatory information intended for the ultimate recipient of the message.

3 Web Service-Oriented Architectures (WSOA)

Web services represent an important realization of SOA. We cite [3] for a definition of a contemporary SOA:

> *"Contemporary SOA represents an open, extensible, federated, composable architecture that promotes service-orientation and is comprised of autonomous, QoS-capable, vendor diverse, interoperable, discoverable, and potentially reusable services, implemented as Web services."*

Service Composition [15] is the process of developing a composite service. Service composition can be either performed by composing elementary or composite services. Composite services in turn are recursively defined as an aggregation of elementary and composite services. A client invoking a composite service can itself be exposed as a Web service. Service composition is a very complex and challenging task. It becomes necessary to develop a Service Composition Middleware to support composition in terms of abstractions and infrastructure as well.

Service composition could be static or dynamic. These two types of composition strategies concern the time when Web services are composed. They are equivalent to design-time and run-time composition, respectively. Static composition takes place during design-time when the architecture and the design of the software system are planned. The components to be used are chosen, linked together and finally compiled and deployed. This may work fine as long as the Web service environment business partners and service components does not or only rarely change. Microsoft Biztalk and Bea WebLogic are examples of static composition engines [16].

Service composition can be of two types, either orchestration or choreography [17]. Orchestration is where a central or master element controls all aspects of the process. Choreography is where each element of the process is autonomous and controls its own behavior.

3.1 Orchestration

In orchestration, the involved web services are under control of a single endpoint central process (another web service). This process coordinates the execution of different operations on the Web services participating in the process. The invoked Web services neither know and nor need to know that they are involved in a composition process and that they are playing a role in a business process definition. Only the central process (coordinator of the orchestration) is conscious

of this aim, thus, the orchestration is centralized through explicit definitions of operations and the invocation order of Web services [18].

Today's standards for orchestration include BPMN (Business Process Modeling Notation) [19] for defining the visual representation of the sequence, and BPEL (business process execution language) as the 'code' that executes the sequence. Almost all SOA infrastructures provide some type of run-time BPEL engine, and most BPM products already support, or are in the process of supporting these standards in their modeling and run-time.

3.2 Choreography

Choreography, in contrast, does not depend on a central orchestrator. Each Web service that participates in the choreography has to know exactly when to become active and with whom to interoperate. Choreography is based on collaboration and is mainly used to exchange messages in public business processes. All Web services which take part in the choreography must be conscious of the business process, operations to execute, messages to exchange as well as the timing of message exchanges [18]. In Figure 2.10, we show orchestration versus choreography.

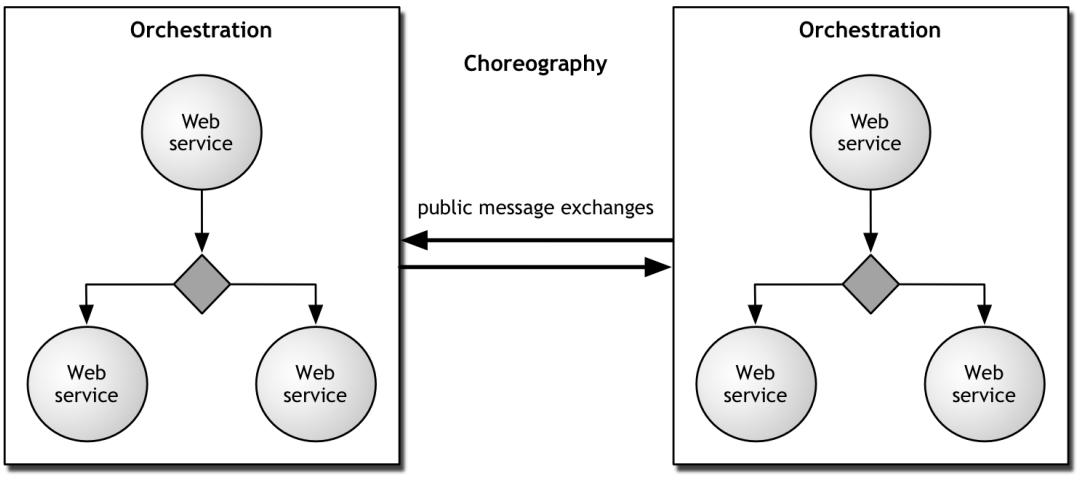


Figure 2.10: Orchestration vs. choreography.

3.3 Business Process Execution Language (BPEL)

BPEL is an XML-based language that defines a new Web service by orchestrating a set of existing services according to a desired functionality. The resulting Web service orchestration is called a business process, the involved services are called *partners*, and the message exchange is referred to as an activity.

A *partner* represents both a consumer of the service that is provided by the business process, and a provider of a service to the business process. Thus, it can be a Web service that a process invokes, or any client that invokes a process. It represents a mapping to a WSDL portType description of a partner's Web service.

A BPEL process interacts with each partner using a *partnerLink* construct, which represents a channel for establishing a conversation with a partner. In other words, a business process contains a set of activities and invokes external partner services, and can be exposed as a Web service by a WSDL description file.

A BPEL process contains several elements, we list and define the main ones:

- The process element: It is the root element of BPEL process definition. It has a name attribute and it is used to specify the definition's related namespaces.
- Partner Links elements: These elements in a BPEL process define the interaction of participating services with the process. They describe the process and services roles in the flow, as well as defining the kind of data to be handled by the parties defined in those roles.
- Variables elements: A BPEL process allows to declare variables in order to receive, manipulate, and send data. These variables hold the state of a business process, to facilitate stateful interactions among Web services.
- Activities elements: Activities are the processing steps, performed in a BPEL body. BPEL supports basic as well as structured activities.

The main basic activities are:

- <invoke>: invoking Web service operations.
- <receive>: waiting for a process operation to be invoked by an external client.
- <reply>: generating a response for synchronous operations.
- <assign>: copying data between variables.

The main structured activities are:

- <sequence>: defining a set of activities that will be executed in an ordered sequence.
- <flow>: defining a group of activities which will be invoked in parallel.
- <switch>: Case-switch construct for implementing branches
- <while>: defining loops.
- <pick>: selecting one of several alternative paths.
- <if>: expressing a condition.

In Figure 2.11, we illustrate an overview of a BPEL process structure. Inside this process, we can see several activities that define the business logic, the data flow, and the order of Web services invocation. The process is exposed to the external world as a new Web service. It can be invoked by a client, who links to its portType through a partnerLink. The process execution starts at the activity <receive>, it invokes the participating partners (Web services), and the execution ends at the <reply> activity, which returns the process response.

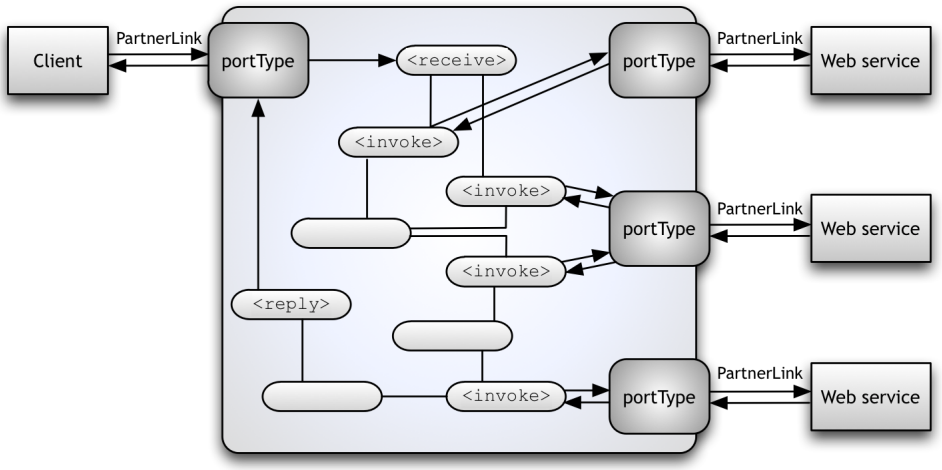


Figure 2.11: The structure of a BPEL process with some activities.

BPEL distinguishes between two levels of process description abstract and executable. An abstract business process specifies the external message exchange between parties only. It does not contain the internal details of the process flow, and it is not executable. While, an executable process describes both the external message exchange and the complete internal details of the process flow and business logic. It is executable, as its name implies.

4 Summary

In this chapter, we presented the basic definitions for Web services, their description, discovery, and invocation, along with their main standards. We also presented the notion of building composite functionality based on interacting services.

As we have seen so far, the standards on which Web services are built cause the main challenges surrounding the utilization of Web services. We discuss these challenges according to each Web service standard:

– WSDL description: depends on syntactic information only. It does not support semantics, nor it does support providing information related to the service's QoS. In addition to that, it does not impose any constraints on the providers for supporting a textual documentation about the service functionality, nor for following naming conventions.

– UDDI discovery: has failed to be the primary discovery mechanism. Alternatives are embodied in Web service search engines and providers' catalogs. They support searching by keywords only, which may not be adequate for an efficient and precise service retrieval. Especially that WSDL interfaces have a small quantity of text inside them, according to which, they can be indexed. These search engines do not allow searching for a needed operation, nor searching along with QoS constraints.

– BPEL composition: is built statically at design-time. This means that the user must have a previous knowledge about the existing services, and must select the services prior

to composition. This imposes going through the hard and time-consuming issue of Web service discovery and selection.

Thus, service selection considering current mechanisms and circumstances is not efficient nor practical. In the following chapter, we study the literature's proposed solutions for this problem, with their advantages and shortcomings.

Chapter 3

Related Work

Contents

1 Introduction

In this chapter, we study the state of the art dealing with the selection of a Web service to use. We evaluate the studied works according to the following criteria:

- design-time or run-time: whether the studied work presents a static or dynamic solution,
- semantics: whether the studied work uses semantic Web services or not,
- discovery: whether the studied work depends on a registry or a predefined set of services,
- functionality: whether the studied work proposes a filter for checking the compatibility of a discovered service with the required functionality,
- composability: whether the studied work can identify composable services,
- QoS: whether the studied work considers the QoS for ranking the services,
- backups: whether the studied work offers facilities to discover similar services that can replace each other.

We group the studied works into two categories. The first category represents the works that propose a solution for the selection of a single independent service. The second category represents the works support the selection of a set of services for building a composition. We conclude the chapter by a comparison of the studied works, according to the previous criteria.

2 Works about Selecting an Independent Web Service

We list the works in this category according to four sets of classification-based works.

2.1 Information Retrieval based Classification

A quick overview of some of the works can be acquired from [20, 21]. These works propose several approaches, based mainly on information retrieval (IR) [22] mechanisms.

The vector space model is used for service retrieval in several existing works as in [23, 24, 25]. Terms are extracted from every WSDL file and vectors are built for describing services. A query vector is also built, and similarity is calculated between the service vectors and the query vector. This model is sometimes enhanced using WordNet structure matching algorithms to ameliorate similarity scores as in [24], or by partitioning the search space into smaller subspaces as in [25].

Many approaches use machine learning techniques, in order to discover and group similar services. In [26, 27], service classifiers are defined depending on sets of previously categorized services. The resulting classifiers are then used to deduce relevant categories for new services. In case there are no predefined categories, unsupervised clustering is used. In [28], the CPLSA approach is defined that reduces a service set then clusters it into semantically related groups.

In [29], the Woogle Web service search engine is presented, which takes a needed operation as input and searches for all the services that offer a similar operation. Woogle supports similarity search for Web services, like finding similar Web service operations and finding operations that compose with a given one. It considers similarity between the textual descriptions of both the operations and the entire Web services, as well as the similarity between parameter names. It clusters parameter names in the collection of Web services into semantically meaningful concepts. These concepts can be used instead of parameters to achieve good similarity measures by comparing the concepts that input/output parameters belong to.

2.2 Concept Lattice based Classification

In [30], Aversano *et al.* classify Web services using FCA as a means for WSDL browsing. Their formal contexts are composed according to three levels: service level, operation level and type level, together with keywords. These keywords are identified from the WSDL files by applying vector space metrics with the help of WordNet to discover synonyms. The resulting service lattice indexes Web services: it highlights the relationships between the services and permits the identification of different categorizations of a certain service.

In [31], Bruno *et al.* also use keywords extracted from services' interfaces together with FCA to build a Web services lattice. They analyze the extracted words, process them using WordNet and other IR techniques. Then, they classify them into vectors using support vector machines (SVM). The obtained vectors categorize the services into domains. Then, service lattices can be obtained for each domain using FCA.

In [32], Peng *et al.* present an approach to classify and select services. They build lattices upon contexts where objects are Web services and attributes represent the operations of these services. The approach allows similar services clustering by applying similarity search techniques that compare operation descriptions and input/output messages data type. In Peng et al. [33], similarity values are calculated for service operations, and depending on a chosen threshold, a service lattice is built.

Fenza and Senatore [34] describe a system for supporting the user in the discovery of semantic Web services that best fit personal requirements and preferences. Through a concept-based navigation mechanism, the user discovers conceptual terminology associated to the Web services and uses it to generate an appropriate service request which syntactical matches the names of input/output specifications. The approach exploits fuzzy FCA [35] for modelling concepts and relative relationships elicited from Web services. After request formulation and submission, the system returns the list of semantic Web services that match the user query.

In [36, 37], Chollet *et al.* propose an approach based on FCA to organize the service registry at runtime and provide the best service selection among heterogeneous and secured services. The service registry is viewed as a formal context where the services are the objects and the services types, functional, and non-functional characteristics (security characteristics) are attributes.

In [38], Driss *et al.* propose a requirement-centric approach to Web service, modelling, discovery, and selection. They consider formal contexts with services as objects and QoS characteristics as attributes. The obtained lattices are used to check out relevant (that best fit functional requirements) and high QoS Web services.

2.3 QoS-based Classification

In addition to these works [36, 37, 38], we can find more works that took into consideration QoS information, as below.

In [39], the authors present a model for discovering Web services based on QoS constraints. They extend the basic Web service model (provider - registry - consumer), by adding another entity called the *QoS certifier*. This QoS certifier verifies the service provider's QoS claims before the registration of a service in the UDDI registry. A service consumer may add some QoS constraints to the functional requirements. The UDDI data structure is extended by an additional element called *qualityInformation*, which provides the description of different QoS aspects, such as availability, reliability or performance.

Several other works propose to extend the basic Web service model in a similar way to the previous work. The works presented in [40, 41, 42, 43, 44] propose to add a QoS broker for Web services selection and QoS management. The model is now (provider - registry - consumer - broker). The role of this QoS broker is to support Web services QoS verification, certification, confirmation, selection and monitoring. It acts as an intermediary third party for Web services selection and QoS negotiation on behalf of the consumer.

2.4 Semantic-based Classification

In [45], they use a semantic registry for semantic Web services, which are equipped with an exploitation language for supporting semantic based process discovery. Semantic Web services are being described in an ontology of services, with a subsumption relationship. Their input and output parameters refer to concepts of a domain ontology.

In [46], They propose an approach based on service ontologies and semantic indexations. They propose a persistent architecture centred around ontology-based database to store and index the various services, as well as their compositions. The prototype implements semantic concepts for service and workflow. This enables storing, retrieving, reusing existing services and workflows, and building new ones incrementally.

In [47], the authors propose a QoS based semantic Web service selection mechanism. They use the Web Service Modeling Ontology (WSMO) [48] to describe a QoS model, including specific quality metrics, value attributes, and their respective measurements. They take into consideration user quality requirements.

In [49], the authors propose a powerful algorithm for semantic Web service matching which considers service specialization. However, this method assumes that both services and requests are specified using the same ontology.

In [50], Benatallah et al. propose a matchmaking algorithm. It takes as input a service request and an ontology, then finds a set of services whose descriptions contain as much common information with the query as possible, and as little extra information with the query as possible.

3 Works about Selecting Composable Web Services

In this section, we describe two sets of works, according to the considered composition (manual or automatic).

3.1 Manual Composition

In this section, we cite several tools (Triana [51], CAT [52], SWORD [53], WSTK [54], ZenFlow [55], BPEL2B [56], GWE [46]), which help users in building their desired composition. We choose to describe the three following tools: Triana [51], which is based on WSDL; CAT [52], which is based on WSDL that is supported by an ontology; and SWORD [53], which defines its own model for service description and selection.

Triana [51] is a graphical Web service composition toolkit that help users graphically and transparently create Web services workflows. A composite service is created by dragging the services and connecting them. Triana supports loops and conditional constructs. Composed applications may be written as BPEL4WS graphs by using the Triana pluggable architecture and executed from within Triana or any Web services choreography engine. Triana also includes

a facility which allows users to publish composite services to the network.

CAT [52] is a tool that guides users in sketching a composition of services by exploiting their semantic description. It first takes existing service descriptions (WSDL) and extends them with off-the-shelf domain ontologies. Parameters from WSDL messages can then be mapped to terms in the domain ontology. It then use a task ontology to describe abstract types of operations and services. It uses these ontologies in examining a user's solution and generating suggestions about how to proceed.

SWORD [53] is a toolset that allows service developers to quickly compose base Web services to realize new composite Web services. SWORD does not rely on the actual deployment of any specific semantic markup language by service providers. It defines its own simple world model based on defining for each Web service logic rules. Then, the analysis of functionality and composition is based on these rules. A composite service can be realized by existing services and to generate the execution plan that when executed instantiates the composite service.

3.2 Automatic Composition

In [57], the authors present an approach for dynamic Web service composition that takes into account the composition's overall quality. The proposed approach extends the heuristic-based approach for dynamic Web service composition proposed by [58] by adding QoS constraints to the heuristic. The approach takes into account both the semantic description of a service and its non-functional properties that compose the quality it delivers. The algorithm receives as input a request, which consists of the provided input concepts, required output concepts and QoS constraints. It produces as output a set of services that can provide together the required concepts specified in the request. Each concept in the request input or output is defined in a domain ontology. Each QoS constraint consists of a triple: the quality criterion, a value representing a constraint on this criterion, and a weight representing the user's preference for this criterion.

The Web Services Composition Platform, StarWSCoP (Star Web Services Composition Platform) [16], is introduced with several modules: an intelligent system to decompose user requirements; a service registry to provide a Web service repository; a service discovery engine to find proper services; a composition engine; a wrapper to achieve interoperability of heterogeneous services; a QoS estimation; and an event monitor to monitor events and notify the composition engine. It focuses on QoS-based dynamic Web services composition by extending WSDL descriptions with QoS attributes, such as time, cost or reliability.

In [13], is presented Agflow, a QoS-aware middleware supporting quality-driven Web service compositions. User satisfaction is expressed as utility functions over QoS attributes, while satisfying the constraints is set by the user and by the structure of the composite service. Two selection approaches are described and compared: one based on local (task-level) selection of services, and the other based on global allocation of tasks to services using integer programming. A composite service is specified as a collection of generic service tasks described in terms of service ontologies and combined according to a set of control flow and data flow dependencies. AgFlow uses statecharts to represent these dependencies. Services are created using IBM's Web

Services Toolkit (WSTK) [54].

Very similar to previous, [59] presents a broker-based architecture to facilitate the selection of QoS-based services. The objective of service selection is to maximize an application-specific utility function under QoS constraints. The problem is modeled in two ways: the combinatorial model and the graph model. The combinatorial model defines the problem as a multi-dimension multi-choice 0-1 knapsack problem (MMKP) [60]. The graph model defines the problem as a multi-constraint optimal path (MCOP) problem.

Another similar work presented in [61]. It proposes an approach for achieving dynamic semantic Web service composition. It is based on the METEOR-S Web service composition framework, on which is added a constraint analyzer module. This module uses an integer linear programming solver for process optimization based on process and business constraints. It proposes a workflow QoS model to enable a global optimization and dynamic composition of service processes.

4 Discussion

In the previous sections, we presented several works for Web service selection and providing help to users for building their desired compositions. We presented the works according to two main categories: works for the selection of a single Web service, and works for the selection of a set of services for building a composition.

We notice that the works that depend only on the syntactic information inside the WSDL description may not be sufficient for guiding a user to select a suitable service. This is basically because inside a WSDL interface, there may not exist enough textual description, which can be used to index a service sufficiently. Thus, when retrieving services depending on such an index, several impertinent services may be retrieved. This complicates the selection of a needed service.

To face the insufficient syntactic side of Web services, several approaches were proposed for dealing with semantic Web services. In this field of works, the selection of a pertinent Web service depends on a shared knowledge between the provider and the consumer. This shared knowledge is embodied in an ontology. This kind of works may solve the problem of selection, but under the condition of having a unique ontology. If several ontologies were used, ontology mapping must be carried out, which is yet another challenge.

In order to have a better refined selection, several works took into consideration QoS information. QoS is an important factor for distinguishing between Web services, especially when having several functionally similar services.

The previous category of works support Web service discovery but do not allow users to specify the needed parameters types. Moreover, they do not support the retrieval of composable services. Thus, they do not facilitate building service compositions.

In the second category of works, we listed a set of works for supporting manual composition

of services, and another one for automatic composition.

In the manual composition building, users can search by keywords and retrieve services to be composed in a graphical interface. Users still have to check for the composability between the services in order to build their composition. Such works do not provide users with QoS information, nor enable searching by a required QoS level.

In the automatic composition building, the works we presented are all based on semantics (which imposes other issues, as we mentioned earlier). Furthermore, they assume that Web services are annotated with semantic information (beyond WSDL). Such semantic information might not be available for current Web services, although it might become available in the future [9]. Some of these works calculate the global QoS value for the whole composition. Users are not allowed to specify a needed QoS for a certain service. Moreover, QoS is specified by numeric values, which may not always be so significant and easy to be determined by users.

The manual and automatic composition approaches that we studied do not support service backups. However, some of the automatic composition works support dynamic reconfiguration for the business process, when a service changes (disappear or have different QoS values). This reconfiguration can cause an overhead if a backup is needed, because of the several remote interactions with the service registry. Thus, the efficiency can become low.

In the following, we summarize the presented works in Table 3.1, according to the criteria presented in the beginning of this chapter. We classify the works in Table 3.1 using FCA, and we show the corresponding lattice in Figure 3.1.

In this lattice (built using ConExp [62]), we can identify the distinguished works, which can be considered better than the others, according to our defined criteria. The interesting concepts to look directly at, are the concepts at the bottom, since they verify more parts of the criteria. Thus, we can identify four interesting works:

- **Woogle**, which offers four parts of the criteria {Functionality, Backups, Discovery, Design-time}. Hence (according to our criteria), it is better than {Concept Lattices, Machine Learning, Vector Space Model, Triana};

- **Semantic-based works**, offering {Backups, Discovery, Design-time, Semantics}. Hence, they are better than {Concept Lattices, Machine Learning, Vector Space Model, Triana, CAT, SWORD};

- **UDDI + QoS**, offering {Backups, Discovery, Design-time, QoS - per service}. Hence, it is better than {Concept Lattices, Machine Learning, Vector Space Model, Triana};

- **AgFlow**, offering {QoS - global, Composability, Run-time, Semantics, QoS - per service}, and thus, it is better than {Oliveria et al., StarWSCoP, QoS-Broker, METEOR-S}.

Table 3.1: Works comparison according to the specified criteria.

Work	Design-time	Run-time	Semantic WS	Discovery	Functionality	Composability	QoS		Backups
							per service	global	
Vector Space Model	✓	×	✓	×	×	×	×	✓	
Machine Learning	✓	×	×	✓	×	×	×	×	✓
Woogle [29]	✓	×	×	✓	✓	×	×	×	✓
UDDI + QoS	✓	×	×	✓	×	×	✓	×	✓
Concept Lattice	✓	×	×	✓	×	×	×	×	✓
Semantic-based	✓	×	✓	✓	×	×	×	×	✓
Triana [51]	✓	×	×	✓	×	×	×	×	×
CAT [52]	✓	×	✓	✓	×	×	×	×	×
SWORD [53]	✓	×	✓	×	×	×	×	×	×
Oliveria et al.	×	✓	✓	×	×	✓	×	✓	×
StarWSCoP [16]	×	✓	✓	×	×	✓	×	✓	×
AgFlow [13]	×	✓	✓	×	×	✓	✓	✓	×
QoS-Broker [59]	×	✓	✓	×	×	✓	×	✓	×
METEOR-S [61]	×	✓	✓	×	×	✓	×	✓	×

5 Summary

In this chapter, we presented several works for Web service discovery and selection. These works are based on several techniques like information retrieval, semantic-based, FCA-based, or QoS-based, as we have seen.

In this thesis, we focus on non-semantic Web services, which exist in a large number on the Internet nowadays. We prefer to propose solutions to make use of these services and their large number. We are interested by two main problems: the selection of a single service, and the selection of a set of services for composition.

Concerning the first problem, we have listed a number of works that are based on information retrieval techniques. They depend on the syntactic data that are present inside the WSDL descriptions. Such techniques may not be sufficiently precise, for performing a direct selection. Users still have to filter the retrieved services, searching for the one offering the functionality that he needs. These techniques were improved in another set of works that took QoS information as a criterion for ranking the retrieved services. This can be considered as a quite good enhancement for IR-based approaches, but when considering several QoS attributes, it becomes hard to decide which service offers the best compromise. This is where FCA-based approaches can play a good role. FCA can classify Web services into concept lattices, in which, each concept represents a group of equivalent Web services regarding the shard attributes. Therefore, these

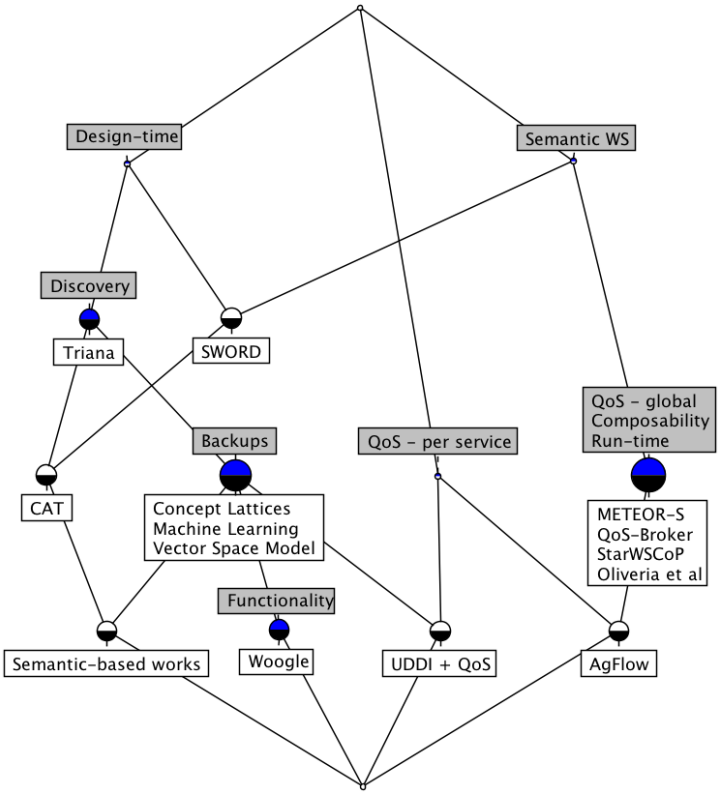


Figure 3.1: The lattice revealing the relations between the works of the state of the art.

concept lattices offer a browsing mechanism that reveals the relations between the services, and enables the identification of a service that is perhaps better than expected. A concept lattice also enables the identification of substitutes (backups) of a certain service. These backups share the same or more attributes of the service that they can substitute. This is quite useful regarding the dynamic nature of Web services.

Concerning the second problem, we have listed two kinds of works. Either manual or automatic composition techniques. When dealing with the manual technique, the user must apply a functional filtration and must check for the composability between two services. Considering the automatic composition techniques, the resulting compositions suffer from inflexibility regarding:

- the QoS, which is calculated globally for the whole composition. This does not allow for users to specify the needed QoS for each service separately;
- they do not support backups identification. Thus, they are hard to maintain in case of failures, because the selection of a replacing service is restricted by the global QoS.

In this thesis, we aim at providing a framework based on several techniques, in order to contribute in providing a solution for Web service selection. Our framework is based on Formal and Relational Concept Analysis (FCA, RCA), which are two classification techniques that are presented in the next chapter.

Chapter 4

FCA and RCA Basics

Contents

1 Introduction

In this chapter, we present the Formal Concept Analysis (FCA) classification technique, as well as its extension called the Relational Concept Analysis (RCA). We use these two techniques as a basis for our approach for Web service classification. We give the basic formal definitions for these two techniques, supported with illustrative examples. Then, we define the notions of queries and relational queries, as tools for navigating and exploiting the lattices.

2 Formal Concept Analysis (FCA)

FCA is a classification technique that takes data sets of objects and their attributes, and extracts relations between these objects according to the attributes they share [63].

2.1 Case Study

We explain the FCA technique along with a case study about Mexican dishes and their ingredients. We suppose the data set in Table 4.1.

Table 4.1: Mexican dishes and their ingredients.

Mexican dish	Ingredients
Burritos	chicken, beef, pork, vegetables, beans, rice, cheese, guacamole, sour-cream, lettuce, and flour-tortilla
Enchiladas	chicken, cheese, sour-cream, and corn-tortilla
Fajitas	chicken, beef, vegetables, cheese, guacamole, sour-cream, lettuce, and flour-tortilla
Nachos	vegetables, beans, cheese, and guacamole
Quesadillas	chicken, beef, cheese, corn-tortilla, and flour-tortilla
Tacos	chicken, beef, beans, cheese, lettuce, corn-tortilla, and flour-tortilla

2.2 Definitions

Definition 1 *A **formal context** is denoted as $K = (O, A, I)$ where O is a set of **objects**, A is a set of **attributes**, and I is a **binary relation** between O and A ($I \subseteq O \times A$). $(o, a) \in I$ denotes the fact that the object $o \in O$ is in relation through I with the attribute $a \in A$ (also read as o has a).*

A formal context is represented as a cross table in which, objects appear as row labels and attributes as column labels. A cross in the cell *(o,a)* of this table indicates that the object o has the attribute a.

From our case study, we can build a formal context of Mexican dishes $O = \{Burritos$, $Enchiladas$, $Fajitas, Nachos, Quesadillas, Tacos\}$ and their ingredients $A = \{chicken$, $beef$, $pork, vegetables, beans, rice, cheese, guacamole, sour\text{-}cream, lettuce, corn\text{-}tortilla$, $flour\text{-}tortilla\}$.

	chicken	beef	pork	vegetables	beans	rice	cheese	guacamole	sour-cream	lettuce	corn-tortilla	flour-tortilla
Burritos	×	×	×	×	×	×	×	×	×	×		×
Enchiladas	×						×		×		×	
Fajitas	×	×		×			×	×	×	×		×
Nachos				×	×		×	×				
Quesadillas	×	×					×				×	×
Tacos	×	×			×		×			×	×	×

Table 4.2: A formal context for Mexican dishes and their ingredients ($O \times A$).

Definition 2 *For a set $X \subseteq O$ of objects, we define the set $X' \subseteq A$ of attributes, which are common to the objects in X, as follows:*

$$X' = \{a \in A \mid oIa,\ \forall\ o \in X\}$$

We define correspondingly, for a set $Y \subseteq A$ of attributes, the set $Y' \subseteq O$ of objects, which have all the attributes in Y, as follows:

$$Y' = \{o \in O \mid oIa,\ \forall\ a \in Y\}$$

For example, if we take the set $X = \{Enchiladas, Quesadillas, Tacos\}$ from Table 4.2, the set of common attributes is $X' = (\{Enchiladas, Quesadillas, Tacos\})' = \{chicken, cheese, corn-tortilla\}$. In the same way, $(\{pork, rice\})' = \{Burritos\}$.

Definition 3 *A **formal concept** of the context $K = (O, A, I)$ is a pair (X, Y), where $X \subseteq O$ is called the **extent**, $Y \subseteq A$ is called the **intent**, $X' = Y$ (or equivalently $Y' = X$), meaning that a concept is a maximal collection of objects sharing a maximal collection of attributes. The set of all concepts of the context K is denoted as $\mathfrak{B}(G, M, I)$. The closure operator of X is defined as $X'' = X$.*

For example, $(\{Enchiladas, Quesadillas, Tacos\}, \{chicken, cheese, corn\text{-}tortilla\})$ is a concept, while $(\{Nachos\}, \{vegetables, beans, cheese, guacamole\})$ is not, because $(\{Nachos\})' = \{vegetables, beans, cheese, guacamole\}$ while $(\{vegetables, beans, cheese, guacamole\})' = \{Burritos, Nachos\}$.

Definition 4 *For an object $o \in O$, we define its **object intent** as $o' = \{a \in A \mid oIa\}$. Correspondingly, we also define for an attribute $a \in A$ its **attribute extent** as $a' = \{o \in O \mid oIa\}$. Thus, an **object concept** (o'', o') is denoted as γo, while an **attribute concept** (a', a'') is denoted as μa.*

For example, the object concept $\gamma(Nachos)$ is $((Nachos)'', (Nachos)') = (\{Burritos, Nachos\}, \{vegetables, beans, cheese, guacamole\})$.

Definition 5 *Having two concepts (X_1, Y_1) and (X_2, Y_2), we say that (X_1, Y_1) is a **subconcept** of (X_2, Y_2), when $X_1 \subseteq X_2$ (equivalently $Y_2 \subseteq Y_1$). Inversely, we say that (X_2, Y_2) is a **superconcept** of (X_1, Y_1). We denote the relation between these two concepts as $(X_1, Y_1) \leq (X_2, Y_2)$, and we call the relation $\leq$ the **order** relation of the concepts.*

For example, $(\{Burritos\}, \{chicken, beef, pork, vegetables, beans, rice, cheese, guacamole, sour\text{-}cream, lettuce, flour\text{-}tortilla\})$ is a subconcept of $(\{Burritos, Nachos\}, \{vegetables, beans, cheese, guacamole\})$.

Definition 6 *$\mathfrak{B}(O, A, I)$ provided with the order relation $\leq$ is a **concept lattice** and is denoted as $\underline{\mathfrak{B}}(O, A, I)$.*

Figure 4.1 illustrates the lattice built for the context shown in Table 4.2. In this lattice we can reveal many facts, as well as the relationships between the presented mexican dishes, according to the ingredients they share, for example:

– all of the mexican dishes contain *cheese*, since it appears in the top concept $c5$ that has all the dishes in its extent;

– a concept having only one object in its extent, has in its intent the attributes characterizing this object, for example: the concept $c14$ lists all the ingredients contained in an *Enchiladas* dish;

– a concept having more than one object in its extent, lists the equivalent objects regarding the attributes in its intent, for example: in concept $c6$, the dishes *Burritos*, *Enchiladas*, *and Fajitas* contain three shared ingredients, which are *cheese*, *chicken*, and *sour-cream*;

– when regarding the super and sub relations between the concepts, we can know for example that: since concept $c14$ is a subconcept of $c8$, then an *Enchiladas* dish resembles a *Quesadillas* dish but has an extra ingredient (*sour-cream*).

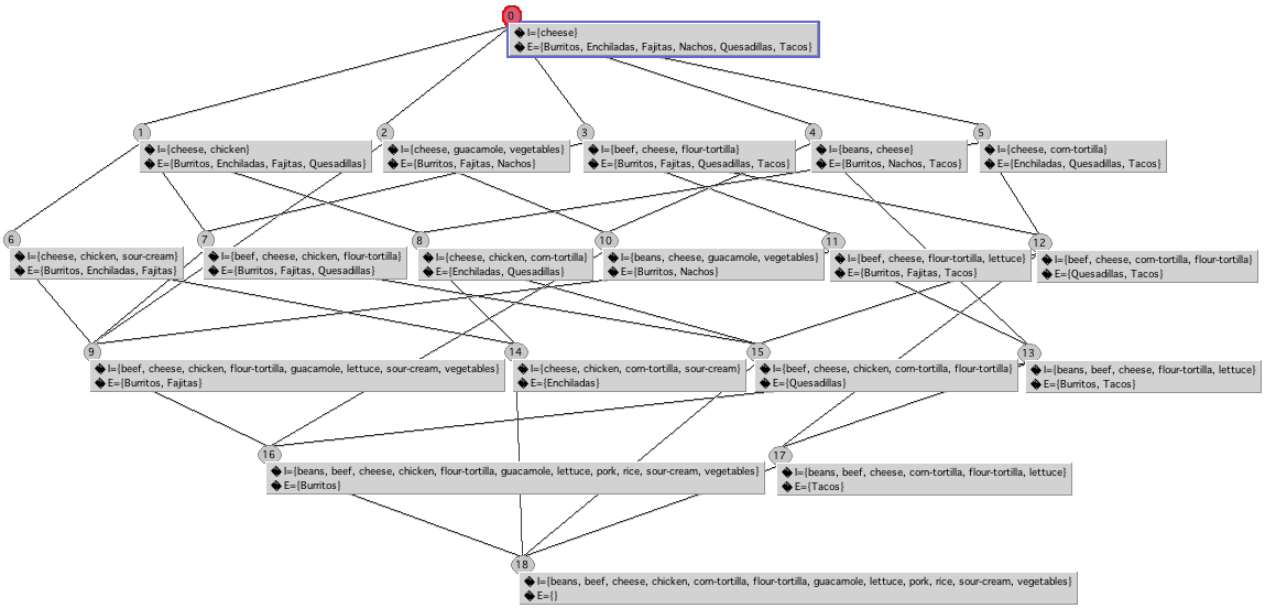


Figure 4.1: The concept lattice (built using Galicia [64]) for the context in Table 4.2.

In a concept lattice, labels can be simplified by putting down each object and each attribute only once. Like this, a lattice with reduced labels (simplified intents and extents) can be read in the same way without losing any information. In Figure 4.2, we can see the precedent lattice after reducing its labels.

Definition 7 *The Galois Sub-Hierarchy (GSH) of a concept lattice L is the sub-order of L made out of the set of attribute concepts and the object concepts. The order relation $\leq$ of concepts in GSH is the same as in the lattice.*

The GSH of the lattice in Figure 4.2 is illustrated in Figure 4.3. Using the GSH, we get the same lattice but without the concepts having empty simplified extent and intent ($c7$, $c8$, $c11$, $c12$, $c18$).

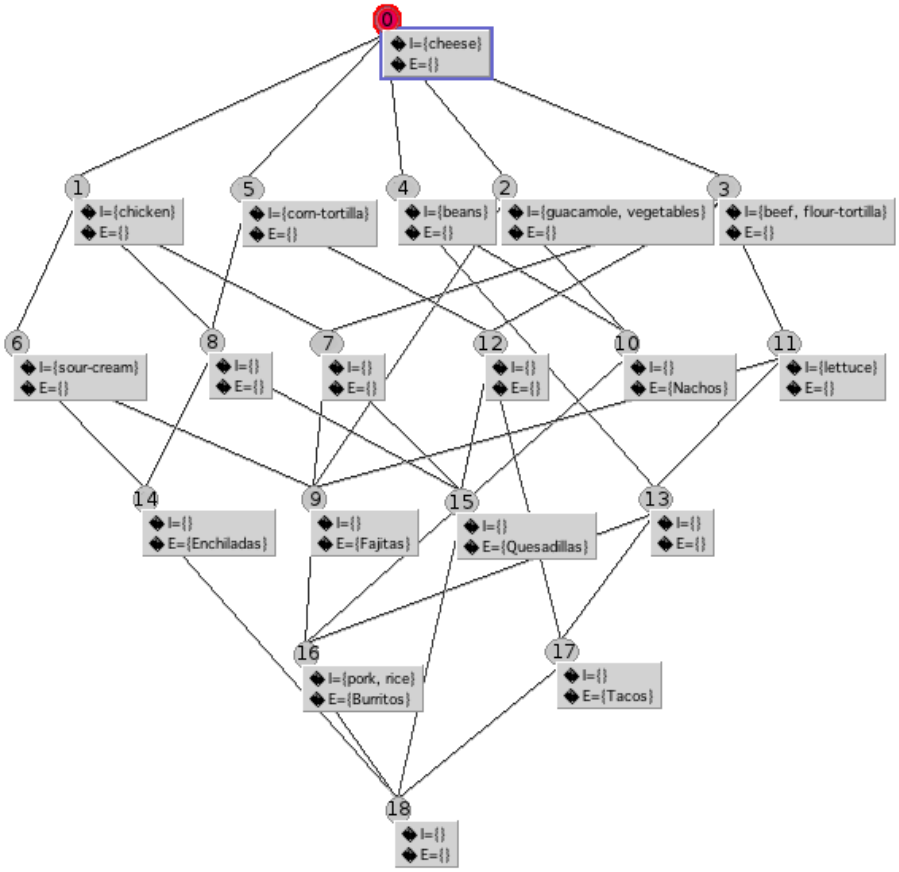


Figure 4.2: The concept lattice with simplified labels.

3 Relational Concept Analysis (RCA)

RCA [66] is an iterative version of FCA in which, the objects are classified not only according to the attributes they share, but also according to the relations between them.

3.1 Case Study

Let us take the previous case study and extend it. We suppose now having a list of countries, a list of restaurants, a list of Mexican dishes, a list of ingredients, and finally a list of salsas. We impose some relations between these entities$\{Country, Restaurant, MexicanDish, Ingredient, Salsa\}$, such that: a Country "has" a Restaurant; a Restaurant "serves" a MexicanDish; a MexicanDish "contains" an Ingredient; and finally a Salsa is "suitable-with" a MexicanDish. We express these entities and their relations by the directed acyclic graph in Figure 4.4. Below, we instantiate this entity-relationship diagram into a relational context family (RCF).

3.2 Definitions

Definition 8 *A relational context family* ***RCF*** *is a pair* $(\mathbb{K}, R)$ *where* $\mathbb{K}$ *is a set of formal (object-attribute) contexts* $K_i = (O_i, A_i, I_i)$ *and* R *is a set of relational (object-object) contexts*

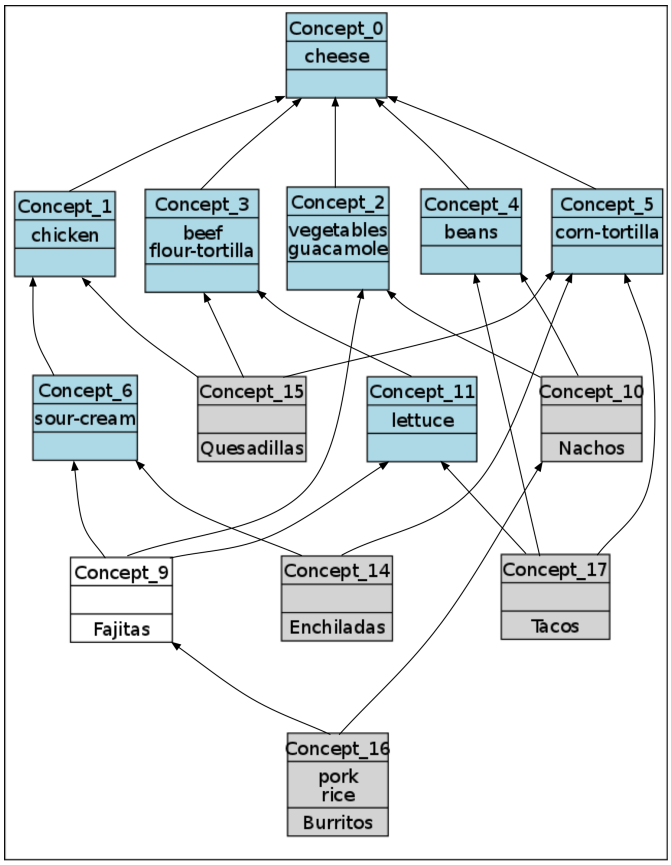


Figure 4.3: The GSH (built using erca [65]) for the context in Table 4.2.

$r_{ij} \subseteq O_i \times O_j$, *where* O_i *(domain of* r_{ij}*) and* O_j *(range of* r_{ij}*) are the object sets of the contexts* K_i *and* K_j*, respectively.*

The RCF corresponding to our example consists of four formal contexts, illustrated in Table 4.3; and five relational contexts, illustrated in Table 4.4.

An RCF is used in an iterative process to generate at each step a set of concept lattices, as illustrated in Figure 4.5. First concept lattices are built using the formal contexts only. Then, in the following steps, formal contexts are concatenated with the relational contexts enriched with knowledge obtained at a previous step. This enrichment is based on the notion of scaling operators that produce scaled relations. Hereafter, there are two examples of scaled relations.

Definition 9 *Let us define* $r_{ij}(o_i) = \{o_j \in O_j | (o_i, o_j) \in r_{ij}\}$*. The* ***exists*** *scaled relation* $r_{ij}^{\exists}$ *associated to* $r_{ij} \subseteq O_i \times O_j$ *is defined as* $r_{ij}^{\exists} \subseteq O_i \times \mathfrak{B}(O_j, A, I)$*, such that:* $(o_i, c) \in r_{ij}^{\exists} \iff \exists\, x \in r_{ij}(o_i) \ : \ x \in Extent(c)$*. Thus,* $\exists$ *is a scaling operator (existential). Let us note that in this definition,* $\mathfrak{B}(O_j, A, I)$ *is any lattice built on the objects of* O_j*.*

Definition 10 *The* ***covers*** *scaled relation* $r_{ij}^{\odot}$ *associated to* $r_{ij} \subseteq O_i \times O_j$ *is defined as* $r_{ij}^{\odot} \subseteq O_i \times \mathfrak{B}(O_j, A_j, I_j)$*, such that:* $(o_i, c) \in r_{ij}^{\odot} \implies \forall\, x \in Extent(c) \ : \ x \in r_{ij}(o_i)$*. Thus,* $\odot$ *is a scaling operator (covers).*

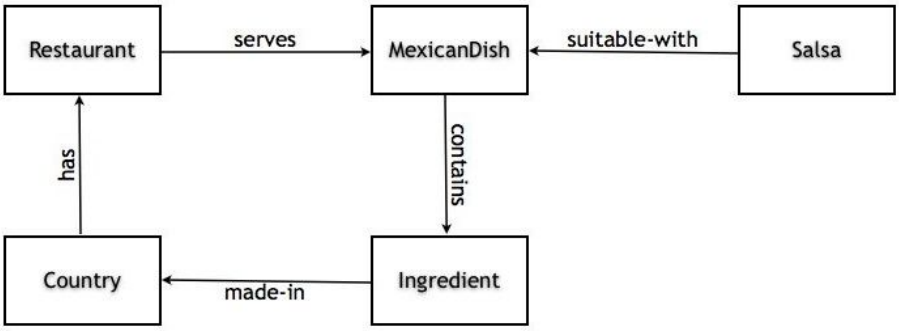


Figure 4.4: The entities of the Mexican food example.

The scaled relations are the relational contexts that are enriched with new attributes resulting from the scaling operator (exists or covers). These attributes are called relational attributes:

Definition 11 *A relational attribute results from scaling a relation $r_{ij} \in R$ where $r_{ij} \subseteq O_i \times O_j$. It expresses the relation between the objects $o \in O_i$ with the concepts of $\mathfrak{B}(O_j, A, I)$. An existential (**exists**) relational attribute is denoted by $r_{ij}^{\exists} : c_m$ where $c_m \in \mathfrak{B}(O_j, A, I)$. A **covers** relational attribute is denoted by $r_{ij}^{\odot} : c_m$. Alternatively, we write $\exists r_{ij}\ c_m$ or $\odot r_{ij}\ c_m$ to evoke notations of description logics.*

For example: $\exists$*has Concept_5*, or $\exists$*contains Concept_12* are examples of relational attributes.

Definition 12 *A concept lattice family **CLF** is a set of lattices that correspond to the formal contexts, after enriching them with relational attributes.*

For example, we used the **exists** scaled relation to generate the concept lattice family corresponding to our case study. It consists of five lattices: country lattice in Figure 4.6, restaurant lattice in Figure 4.7, mexican dish in Figure 4.8, ingredient lattice in Figure 4.9, salsa lattice in Figure 4.10.

4 Simple and Relational Queries

In this section, we define the notion of *query* and *answer to a query*, which will help us to find solutions to our problem of selecting a suitable Web service, according to certain requirements (queries). First (Section 4.1) we remind of simple queries that help in navigating concept lattices. Then (Section 4.2), we generalize to relational queries that guide the navigation between lattices of a concept lattice family along relational attributes.

4.1 Simple queries

Definition 13 *A query on a context $K = (O, A, I)$, denoted by $q|_K$ (or q when it is not ambiguous), is a pair $q = (o_q, a_q)$. o_q is the query object(s): the set of objects satisfying the query. a_q is the set of attributes defining (the constraint) of the query. By definition, we have: $o'_q \supseteq a_q$, where $a_q \subseteq A$.*

Table 4.3: The formal contexts of the Mexican Food RCF.

	ca	en	fr	lb	mx	es	us	America	Asia	Europe	
Canada	×							×			
England		×								×	
France			×							×	
Lebanon				×					×		
Mexico					×			×			
Spain						×				×	
USA							×	×			

	r1	r2	r3	r4	r5	r6	r7
Chili's	×						
Chipotle		×					
El Sombrero			×				
Hard Rock				×			
Mi Casa					×		
Taco Bell						×	
Old el Paso							×

	d1	d2	d3	d4	d5	d6	
Burritos	×						
Enchiladas		×					
Fajitas			×				
Nachos				×			
Quesadillas					×		
Tacos						×	

	i1	i2	i3	i4	i5	i6	i7	i8	i9	i10	i11	i12	
chicken	×												
beef		×											
pork			×										
vegetables				×									
beans					×								
rice						×							
cheese							×						
guacamole								×					
sour-cream									×				
lettuce										×			
corn-tortilla											×		
flour-tortilla												×	

	s1	s2	s3	s4	mild	medium-hot	hot	
Fresh Tomato	×				×			
Roasted Chili-Corn		×				×		
Tomatillo-Green Chili			×			×		
Tomatillo-Red Chili				×			×	

For example $q|_{K_{country}} = (\{England, France, Spain\}, \{Europe\})$ is a query on the country context (in Table 4.3), asking for countries in Europe.

The answer set of a query $q|_K = (o_q|_K, a_q|_K)$ is the set of objects $o_q|_K$. $\{England, France, Spain\}$ is the answer set of $q|_{K_{country}}$.

When a_q is closed, solving the query consists in finding the concept $C = (a'_q, a_q)$. To ensure that such a concept exists, a virtual query object ov_q that satisfies $ov'_q = a_q$ can be added to the context (as an additional line). Then, three types of answers can be interesting: the more precise answers are in a'_q, less constrained (with less attributes) answers are in extents of super-concepts of C, more constrained (with more attributes) answers are in extents of sub-concepts

Table 4.4: The relational contexts of the Mexican Food RCF.

	chicken	beef	pork	vegetables	beans	rice	cheese	guacamole	sour-cream	lettuce	corn-tortilla	flour-tortilla
Burritos	×	×	×	×	×	×	×	×	×	×		×
Enchiladas	×						×		×		×	
Fajitas	×	×		×			×	×	×	×		×
Nachos				×	×		×	×				
Quesadillas	×	×					×				×	×
Tacos	×	×			×		×			×	×	×

	Chili's	Chipotle	El Sombrero	Hard Rock	Mi Casa	Taco Bell	Old el Paso
Canada	×	×	×	×		×	
England		×		×		×	
France			×	×			×
Lebanon	×			×		×	
Mexico	×				×	×	
Spain				×		×	
USA	×	×	×	×	×	×	

	Burritos	Enchiladas	Fajitas	Nachos	Quesadillas	Tacos
Chili's			×		×	×
Chipotle	×					×
El Sombrero	×	×	×	×	×	×
Hard Rock			×	×		
Mi Casa	×	×		×	×	×
Taco Bell	×			×	×	×
Old el Paso						×

	Burritos	Enchiladas	Fajitas	Nachos	Quesadillas	Tacos
Fresh Tomato	×	×	×	×	×	×
Roasted Chili-Corn	×			×		
Tomatillo-Green Chili	×			×		
Tomatillo-Red Chili	×	×	×	×	×	×

of C. When a_q is not closed, and we don't use the virtual query object, searching for answers needs to find the more general concept C whose intent contains a_q.

Then, answers are dispatched similarly relatively to C.

More formally, in all the cases, if ov_q has been added to the context, $o_q|_K = \{o \mid o \in Extent(c)\ where\ ov_q \in Extent(c),\ c \in \underline{\mathfrak{B}}(O \cup \{ov_q\},\ A,\ I \cup \{(ov_q, a) \mid a \in a_q|_K\})\}$

4.2 Relational queries

In this study, a relational query is composed of several simple queries, to which we add relational constraints. The relational constraints are expressed via virtual query objects, one for each formal context, where we want to find an object. A virtual query object may have relations (according to the relational contexts) with objects of other contexts, as well as with other virtual query objects.

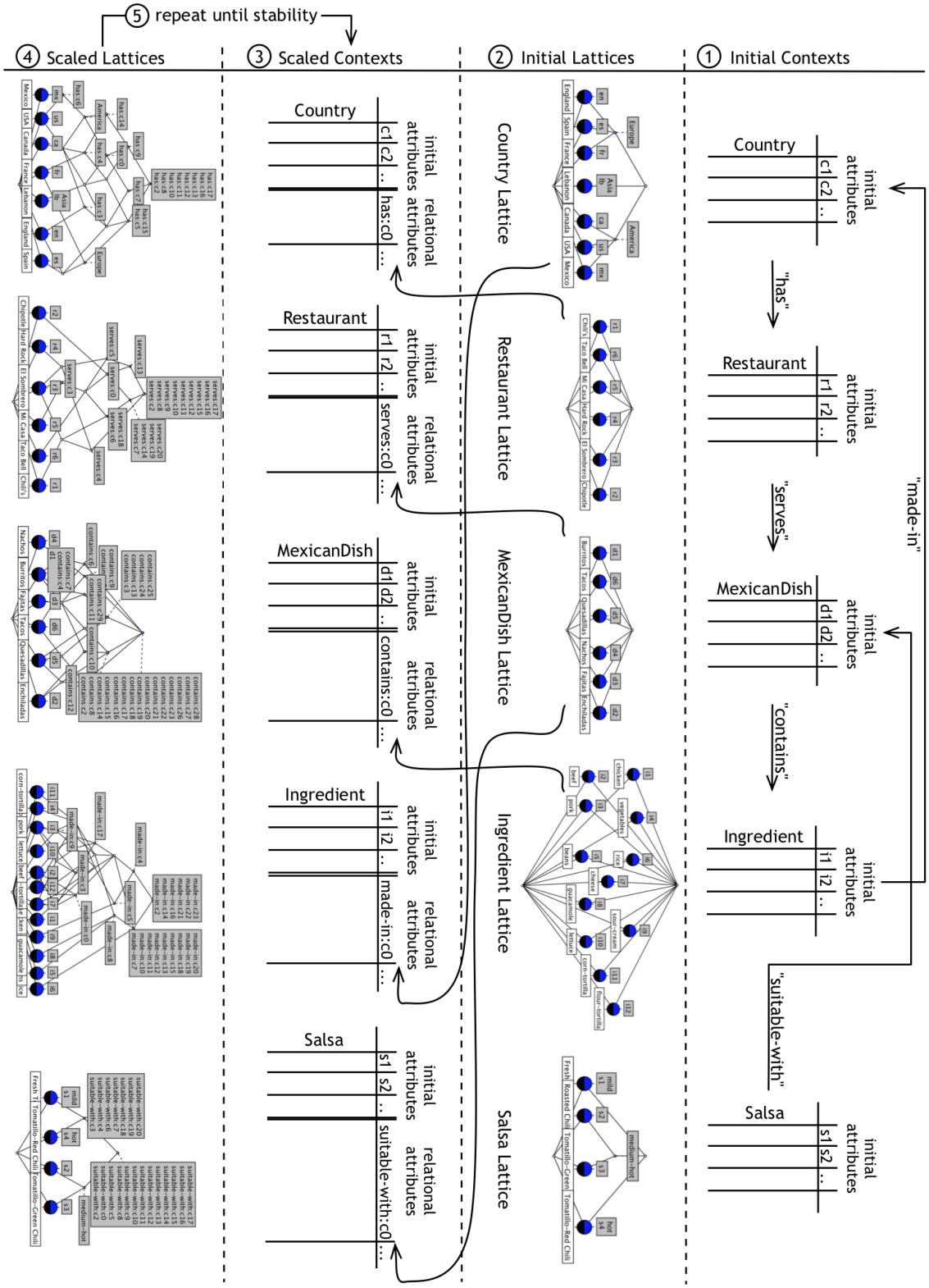


Figure 4.5: RCA Iterations.

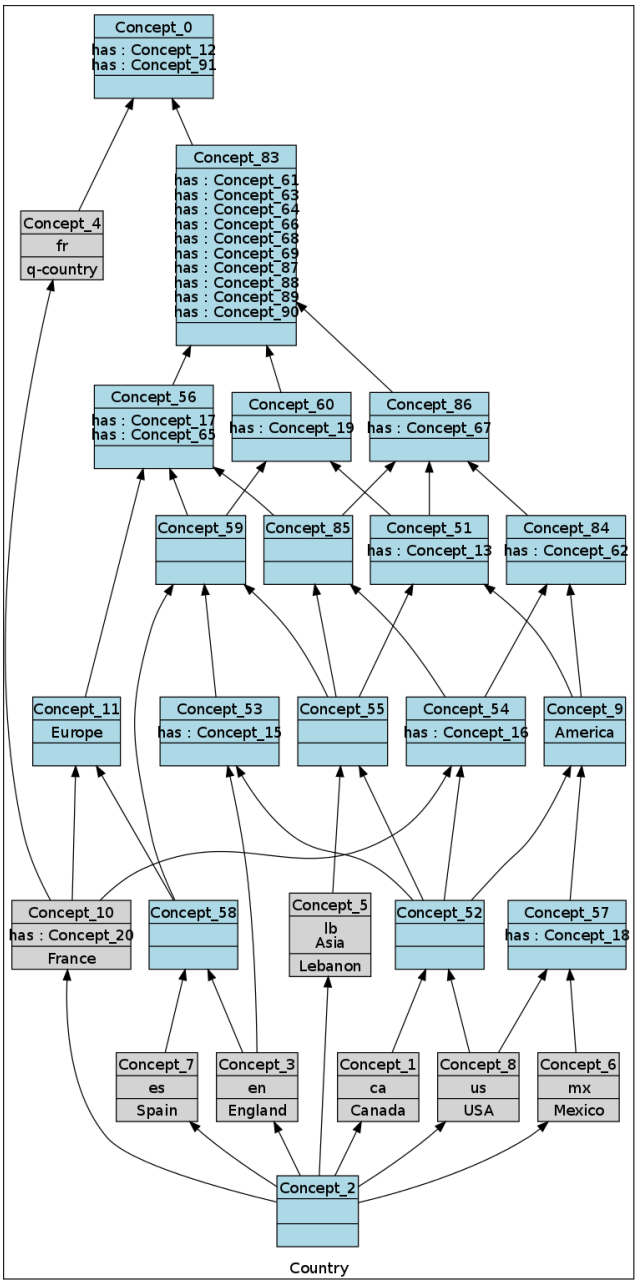


Figure 4.6: The country concept lattice.

Definition 14 *A relational query Q on a relational context family $(\mathbb{K}, R)$ is a pair $Q = (A_q, O_{vq}, R_q)$, where:*

1. *A_q is a set of simple queries, $A_q = \{q|_{K_i} = (o_q|_{K_i}, a_q|_{K_i}) \mid q|_{K_i}$ is a query on $K_i \in \mathbb{K}\}$*
2. *There is a one-to-one mapping between A_q and O_{vq}.*
3. *R_q is a set of relational constraints, $R_q = (o_{vq|_{K_i}}, r_{ij}, O_q)$, where $o_{vq|_{K_i}}$ is the virtual object associated with $q|_{K_i}$, $O_q \subseteq O_j \cup \{o_{vq|_{K_j}}\}$, with $o_{v_{q|_{K_j}}}$ is the virtual object associated with*

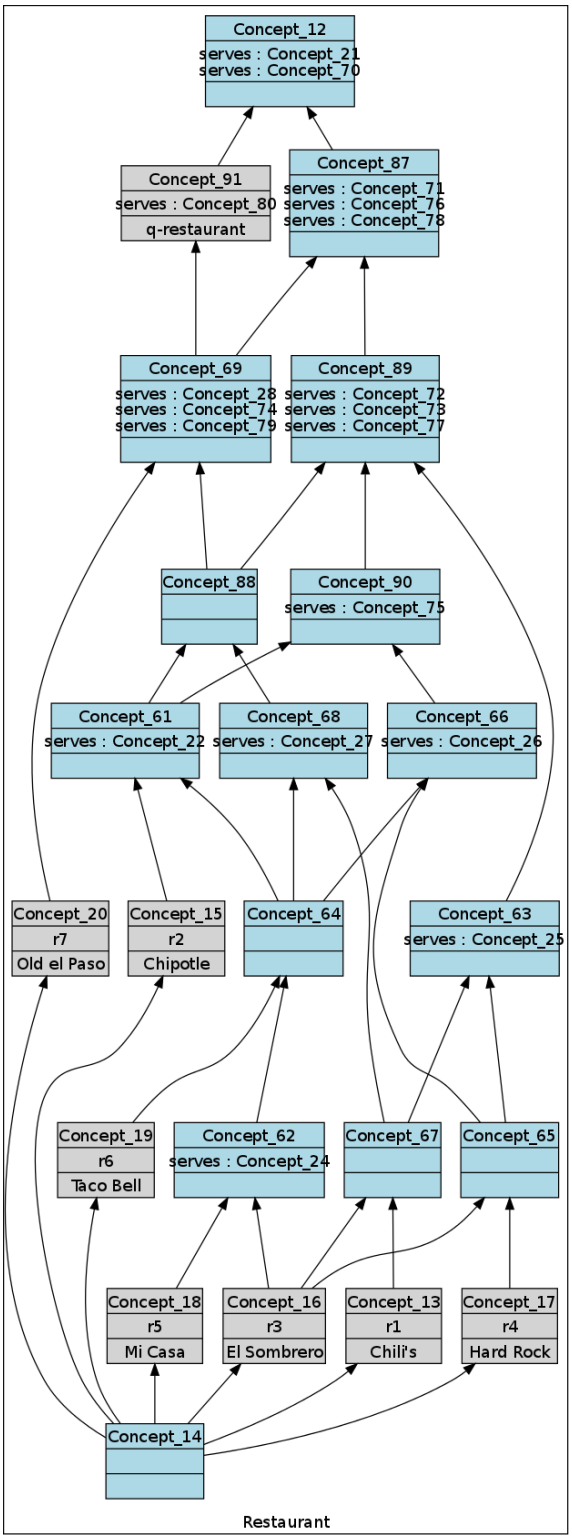


Figure 4.7: The restaurant concept lattice.

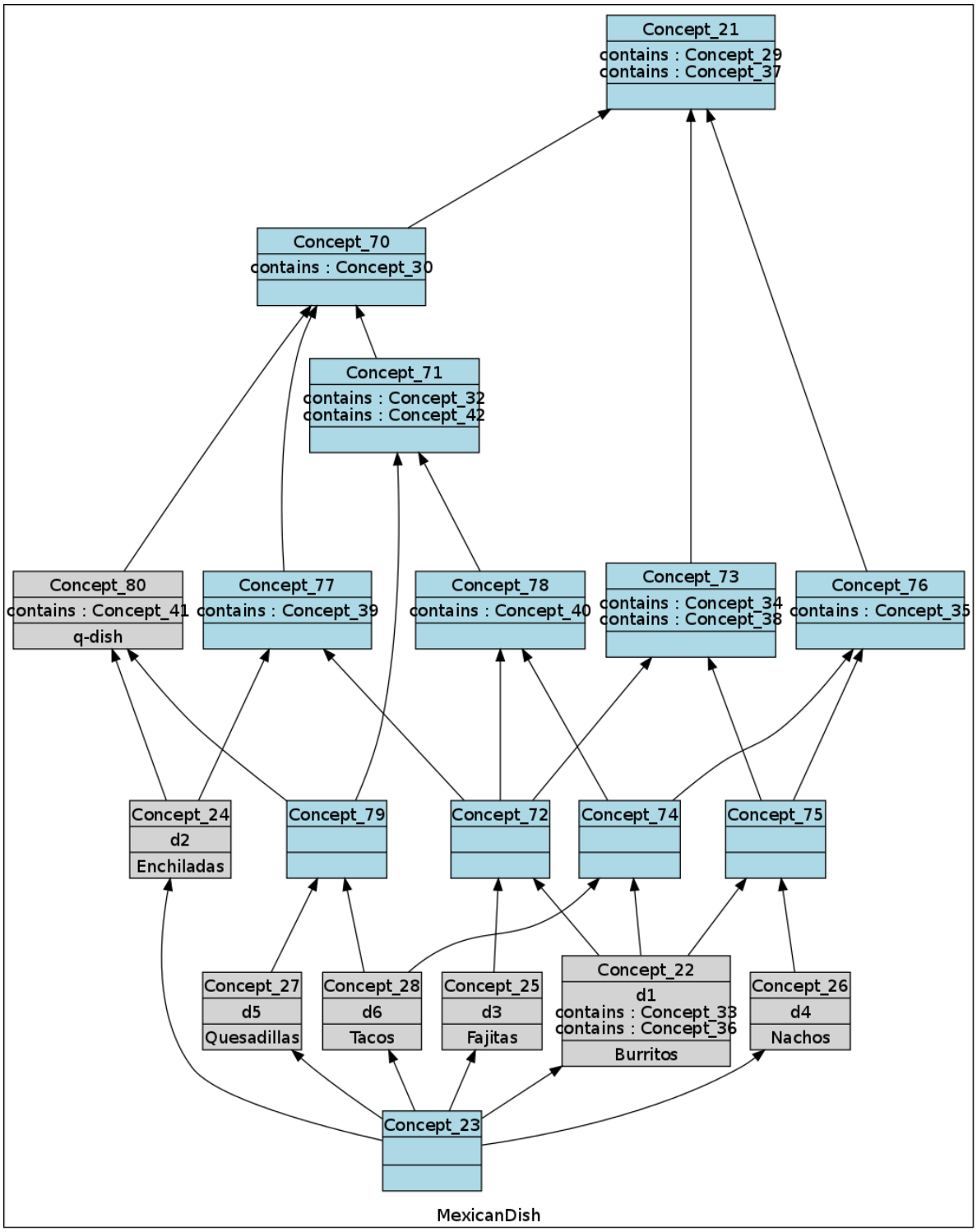


Figure 4.8: The mexican dish concept lattice.

K_j.

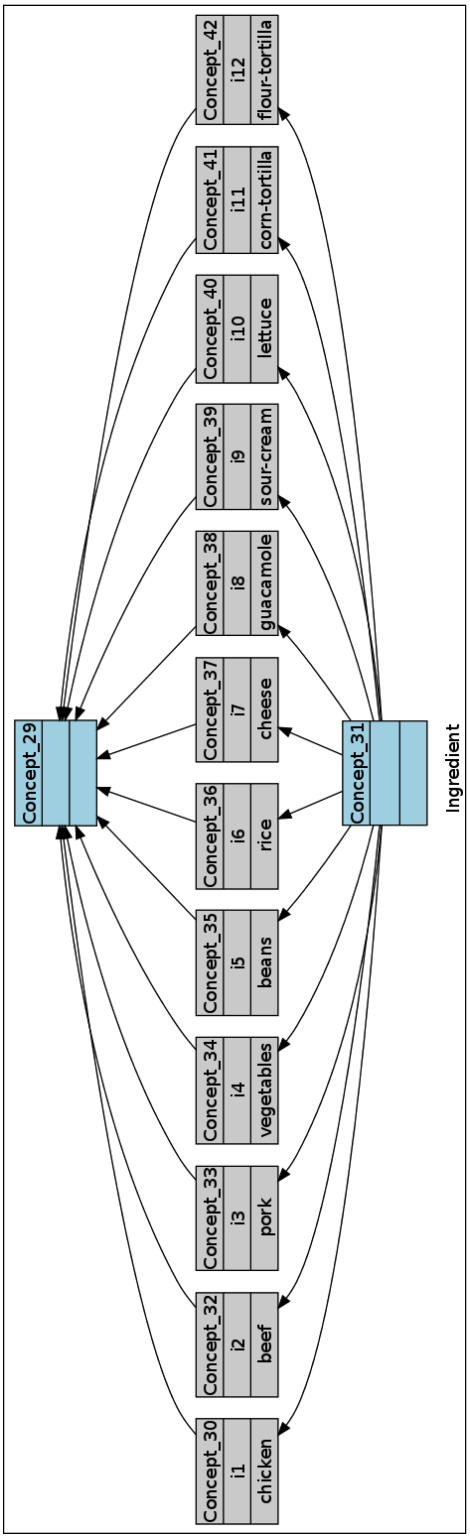


Figure 4.9: The ingredient concept lattice.

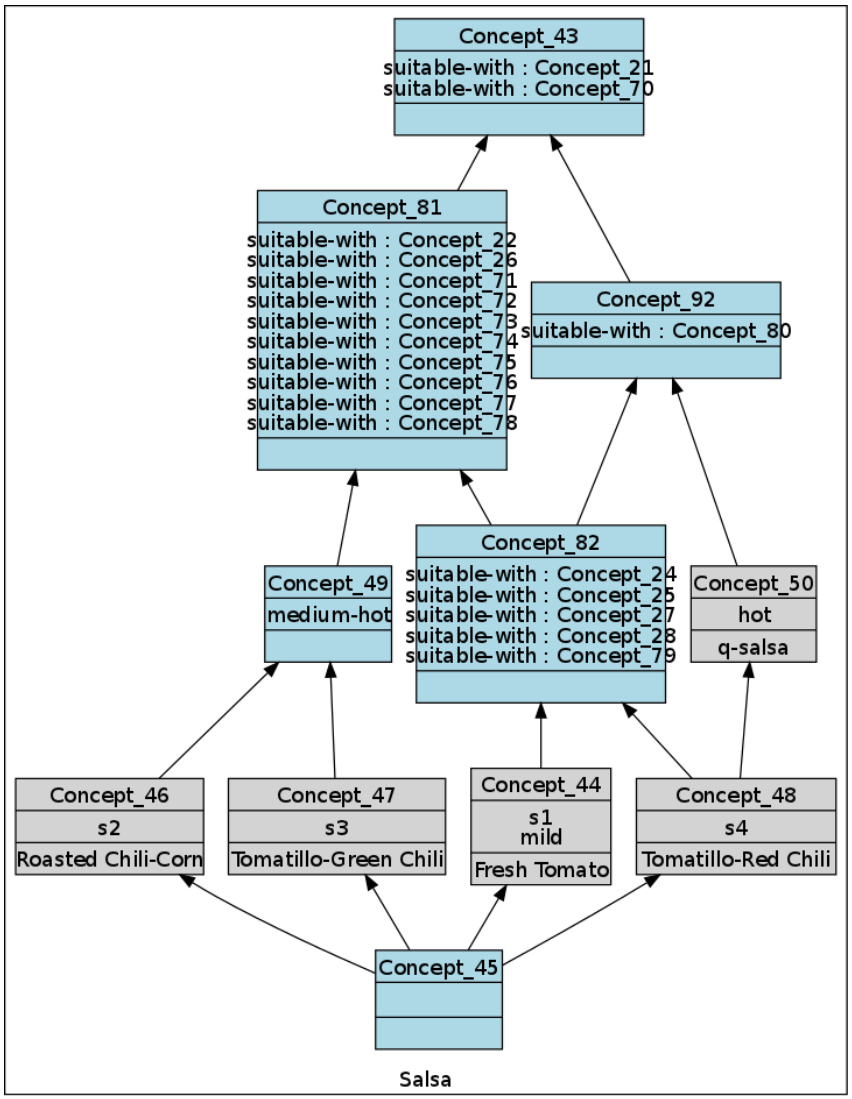


Figure 4.10: The salsa concept lattice.

For example, let us consider the following query: I am searching for a country which is described by the attribute "fr", a restaurant of this country that serves a Mexican dish containing (chicken, cheese, and corn-tortilla), and a salsa which is "hot" and suitable with the dish. This query can be translated into a relational query $Q_{example} = (A_q, O_{vq}, R_q)$ as follows:
$A_q = \{q_{country}, q_{restaurant}, q_{dish}, q_{salsa}\}$, $a_{q_{country}} = \{fr\}$, $a_{q_{restaurant}} = a_{q_{dish}} = \emptyset$, $a_{q_{salsa}} = \{hot\}$.
$O_{vq} = \{o_{vq_{dish}}, o_{vq_{country}}, o_{vq_{restaurant}}, o_{vq_{salsa}}\}$
$R_q = \{(o_{vq_{dish}}, contains, \{chicken, cheese, corn\text{-}tortilla\}), (o_{vq_{country}}, has, \{o_{v_{q_{restaurant}}}\}),$
$(o_{vq_{restaurant}}, serves, \{o_{v_{q_{dish}}}\}), (o_{vq_{salsa}}, suitable\text{-}with, \{o_{v_{q_{dish}}}\})\}$.
By definition, a query corresponds to the data model (Fig. 4.11), thus must respect the schema of the RCF.

A maximal answer to the relational query is composed of the answers of the simple queries. For our example, it would be $o_{q_{country}} = \{France\}$, $o_{q_{restaurants}}$ contains all the restaurants, $o_{q_{dish}}$ contains all the dishes, $o_{q_{salsa}} = \{Tomatillo - Red\ Chili\}$. If we consider these objects

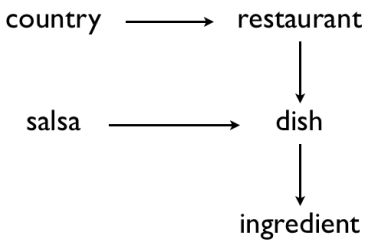


Figure 4.11: Query schema.

connected with the relations, this forms what we call the maximal answer graph. In this graph, we are interested in the subgraphs that cover the query (they have at least one object per element of A_q). These subgraphs are included in the graph of the Figure 4.12. There are various interesting forms of answer: having exactly one object per element of A_q, or having several objects per element of element of A_q.

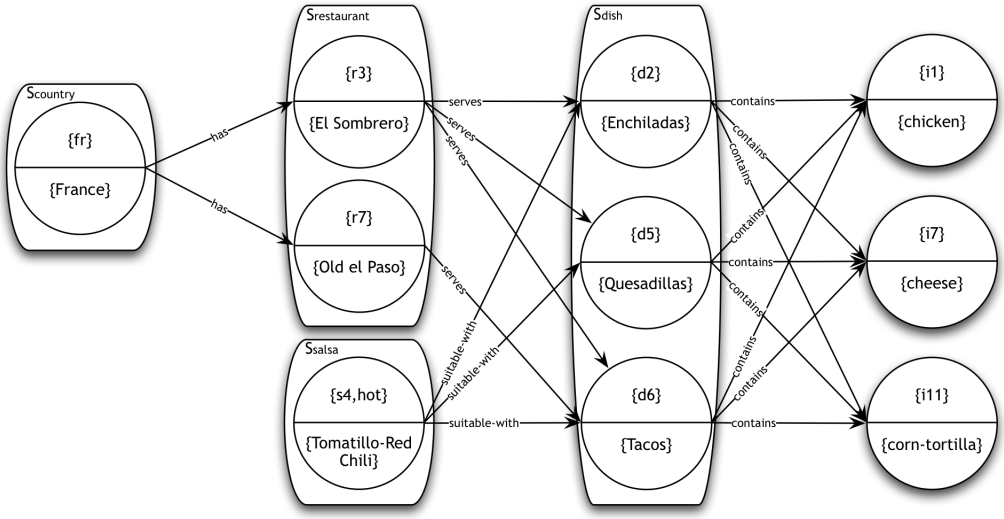


Figure 4.12: The subgraph containing interesting objects for the relational query example.

Definition 15 *An answer to a relational query $Q = (A_q, O_{vq}, R_q)$ is a set of objects X having a unique object per each context that is involved in the query:*

$$X = \{o_i \mid o_i \in O_i \text{ with } 1 \leq i \leq n\}$$

These objects satisfy the query $Q = (A_q, O_{vq}, R_q)$, when they have the requested attributes:
$\forall\ a_{q|_{K_i}} \in A_q,\ \exists\ o_i \in X\ :\ o_i' \supseteq a_{q|_{K_i}}$ *and they are connected as expected:*
$\forall\ (o_{vq|_{K_i}}, r, O_q) \in R_q$ *with* $r \subseteq O_i \times O_j$, *(and thus :* $O_q \subseteq O_j \cup \{o_{vq|_{K_j}}\}$*) and* $\forall\ o \in O_q$, *we have :*

1. *if* $o \in O_j$, *we have* $(o_i, o) \in r$
2. *if* $o = o_{v_{q|_{K_j}}}$, *we have* $(o_i, o_j) \in r$ *with* $o_j \in X \cap O_j$

For example, an answer to the query $Q_{example}$ can be the set:
$\{France, El\ Sombrero, Enchiladas, Tomatillo\text{-}RedChili\}$

For our example, the set of answers to the relational query, is:
$\{\{France, El\ Sombrero, Enchiladas, Tomatillo\text{-}Red\ Chili\}, \{France, El\ Sombrero, Quesadillas, Tomatillo\text{-}Red\ Chili\}, \{France, El\ Sombrero, Tacos, Tomatillo\text{-}Red\ Chili\}, \{France, Old\ el\ Paso, Tacos, Tomatillo\text{-}Red\ Chili\}\}$.

Answers can be provided with an aggregated form which can be found in lattices, as we explain below. They allow us to discover sets of equivalent objects relatively to the answer. E.g. $\{Enchiladas, Quesadillas, Tacos\}$ are equivalent objects if we choose $France$ and $El\ Sombrero$.

Definition 16 *An aggregated answer to a query $Q = (A_q, O_{vq}, R_q)$ is the set:*
$AR = \{S_i \mid$

- *there is a one-to-one mapping between AR and A_q which maps each $q|_{K_i}$ to a set S_i*
- $\forall\ q|_{K_i} \in A_q,\ \forall\ o_i \in S_i,\ o_i' \supseteq q|_{K_i}$ *(objects of S_i have the requested attributes)*
- *when* $(o_{v_{q|_{K_i}}}, r, O_q) \in R_q$
  - $if\ o_{vq|_{K_j}} \in O_q,\ r \subseteq O_i \times O_j,\ thus:\ \forall\ o_i \in S_i,\ \forall\ o_j \in S_j,\ S_j \in AR,$ *we have $(o_i, o_j) \in r$ (virtual objects are connected if requested)*
  - $for\ each\ o_j \in O_q \cap O_j\ we\ have:\ (o_i, o_j) \in r$ *(connections with particular objects are satisfied).*

$\}$

For example, (see Figure 4.12), an aggregated answer for our query is $France$ (having the attribute $\{fr\}$) that has the restaurant $\{ElSombrero\}$, which serves the dishes $\{Enchiladas, Quesadillas, Tacos\}$, which contain $\{chicken, cheese, corn\text{-}tortilla\}$, with which $\{Tomatillo\text{-}RedChili\}$ that is $\{hot\}$ is suitable.

$\{S_{country}, S_{restaurant}, S_{dish}, S_{salsa}\}$
$= \{\{France\}, \{ElSombrero\}, \{Enchiladas, Quesadillas, Tacos\}, \{Tomatillo\text{-}RedChili\}\}$

Another aggregated answer is $= \{\{France\}, \{OldelPaso\}, \{Tacos\}, \{Tomatillo\text{-}RedChili\}\}$

4.3 Lattice Navigation by Relational Queries

In this section, we explain how the navigation between the concept lattices can be guided by a relational query. Following relational attributes that lead us from one lattice to another, we navigate a graph whose nodes are the concept lattices. In a first subsection, we propose an algorithm which gives a general browsing schema that applies to concept lattices built with the existential scaling. Then we present several variations of this navigation algorithm.

4.4 A Query-Based Navigation Algorithm

Our approach for navigating the concept lattices along the relational attributes is based on the observations made during an experimental use of RCA, for finding the appropriate Web services to implement an abstract business process (as we shall see in Chapter 8). We consider an RCF and a query that respects the RCF relations. From our experience, we observed that an expert often expresses his query by a phrase, where chronology in introduction of principal verbs (relations) gives a natural path for the query flow. This will be our guiding hypothesis. Let us consider the query specified previously: *I am searching for a country, called "fr", that has a restaurant which serves dishes containing chicken, cheese and corn-tortilla; I am searching for a hot salsa suitable with this dish.* In order to simplify the notation, we use the same notation for queries $q|_{k_i}$ and the corresponding virtual objects $o_{vq|_{K_i}}$.

We can formalize the query path by a totally ordered set of arcs, and sometimes specific sets of objects. For our example, the path is a total order for $R_q = \{(q_{country}, has, \{q_{restaurant}\}), (q_{restaurant}, serves, \{q_{dish}\}), (q_{dish}, contains, \{chicken, cheese, corn\text{-}tortilla\}), (q_{salsa}, suitable\text{-}with, \{q_{dish}\})\}$. Each arc corresponds to a relation used in the query. All the relations involved inside a query are covered by this path. This translation of the expert query determines a composition on the relations. The query path does not always correspond to a directed chain in the object graph (*e. g.* dishes are the end of two of the considered relations (serves and suitable-with).

We propose the algorithms 1 to 4 for navigating through a concept lattice family using queries. During the exploration, we fill a set X by objects that will constitute an answer at the end (at most one object for each formal context). In this section, the algorithm is presented as an automatic procedure, its use to guide an expert in its manual exploration of the data is discussed afterwards.

Algorithm 1 identifies three main cases:

- line 2, the arc connects two query objects, *e.g.* $(q_{country}, has, \{q_{restaurant}\})$;
- line 5, the arc connects a query object to original objects *e.g.* (q_{dish}, *contains*, {*chicken*, *cheese*, *corn-tortilla*});
- line 8, the arc connects a query object to another query object and to original objects *e.g.* (q_{dish}, *contains*, $\{q_{ingredient}, chicken, cheese, corn\text{-}tortilla\}$).

Each of these cases considers, for a given arc a, whether the partial answer X already contains an object source or (inclusively) an object target.

When the arc connects a query object to another query object $a = (q|_{K_s}, r_{st}, q|_{K_t})$, (Algorithm 2), four cases are possible.

- X does not contain any object for K_s and any o_t for K_t: we identify the highest concept that owns the attributes of $q|_{K_s}$ and we select an object in its extent (lines 3-5). Then we continue on the next conditional statement (to find a target).

– X contains an object o_s for K_s and an object o_t for K_t selected in previous steps: we just check if o_s owns the relational attribute pointing at the object concept introducing o_t, that is γo_t (line 8)[1].

– X contains only an object o_s for K_s. We should find a target. We identify, under the highest concept that owns the attributes of $q|_{K_t}$, one of the lowest concepts to which o_s points (lines 12-14). We select a target in its extent.

– X contains only an object o_t for K_t. We should find a source. We identify the highest concept that owns the attributes of $q|_{K_s}$ and the relational attribute that points to o_t (lines 20-23). We select a source in its extent.

When the arc connects a query object to original objects $a = (q|_{K_s}, r_{st}, O_q)$ (Algorithm 3):

– Either X contains an object for K_s and we need to check if the relational attributes confirm that this object is connected to all the original objects in O_q) (line 4);

– Or we have to select an object for K_s, owning the attributes of the query $q|_{K_s}$ and owning the relational attributes ending in the concepts introducing the original objects (line 9-11).

Algorithm 4 is a composition of the two others cases. Note that whenever a condition is not verified, we have to backtrack, this is not specified in the algorithm for simplicity sake. If the query path forms also a directed chain in the entity-relationship diagram, the main algorithm is a depth-first search. But in the general case, in some steps, when we consider an arc, we filled X with an object for the end of the arc, and we need to find a source object.

For example, we start with the arc $(q_{country}, has, \{q_{restaurant}\})$ where the query path begins. We have to identify a source object o_s satisfying the query $\{fr\}$ (Definition 13). For example, we choose the object *France* appearing the extent of $Concept\_4$, whose intent contains *fr*.

We extract the relational attributes of $o_s = France$, they have the form $(\exists r_{st}\ C)$ and they are in pratice in the lattices denoted by $r : C$. For example, we obtain has:*Concept_ 19*, *has:Concept_ 15*, *has:Concept_ 60*, *has:Concept_ 16*, etc. We keep the relational attributes with the concepts satisfying the target query in the corresponding lattice and discard the rest. In our example, the $q_{restaurant}$ is empty. A relational attribute with the smallest concept (C_t) is the one to consider that leads us to find a solution. We choose *Concept_ 15* among the available smallest concepts. Let $\exists\ r_{st}\ C_t$ be the selected relational attribute (if it exists). The object o_t must be in the extent of C_t. In our example, we select *El Sombrero*.

Then we consider the query-to-query arc $(q_{restaurant}, serves, \{q_{dish}\})$. Given that an object is selected for $K_{restaurant}$, we look for a possible target object, lead by the query $q_{dish} = \emptyset$ and the relational attributes owned by the concept object $Concept\_15$ which introduces *El Sombrero*. Suppose we choose (line 13) a relational attribute that goes to one of the minimum concepts, namely $serves : Concept\_23$ (but $serves : Concept\_26$ or $serves : Concept\_25$ are also possible). This leads us to $Concept\_23$, in the extent of which we select *Enchiladas*.

[1] We remind that γo is the object concept introduced by o.

Dealing with the next arc $(q_{dish}, contains, \{chicken, cheese, corn\text{-}tortilla\})$ involves, since we have already selected a dish, to verify (Algorithm 3, line 4) that, the object concept $\gamma Enchiladas$ owns all the relational attributes that go to object concepts introducing chicken, cheese, and corn-tortilla. These are $contains : \gamma\ chicken = Concept\_29$, $contains : \gamma\ cheese = Concept\_36$ and $contains : \gamma\ corn - tortilla = Concept\_40$ and they are indeed inherited by $\gamma Enchiladas = Concept\_23$.

When the arc $(q_{salsa}, suitable\text{-}with, \{q_{dish}\})$ is considered, the target (Enchiladas) is in X. Thus we identify a source in the extent of the $Concept\_47$ which satisfies the target query $\{hot\}$. Its intent contains $suitable - with : Concept\_23$ which is $Enchiladas$. A target object (Tomatillo-Red Chili) is selected in the extent of $Concept\_47$. The answer is now complete.

The process is different if we use a different path and even with the considered path it may fails. For example, when dealing with the edge $(q_{restaurant}, serves, \{q_{dish}\})$, if we choose $serves : Concept\_25$, we get $Nachos$ in the extent of $Concept\_25$. For the next arc $(q_{dish}$, $contains$, $\{chicken$, $cheese$, $corn\text{-}tortilla\})$, it appears that $Concept\_25$ does not inherit $contains : \gamma\ corn - tortilla = Concept\_40$ and we need to backtrack to the choice of a dish.

Algorithm 1 Navigate(RCF, Q, P_Q) $//P_Q = (a_k) \mid a_k = r_{ij}$ *and* $r_{ij} \in R_Q$

Data: $(\mathbb{K}, R)$: an RCF; $Q = (A_q, O_{vq}, R_q)$: a query on $(\mathbb{K}, R)$; and a query path
Result: X: an object set (answer for Q) or fail

```
foreach arc a ∈ P_Q do                                              1
    if a = (q|_{K_s}, r_st, q|_{K_t}) then                          2
        Case_pure_query                                             3
    else                                                            4
        if a = (q|_{K_s}, r_st, O_q) with O_q ⊆ O_t then             5
            Case_pure_objects                                       6
        else                                                        7
            if a = (q|_{K_s}, r_st, q|_{K_t}) with q|_{K_t} ∈ O_q then   8
                Case_query_and_objects                              9
```

4.5 Variations about the Algorithm

Integrating queries into the contexts One approach that has been investigated in the case of simple queries consists to integrate the virtual query object in the context, then to build the concept lattice. This can also be done for relational queries. A relational query $Q = (A_q, O_{vq}, R_q)$ can be integrated into an RCF by adding the virtual query objects $o_{v_{q|_{K_i}}}$ into the context K_i. Each virtual query object $o_{vq|_{K_i}}$ owns the attributes of the query $a_{q|_{K_i}}$ and for each arc $(o_{v_{q|_{K_i}}}, r_{ij}, o_{v_{q|_{K_j}}})$, the relational context of r_{ij} is enriched by a line for $o_{v_{q|_{K_i}}}$, a column for $o_{v_{q|_{K_j}}}$ and the relation $(o_{v_{q|_{K_i}}}, o_{v_{q|_{K_j}}})$.

We integrated the relational query into our Mexican Food RCF. The formal contexts with queries are in Table 4.5, and relational contexts with queries are in Table 4.6.

Algorithm 2 Case_pure_query

Let $a = (q|_{K_s}, r_{st}, q|_{K_t})$ 1
if *// X does not contain a source and a target for the current arc a* 2
$X \cap O_s = \emptyset$ *and* $X \cap O_t = \emptyset$ **then**
  *// **select** a source in the extent of a concept that verifies the source query*
  Let C_s be the highest concept having Intent $(C_s) \supseteq q|_{K_s}$ 3
  select $o_s \in Extent(C_s)$ 4
  $X \leftarrow X \cup \{o_s\}$ 5
if *// X contains a source and a target for the current arc a* 6
$X \cap O_s = \{o_s\}$ *and* $X \cap O_t = \{o_t\}$ **then** 7
  // verify that the source is connected to the target
  check $\exists r_{st}\ \gamma_{o_t} \in Intent(\gamma_{o_s})$ 8
else
  if *// X contains a source for the current arc a* 10
  $X \cap O_s = \{o_s\}$ **then**
    //select a target in the extent of a concept that verifies the target query and is connected 11
    to the source
    Let C_t be the highest concept having Intent $(C_t) \supseteq q|_{K_t}$ 12
    and $C_t \in \min(C \mid \exists\ (\exists r_{st}\ C) \in Intent(\gamma o_s))$ 13
    select $o_t \in Extent(C_t)$ 14
    $X \leftarrow X \cup \{o_t\}$ 15
  else
    // X contains a target for the current arc a 17
    // select a source in the extent of a concept that verifies the source query 18
    and is connected to the target 19
    Let $o_t \in X \cap O_t$ 20
    Let C_s be the highest concept having Intent $(C_s) \supseteq q|_{K_s}$ 21
    and $\exists r_{st}\ \gamma o_t\ \in Intent(C_s)$ 22
    select $o_s \in Extent(C_s)$ 23
    $X \leftarrow X \cup \{o_s\}$ 24

Algorithm 3 Case_pure_objects

Let $a = (q|_{K_s}, r_{st}, O_q)$ with $O_q \subseteq O_t$ 1
if *// X contains a source for the current arc a* 2
$X \cap O_s = \{o_s\}$ **then**
  // verify that the source is connected to the objects in O_q 3
  check $\forall o \in O_q, \exists r_{st}\ \gamma_o \in Intent(\gamma_{o_s})$ 4
else
  // X does not contain a possible source 6
  // select a source in the extent of a concept that verifies the source query 7
  // and is connected to the target objects 8
  Let C_s be the highest concept having Intent $(C_s) \supseteq q|_{K_s}$ 9
  and $\forall o \in O_q, \exists r_{st}\ \gamma_o \in Intent(C_s)$ 10
  select $o_s \in Extent(C_s)$ 11
  $X \leftarrow X \cup \{o_s\}$ 12

We generate the corresponding concept lattice family, considering the existential scaling. The set of lattices are shown in Figures 4.13, 4.14, 4.15, 4.16, 4.17.

Locating the highest concept that introduces all the attributes of each query of each concerned context, now is much more easy because it introduces the virtual query object. Then, we can navigate in a similar way as before.

Variations of Navigation According to Scaling Operators The scaling operators of Relational Concept Analysis offer various other opportunities to browse the data. In our web service application, we used the scaling operator *covers* which captures the following information. If an object o_s is connected to a set of objects T, then in the scaled relation, o_s will be connected to the concepts whose extent is included in T. This is used to form for example a group of restaurants, which serve all the dishes containing sour cream, by contrast to the existential scaling operator which rather forms a group of restaurants, which serve at least one of the dishes containing sour cream.

With the covers scaling (Definition 10), we know that an object belonging to some concepts is connected to all the objects of the target concept according to the relational attribute. Thanks to this property, it is easier to find aggregate answers in the concept lattice family because all the objects in a concept have a same set of properties (attributes and connections)[2]. It is clear that on large data, it is not practical to build many different concept lattice families using different operators. But the approach can be valuable on a small subset of the data as it was our case after filtering the huge set of web services by quality and functionality requirements.

[2]For reader interested to see the lattices: http://www.lirmm.fr/~huchard/RCA_queries/mexicoCoversWithoutQuery.rcft.svg

Algorithm 4 Case_query_and_objects

Let $a = (q|_{K_s}, r_{st}, O_q)$ with $q|_{K_t} \in O_q$ 1

if *// X does not contain a source and a target for the current arc a* 2
$X \cap O_s = \emptyset$ *and* $X \cap O_t = \emptyset$ **then**
  // select a source in the extent of a concept that verifies the source query // and is connected 3
  to the objects of O_q
  Let C_s be the highest concept having Intent $(C_s) \supseteq q|_{K_s}$ 4
  and $\forall o \in O_q, \exists r_{st}\ \gamma_o \in Intent(C_s)$ 5
  select $o_s \in Extent(C_s)$ 6
  $X \leftarrow X \cup \{o_s\}$ 7

if *// X contains a source and a target for the current arc a* 8
$X \cap O_s = \{o_s\}$ *and* $X \cap O_t = \{o_t\}$ **then**
  // verify that the source is connected to the target and to all objects of O_q 9
  check $\exists r_{st}\ \gamma_{o_t} \in Intent(\gamma_{o_s})$ 10
  and $\forall o \in O_q, \exists r_{st}\ \gamma_o \in Intent(\gamma o_s)$ 11
else
  if *// X contains a source for the current arc a*
  $X \cap O_s = \{o_s\}$ **then**
    // verify that the source is connected to all objects of O_q 12
    check $\forall o \in O_q, \exists r_{st} :\ \gamma_o \in Intent(\gamma o_s)$ 13
    // and select a target in the extent of a concept that verifies the target query and is 14
    connected to the source
    Let C_t be the highest concept having Intent $(C_t) \supseteq q|_{K_t}$ 15
    and $C_t \in \min(C \mid \exists\ (\exists r_{st}\ C) \in Intent(\gamma o_s))$ 16
    select $o_t \in Extent(C_t)$ 17
    $X \leftarrow X \cup \{o_t\}$ 18
  else
    // X contains a target for the current arc a 19
    // select a source in the extent of a concept that verifies the source query 20
    and is connected to the target and to the objects in O_q 21
    Let $o_t \in X \cap O_t$ 22
    Let C_s be the highest concept having Intent $(C_s) \supseteq q|_{K_s}$ 23
    and $\exists r_{st}\ \gamma o_t\ \in Intent(C_s)$ 24
    and $\forall o \in O_q, \exists r_{st}\ \gamma_o \in Intent(C_s)$ **select** $o_s \in Extent(C_s)$ 25
    $X \leftarrow X \cup \{o_s\}$ 27

Table 4.5: The formal contexts of the Mexican Food RCF with the integrated relational query.

	ca	en	fr	lb	mx	es	us	America	Asia	Europe	
Canada	×							×			
England		×								×	
France			×							×	
Lebanon				×					×		
Mexico					×			×			
Spain						×				×	
USA							×	×			
q-country			×								

	r1	r2	r3	r4	r5	r6	r7
Chili's	×						
Chipotle		×					
El Sombrero			×				
Hard Rock				×			
Mi Casa					×		
Taco Bell						×	
Old el Paso							×
q-restaurant							

	d1	d2	d3	d4	d5	d6	
Burritos	×						
Enchiladas		×					
Fajitas			×				
Nachos				×			
Quesadillas					×		
Tacos						×	
q-dish							

	i1	i2	i3	i4	i5	i6	i7	i8	i9	i10	i11	i12	
chicken	×												
beef		×											
pork			×										
vegetables				×									
beans					×								
rice						×							
cheese							×						
guacamole								×					
sour-cream									×				
lettuce										×			
corn-tortilla											×		
flour-tortilla												×	

	s1	s2	s3	s4	mild	medium-hot	hot	
Fresh Tomato	×				×			
Roasted Chili-Corn		×				×		
Tomatillo-Green Chili			×			×		
Tomatillo-Red Chili				×			×	
q-salsa							×	

Opportunities of browsing offered by the exploration As we explained before, the algorithm described in the previous section can be understood as an automatic procedure to determine a solution to a query. Nevertheless, it is more interesting to use it as a guiding method for the exploration of data by a human expert. Each object selection is a departure point for inspecting the objects of the selected concept, and, explore the neighborhood, going up by relaxing constraints or going down by adding constraints.

Table 4.6: The relational contexts of the Mexican Food RCF with the integrated relational query.

	chicken	beef	pork	vegetables	beans	rice	cheese	guacamole	sour-cream	lettuce	corn-tortilla	flour-tortilla
Burritos	×	×	×	×	×	×	×	×	×	×		×
Enchiladas	×						×		×		×	
Fajitas	×	×		×			×	×	×	×		×
Nachos				×	×		×	×				
Quesadillas	×	×					×				×	×
Tacos	×	×			×		×			×	×	×
q-dish	×						×				×	

	Chili's	Chipotle	El Sombrero	Hard Rock	Mi Casa	Taco Bell	Old el Paso	q-restaurant
Canada	×	×	×	×		×		
England		×		×		×		
France			×	×			×	
Lebanon	×			×		×		
Mexico	×				×	×		
Spain				×		×		
USA	×	×	×	×	×	×		
q-country								×

	Burritos	Enchiladas	Fajitas	Nachos	Quesadillas	Tacos	q-dish
Chili's			×		×	×	
Chipotle	×					×	
El Sombrero	×	×	×	×	×	×	
Hard Rock			×	×			
Mi Casa	×	×		×	×	×	
Taco Bell	×			×	×	×	
Old el Paso						×	
q-restaurant							×

	Burritos	Enchiladas	Fajitas	Nachos	Quesadillas	Tacos	q-dish
Fresh Tomato	×	×	×	×	×	×	
Roasted Chili-Corn	×			×			
Tomatillo-Green Chili	×			×			
Tomatillo-Red Chili	×	×	×	×	×	×	
q-salsa							×

A point in favor of the lattices is that they not only give us a solution: besides they classify the objects of the solutions and provide a navigation structure. They also give other information about the objects which can be useful for the expert: attributes that objects of the answer set have necessarily, attributes that appear in the same time as attributes of the answer, etc.

In our web service application, we preferred the solution which integrates the query in the RCF because we found easier to identify the answers. The lattices show how the existing objects match and differ from the query, thanks to the factorization of attributes between the query and the existing objects. Nevertheless, having several queries at the same time would not be efficient. Thus the solution has be used only for specific problems. An incremental algorithm can be used to introduce the query, which enlightens the process of modifying the lattice and gives information to the expert about the structure of data. We can conserve the original lattice (before query integration), and save the query objects together with the resulting concepts in an auxiliary structure. This way, we can always easily go back to the original lattices.

5 Summary

In this chapter, we gave a comprehensive background about the Formal and Relational Concept Analysis (FCA, RCA), together with their formal definitions illustrated by examples. We also defined the notions of queries and relational queries, as a mechanism for lattice navigation. These two classification techniques are used in our framework for solving the problems surrounding Web service classification and selection. We point that there are several algorithms for FCA and RCA. The complexity of these algorithms varies according to the input contexts and their density [67, 68].

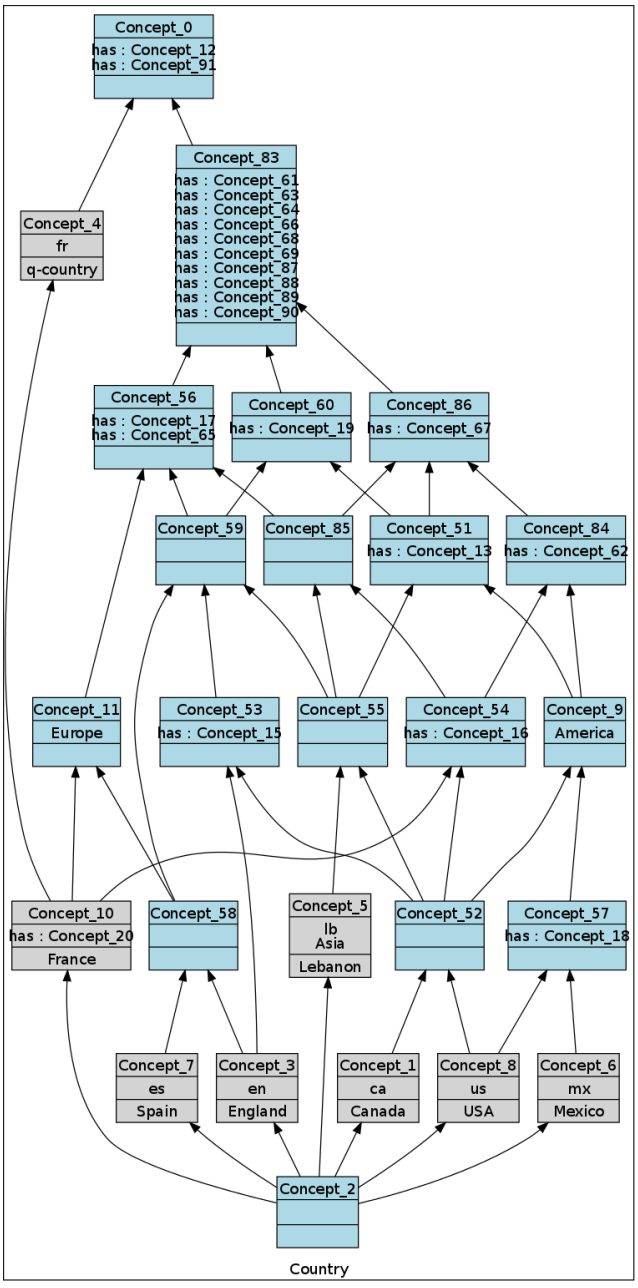


Figure 4.13: The country concept lattice with query q-country.

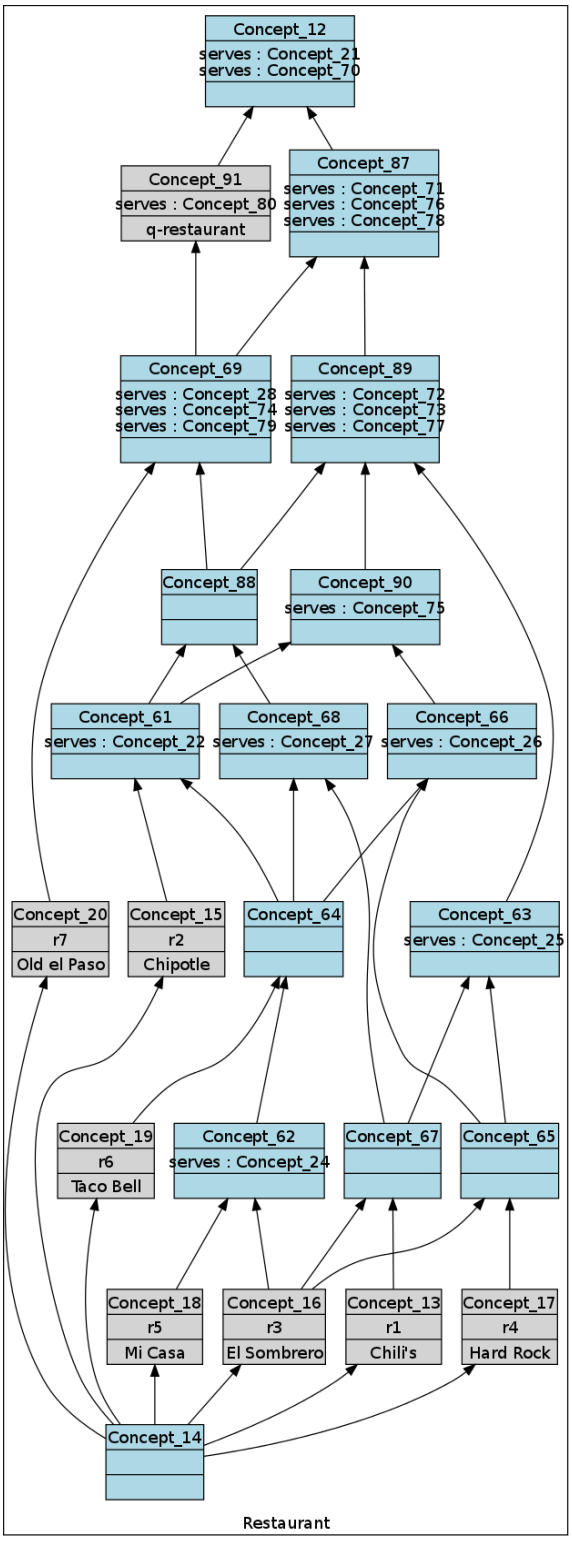


Figure 4.14: The restaurant concept lattice with query q-restaurant.

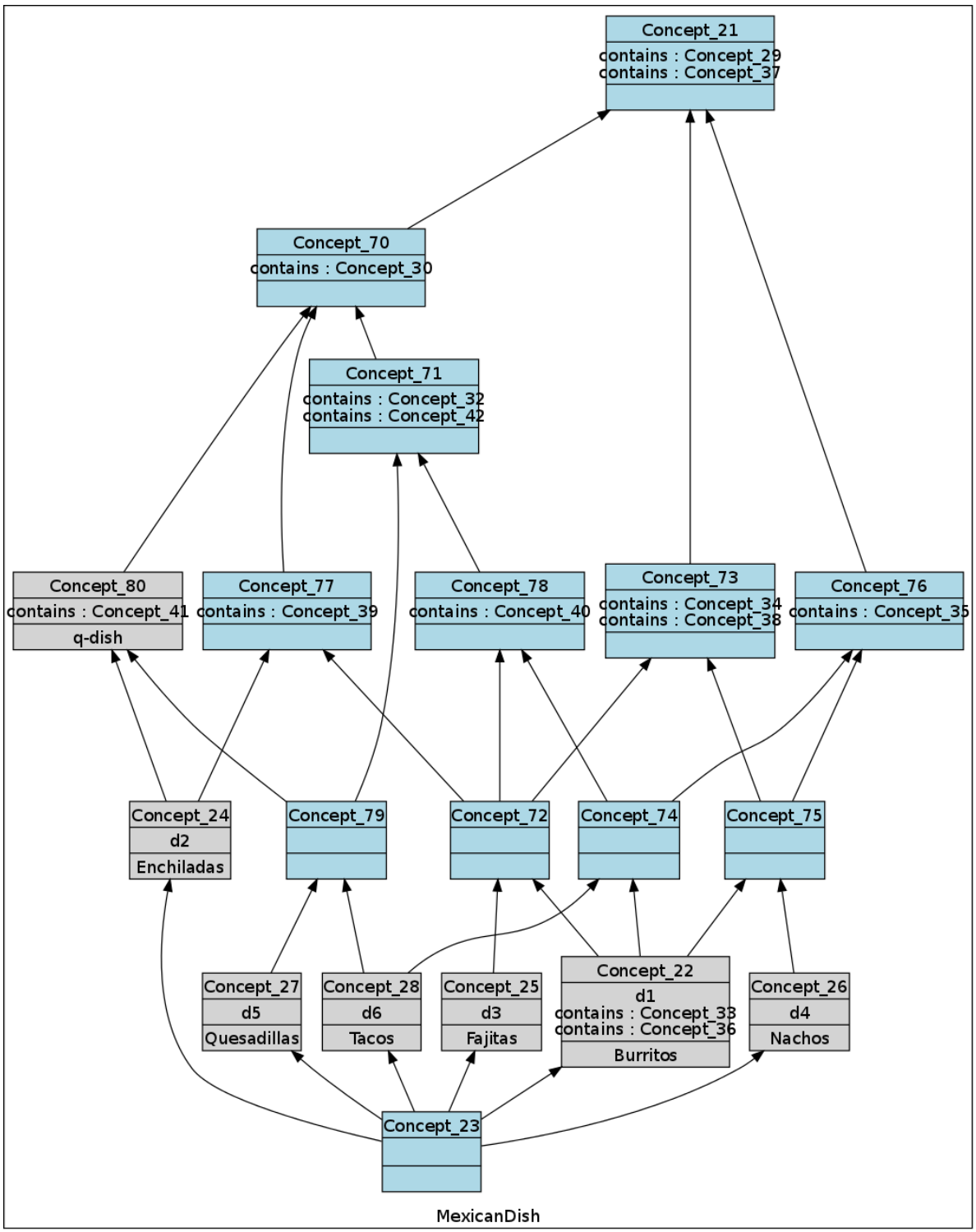


Figure 4.15: The mexican dish concept lattice with query q-dish.

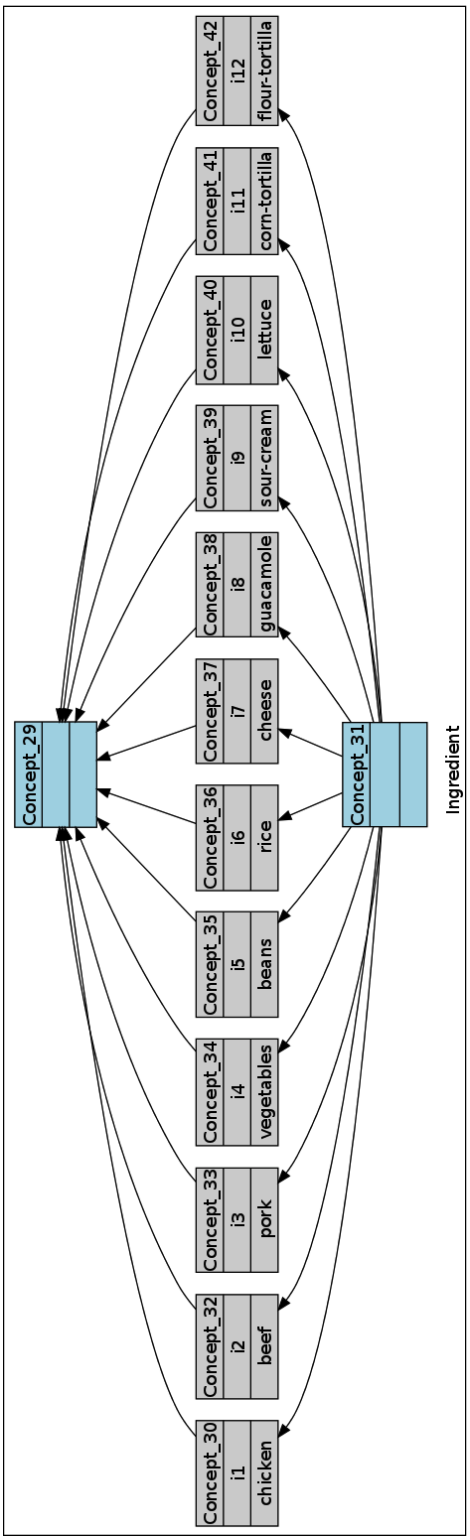


Figure 4.16: The ingredient concept lattice.

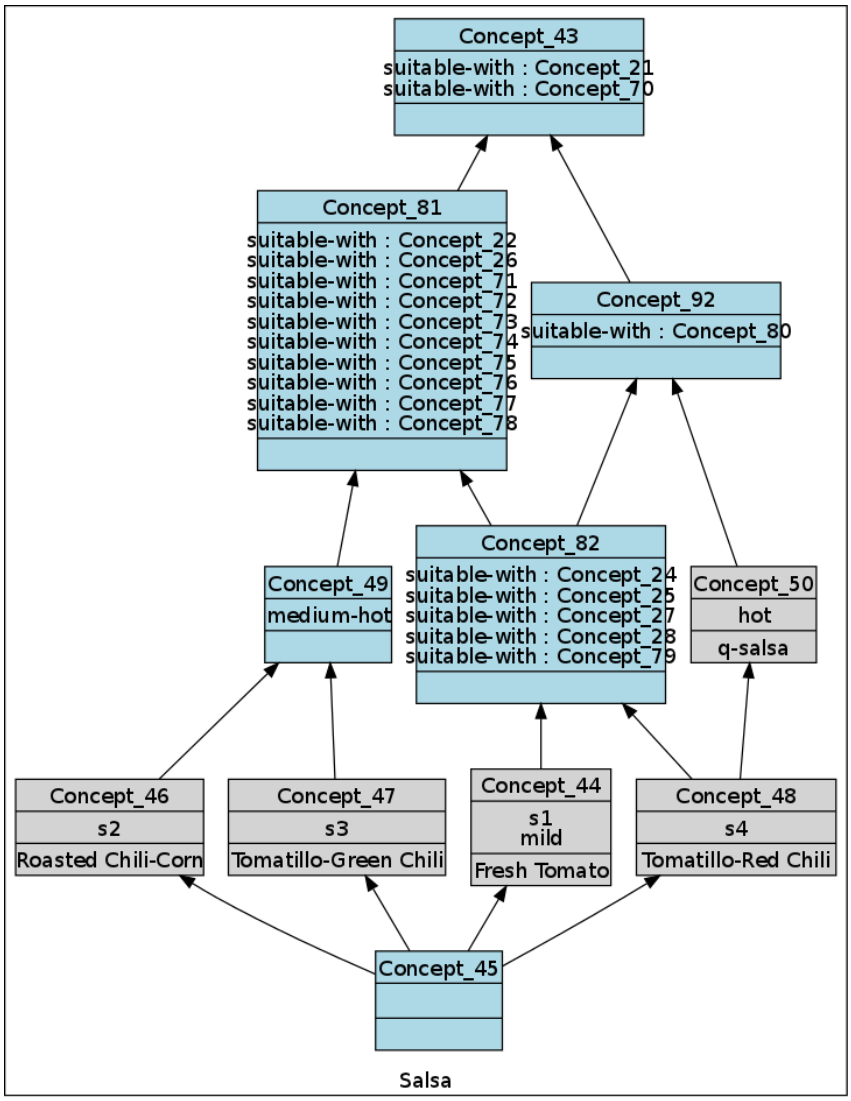


Figure 4.17: The salsa concept lattice with query q-salsa.

Part II

Contributions

Chapter 5

The Selection Framework

Contents

1 Overview

Building a business process requires a multi criteria-based service selection model. This selection model must enable a user to efficiently identify services that satisfy the required functionality, by being compatible and composable, as well as having the expected QoS values.

In this thesis, we propose a framework for achieving such a selection model based on both functional and non-functional properties. It enables a user to express his functional and non-functional requirements for building his desired business process. It works on analysing these requirements, discovering candidate services and classifying them (see Figure 5.1). This enables an efficient selection of the needed services as well as their potential substitutes. In the following, we present our framework in more details. We describe its various layers, and we explain the components involved in each of them.

2 The Selection Framework

Our framework assists users to select Web services efficiently during design-time, in order to achieve their desired Web service-based applications. We illustrate our framework in Figure 5.2, in which we see a general overview that can be specialized according to the needed use case.

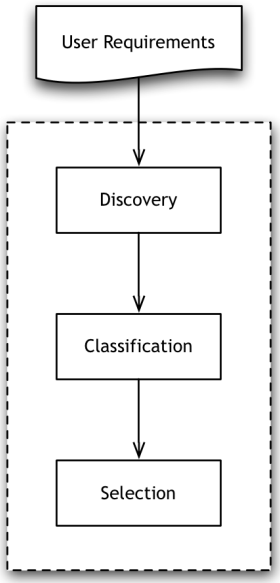


Figure 5.1: The framework's main layers.

This framework consists of four main layers: user requirements, discovery, classification, and selection. In this chapter, we focus on describing in details the user requirements layer and the discovery layer. The remaining layers are described briefly here, and are further detailed in the following chapters (according to some considered use cases).

2.1 User Requirements Layer

In the user requirements layer, a user is able to express his requirements for Web services according to the needed functional as well as non-functional properties. According to the desired service-based application, a functionality can be either an individual service providing a needed operation, or several services that can be composed together. The requirements can be expressed by specifying an abstract WSDL (AWSDL) that describes the needed services. This AWSDL file can be also used to define an abstract BPEL (ABPEL). The ABPEL describes the desired business process by specifying the interactions between the services described in the AWSDL file. These two files are analyzed in the discovery layer by the analyzer component, as we shall see shortly.

2.1.1 Abstract WSDL File

A user is allowed to define an abstract WSDL interface, inside which he describes all of the needed services by specifying functional and non-functional requirements. A needed service is characterised by the needed operation(s) that it provides, together with the expected QoS levels for every supported attribute.

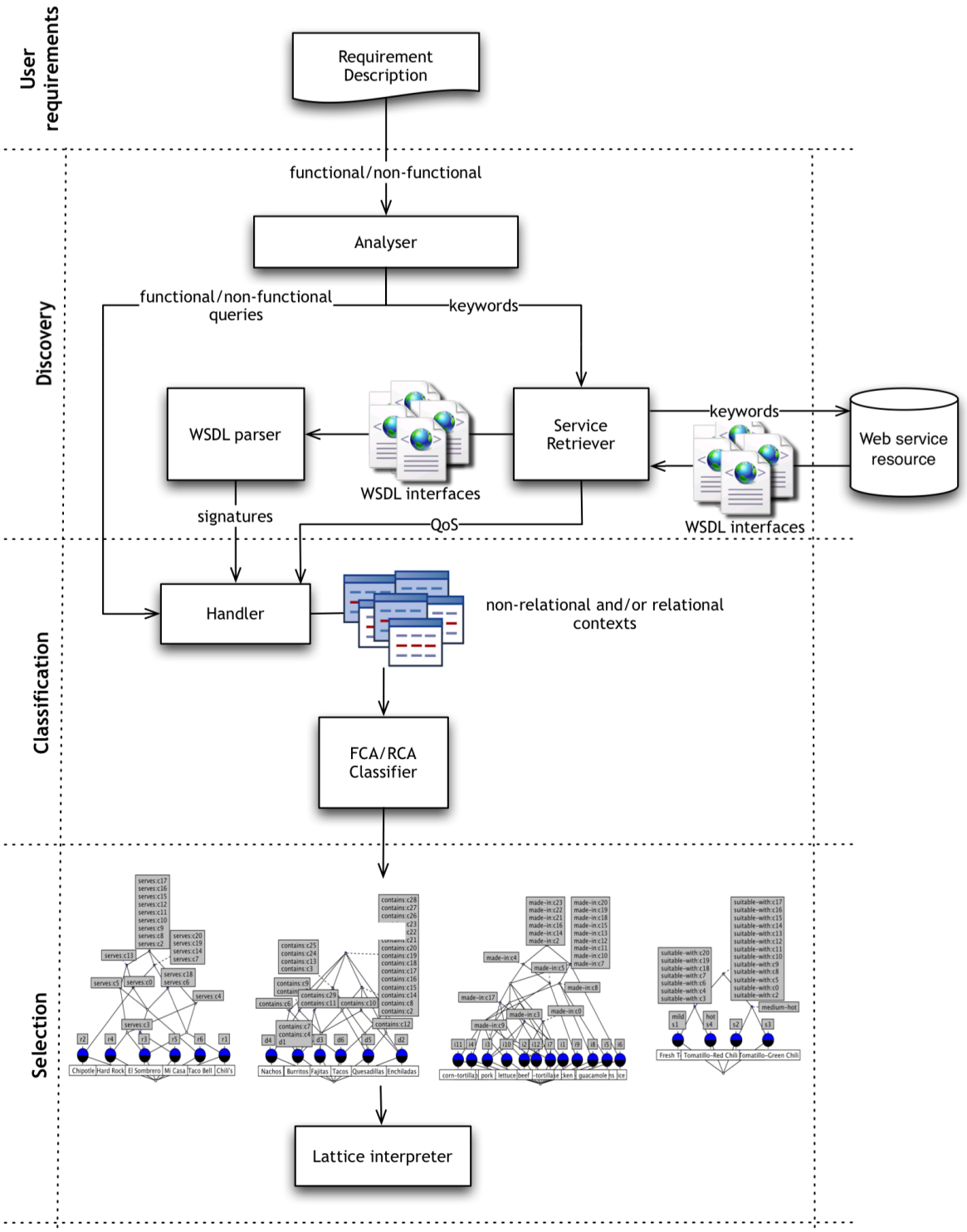


Figure 5.2: A general overview of the framework and its layers.

An abstract WSDL interface contains information about one or more needed services, together with the accepted QoS levels. Services are described by their operations together with their input/output parameters names and types. The functional and non-functional requirements are described by enriching the documentation tags of the WSDL description, as illustrated in Figure 5.3. An abstract WSDL does not have a concrete part, which usually provides information about the service's location and the supported protocol for invocation.

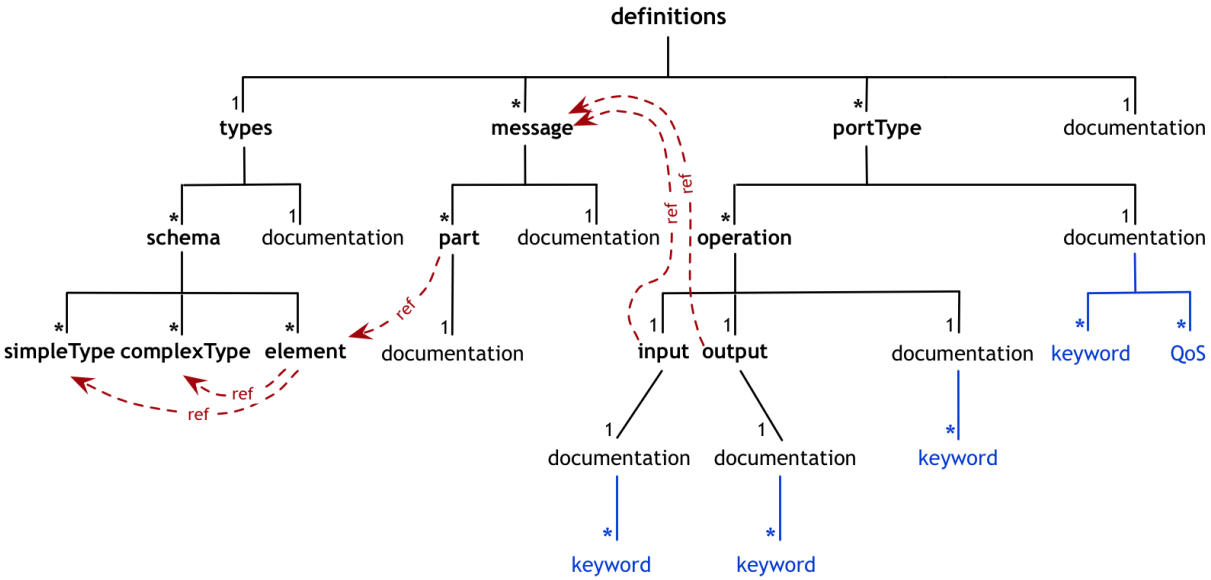


Figure 5.3: Abstract WSDL interface.

We consider that a needed operation can be characterized by its input/output parameters and their types. Thus, when defining an operation inside an abstract WSDL, a user is supposed to provide for each input/output parameter a set of relevant keywords for its name, and another set for its type. For example, for a parameter named *city* the user may provide $\{city, town, metropolis\}$. This can be expressed using the documentation tags in a WSDL interface, which can be defined for each WSDL element such as service, operation, message, part, type, etc. Like this, it is not necessary to extend the WSDL standards with new elements to fulfil our needs.

We suppose also that the user must not provide a specific operation name, he can provide any operation identifier, like for example $op1$, which will be discarded when parsed. We argue that an operation's name can be as much a good indicator for the operation's functionality as it can be bad. For example, if someone is searching for an operation that takes as input a country name and returns its capital, an operation named getCapital or getCapitalForCountry would be exactly the required one. On the other hand, an operation named getCountryInformation would be misleading. It may even get discarded if we consider its name, although it takes a country name as input and returns the capital as one of its outputs. Let us consider another example of an operation for weather information called getWeather that takes a city name and returns information about temperature, humidity, wind speed, etc. If someone is searching for

an operation called getTemperature, it is for sure that he will miss the getWeather operation, although it provides the functionality he is asking for.

2.1.2 Abstract BPEL File

An abstract BPEL in literature specifies the external message exchange between Web services and does not contain any internal details of the business process, as we mentioned previously in Chapter 4.

We define the concept of an abstract BPEL as a normal concrete BPEL, which is based on a locally defined abstract WSDL, instead of services retrieved from the Web. In this way, a user aiming at building a business process does not have to be aware of the existing services on the Web. Nevertheless, he should have enough knowledge about dividing the functionality of the desired process into linked smaller pieces that can be represented as orchestrated Web services. These imagined Web services can then be specified by an AWSDL file, which can be used afterwards to define the desired process by an ABPEL file.

The difference between defining a normal BPEL and an ABPEL lies in specifying partner links. They reference portTypes from the abstract WSDL, instead of referencing external portTypes (see Figure 5.4). This can be done by creating a wrapper for each portType, that defines a new partnerLinkType with the role "processRole".

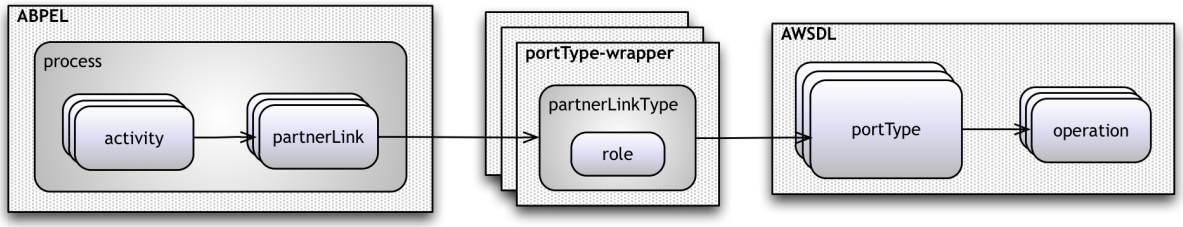


Figure 5.4: The abstract BPEL file.

Thus, an abstract WSDL allows the user to define an abstract BPEL process transparently, without being aware of the actual services, nor by whom they are being provided.

2.2 Discovery Layer

In this layer, the AWSDL and ABPEL files are analyzed in order to extract the specified functional and non-functional requirements. The keywords specifying the functional requirements are exploited by the service retriever, which retrieves a set of services (WSDL interfaces) using a Web service resource. These retrieved services are filtered and parsed. Then, the set of specified input/output parameters are used for checking the compatibility of each retrieved service. Concerning the non-functional requirements, they will be represented as queries to be processed in the classification layer. They are used to identify the services satisfying them.

2.2.1 Web Service Retriever

The requirements analyzer sends to this component the keywords specified for the parameter names input/output for each needed operation. The service retriever works also on retrieving the QoS information, when it is provided by the service resource. The retrieved WSDL interfaces are then parsed, in order to extract their operations signatures including their parameters names and types.

2.2.2 WSDL Parser

Each set of the retrieved services is passed to this parser, in order to extract for each service its operations with their input/output parameters. The parser works on resolving the references found inside each passed WSDL interface. It formulates the service signatures by extracting the operation name together with the input/output parameters together with their types. In WSDL interfaces, parameters are specified as elements of an XML schema, they may be either simple or complex. Simple types are the XML schema primitive types as int, float, string, etc. Complex types contain several elements having either complex or simple types. The parser works on unfolding these complex types, in order to extract all of its sub elements. The unfolded complex type forms a string of concatenated sub elements, representing a type hierarchy. In this way, all of the elements are conserved and none of the semantics conveyed by them is lost. For example, in Figure 5.5, we can see a WSDL interface for a Weather service with one operation "GetWeatherInfo". This operation returns a parameter of a complex type called Weather.

```
<definitions ...>
    <types>
        <xsd:schema ...>
            <xsd:complexType name="Weather">
                <xsd:sequence>
                    <xsd:element name="Visibility" type="xsd:float" />
                    <xsd:element name="WindSpeed" type="xsd:float" />
                    <xsd:element name="Humidity" type="xsd:int" />
                    <xsd:element name="Temperature">
                        <xsd:complexType>
                            <xsd:sequence>
                                <xsd:element name="Celsius" type="xsd:float" />
                                <xsd:element name="Fahrenheit" type="xsd:float" />
        ...
        ...
    <message name="GetWeatherInfoRequest">
        <part name="Country" type="xsd:string"/>
        <part name="City" type="xsd:string"/>
    </message>
    <message name="GetWeatherInfoResponse">
        <part name="Result" type="tns:Weather"/>
    </message>
    <portType name="WeatherServicePortType">
        <operation name="GetWeatherInfo">
            <input name="input" message="tns:GetWeatherInfoRequest"/>
            <output name="output" message="tns:GetWeatherInfoResponse"/>
        </operation>
    </portType>
</definitions>
```

Figure 5.5: A WSDL interface describing a weather service.

The result returned by the parser is shown in Figure 5.6.

```
Web service name : WeatherService
-----------------------------------------------------------------------------------------------------------
Operation name : GetWeatherInfo

Input parameters
name : GetWeatherInfoRequest.Country, type : string
name : GetWeatherInfoRequest.City, type : string

Output parameters
name : GetWeatherInfoResponse.Result.Weather.Visibility, type : float
name : GetWeatherInfoResponse.Result.Weather.WindSpeed, type : float
name : GetWeatherInfoResponse.Result.Weather.Humidity, type : int
name : GetWeatherInfoResponse.Result.Weather.Temperature.Celsius, type : float
name : GetWeatherInfoResponse.Result.Weather.Temperature.Fahrenheit, type : float

-----------------------------------------------------------------------------------------------------------
```

Figure 5.6: The operation signature returned by the WSDL parser.

2.3 Classification Layer

In the classification layer, the extracted service signatures, the retrieved QoS, as well as the specified queries are processed inside the handler component. The handler component can be very variable according to the use case, as we shall see in the following chapters. In general, it prepares the collected services with their data, in order to generate the corresponding contexts for FCA/RCA classifier. This classifier analyzes the generated contexts and builds the corresponding FCA/RCA lattices according to the considered use case.

2.4 Selection Layer

Finally, in the selection layer, the lattice interpreter navigates the generated lattices, and extracts the candidate (substitutable) services that best match the specified user requirements. This interpreter implements the algorithm specified in Section 4-4.3, which navigates the resulting lattices and identifies the services that best match the user specified requirements.

3 Contribution Outline

In this thesis, we validated our proposed framework according to three different use cases, which are presented respectively in the following chapters. Each use case represents a specialization of this general framework for the realization of a certain purpose. Therefore, each one of these use cases can be read and understood independently of the others.

In Chapter 6, we present the first framework's use case, which illustrates a browsing mechanism that facilitates the selection of Web services by keywords, and the identification of their backups.

In Chapter 7, we present the second use case, in which, we propose using the framework to extract groups of mutually similar operations. Each one of these groups is considered as a

functionality. This enables classifying the services by their functionality, and reveals the substitutability relations between them.

In Chapter 8, we present the third and final use case, which clarifies the framework utilization as an assistant for building Web service orchestrations. In this use case, we deal with Web service selection according to user requirements, specified by an abstract WSDL and an abstract BPEL files. These requirements are specified on three levels: the services' functionality, their QoS, and their composability.

In Chapter 9, we demonstrate the experiments that are conducted for each one of the framework's use cases. In these experiments, we use real Web services retrieved from Web service search engines, in order to validate our framework and its various components.

Chapter 6

Web Service Selection by Tags

Contents

1 Introduction

In this chapter, we present the first use case of our framework. In this use case, we show the utility of our framework for the selection of one or more independent services, along with their possible substitutes.

We use Formal Concept Analysis (FCA) to classify Web services into concept lattices according to their automatically extracted tags (the significant keywords appearing in their documentations). A service lattice reveals the invisible relations between the services. It is considered as a browsing mechanism that facilitates the selection of a needed service, and the identification of its candidate backups. The dynamic use of such backups to replace a defecting service enables a continued functionality of a Web service, which becomes indispensable, especially when a service represents a part of a composite application.

We explain this chapter using a scenario for travel reservation, which we call the *Travel Composite Service (TCS)*. Afterwards, we describe the components used in this specialization of our framework, especially in the classification layer.

Scenario

Let us consider the following travel scenario: a traveller needs to reserve a plane ticket to a desired city. Supposing that this traveller lives in a small city that has no airport, then, he should also travel to the city where the airport is located, in order to take the plane he reserved. Thus, he must also reserve a train ticket, from his home city to the airport city, taking into consideration the flight's exact time with the time needed to travel between the two cities. This scenario can be achieved by a *Travel Composite Service (TCS)*, in which three functionalities must be satisfied:

- *reservation of a flight* from home city, or airport city, towards a desired city,
- if a train reservation is needed (in case that the home city is not the airport city), then *calculation of the needed time (duration)* to travel between the two cities by train,
- then *reservation of a train* ticket corresponding to the previous duration. The depart time of this train must considers the exact flight time and the calculated duration.

The TCS can be realized by discovering two services offering the described functionalities and composing them. It may look like the composition in Figure 6.1. In this composition, if the $TrainWS$ service crashes, for example, an equivalent service offering at least the two used $calcDuration()$ and $resTrain()$ operations must be searched and discovered, in order to recover the missing functionality, and ensure the continuity of the composition.

2 The Selection Framework: Use Case 1

In Figure 6.2, we can see the different layers of our framework, which are specialized according to the current use case. We explain this use cases on several steps. We start by retrieving a set of WSDL interfaces according to some user requirements specified in an AWSDL file. Then, from the retrieved WSDL interfaces, we extract a set of service documentations. We pass the extracted documentations to the classification layer, where we automatically tag the services with significant keywords. Finally, the services with their tags are used to build the corresponding service lattice using FCA, from which, the needed service along with its backups can be selected.

In the following, we clarify the layers and the components that are used to achieve the desired service classification.

2.1 Discovery Layer

In this layer, the analyzer parses the AWSDL file in order to extract the set of keywords describing each needed service. These keywords are located in the documentation element of the portType. In our scenario, the AWSDL file would contain two portTypes: one for plane reservation, and one for train reservation. The documentation element of the plane portType would have the keywords {reserve, plane}, while they would be {reserve, train, duration} for the train portType. Then, the service retriever fetches a set of services corresponding to the extracted keywords. The service retriever also gathers the user tags associated to the retrieved services (if there is

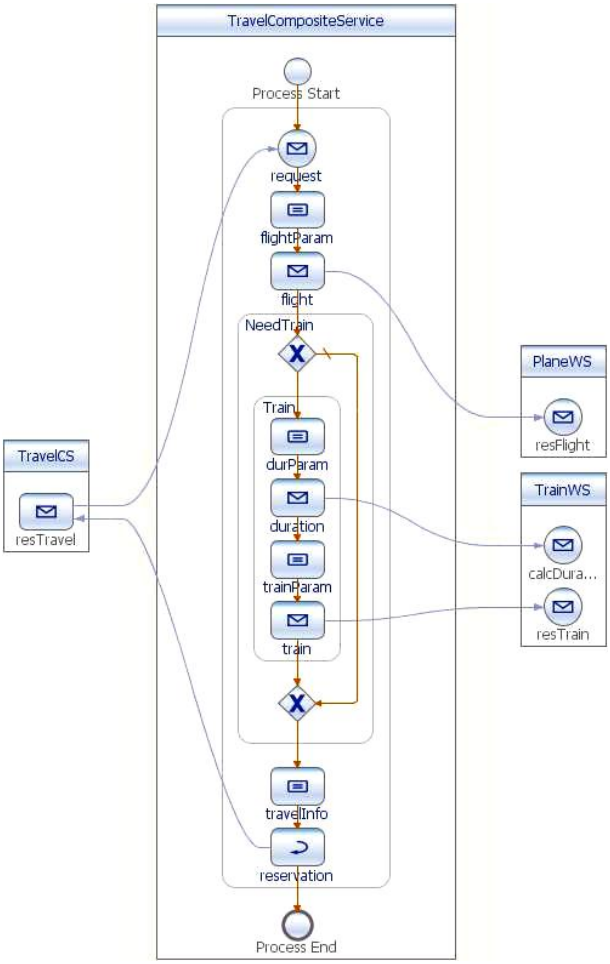


Figure 6.1: The travel composite service (TCS).

any). The WSDL parser extracts the documentation elements in each retrieved service (WSDL interface). Finally, the extracted documentation elements along with the retrieved tags are passed to the classification layer to be processed as described below.

2.2 Classification Layer

In the classification layer, the documentation elements and the user tags are processed, as we mentioned above. They are passed through an automatic services tagger, which is based on text mining and machine learning techniques. This tagger learns from the user tags how to handle the words in documentation elements, in order to automatically extract tags for untagged services.

After having extracted tags for each retrieved service, the set of tags are used to generate a formal context. This formal contexts consist of service identifiers as its objects, and the extracted tags as its attributes. We also insert two keywords queries in the context, which are previously extracted in the discovery layer. This context is passed to the FCA classifier that generated the corresponding lattice.

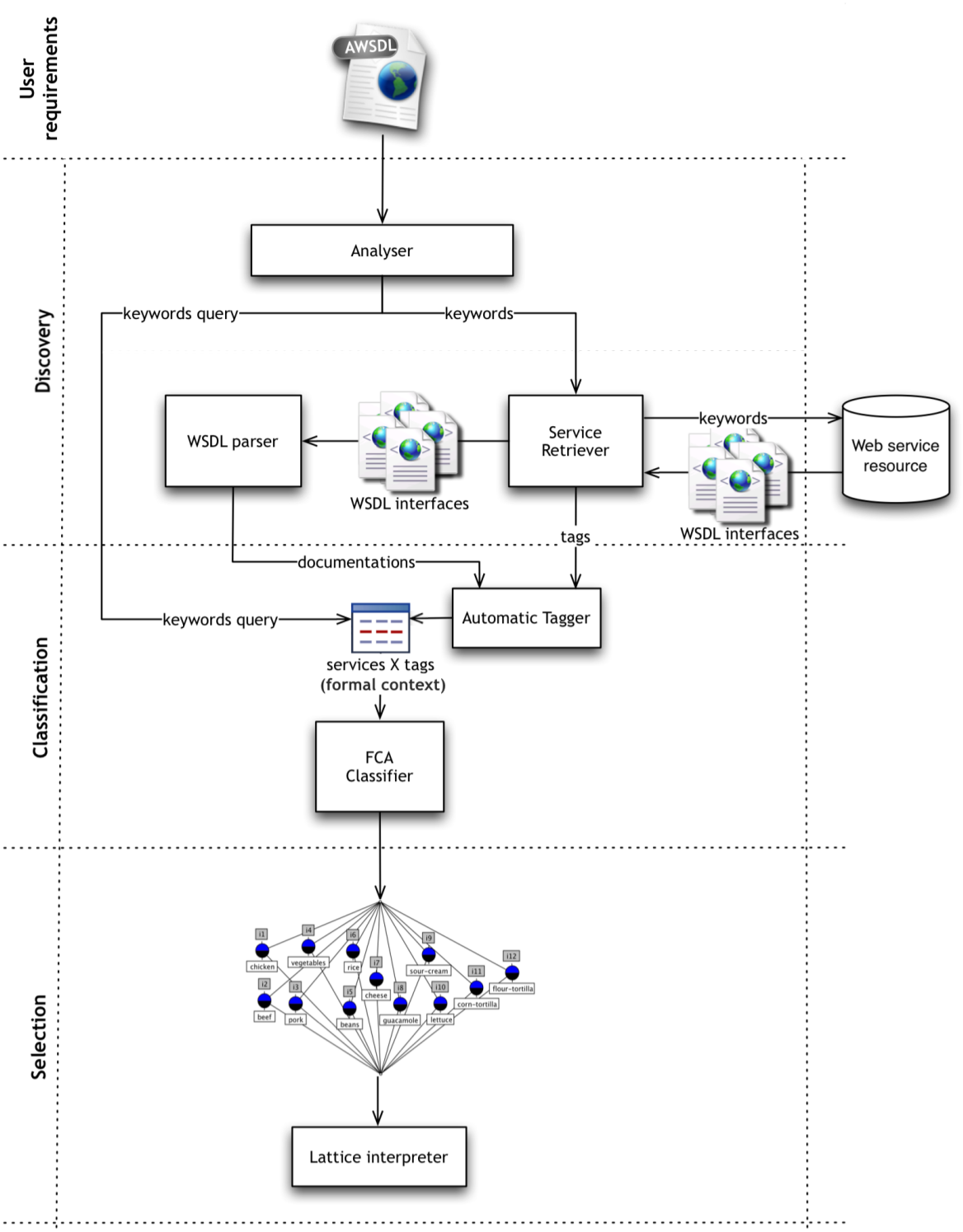


Figure 6.2: The main steps of our approach.

2.2.1 Automatic tagger

In this component, we model the tag extraction problem as the following classification problem: classifying a word into one of the two *tag* and *no tag* classes. Our overall process is divided into two phases: the *training* phase and the *tag extraction* phase.

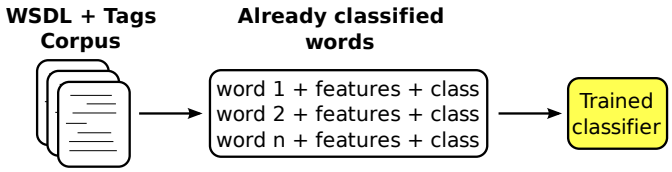


Figure 6.3: The training phase

Figure 6.3 summarizes the behavior of the training phase. In this phase we dispose of a corpus of WSDL files and associated tags. The creation of this *training corpus* is described in Section 2.2.2. From this training corpus, we first extract a list of candidate words by using text-mining techniques. The extraction of these candidates is described in Sections 2.2.3 and 2.2.4. Then several *features* are computed on every candidate. A *feature* is a common term in the machine learning field. It can be seen as an attribute that can be computed on the candidates (for instance the frequency of the words in their WSDL file). Finally, since manual tags are assigned to those WSDL files, we use them to classify the candidate words coming from our WSDL files. Using this set of candidate words, computed features and assigned classes, we train a classifier. This trained classifier will then be used to classify words coming from subsequent WSDL files during the *tag extraction* phase.

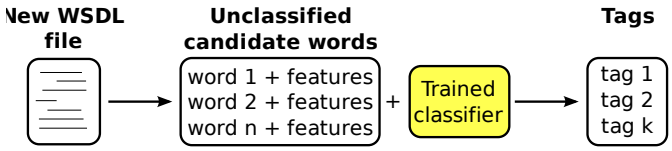


Figure 6.4: The tag extraction phase

Figure 6.4 describes the *tag extraction phase*. First, like in the *training phase*, a list of candidate words is extracted from an untagged WSDL file. The same features as in the training phase are then computed on those words. The only difference with the training phase is that we do not know in advance which of those candidates are true tags. Therefore we use the previously trained classifier to automatically perform this classification. Finally the tags extracted from the WSDL file are the words that have been classified in the *tag* class. It is noteworthy to remark that the *training* phase is only performed once, while the *tag extraction* phase can be applied an unlimited number of times.

2.2.2 Creation of the training corpus

As explained above, our approach requires a training corpus, denoted by $\mathcal{T}$. Since we want to extract tags from WSDL files, $\mathcal{T}$ has to be a set of couples $(wsdl, tags)$, with $wsdl$ a WSDL file, and $tags$ a set of corresponding manually assigned tags.

To clean the tags of the training corpus, we performed the three following operations:

- We removed the non alpha numeric characters from the tags (we found several tags like *_onsale* or *:finance*),
- We removed a meaningless and highly frequent tag (the *_unkown* tag),
- We divided the tags with length $n > 1$ into n tags of length 1, in order to have only tags of length 1. The length of a tag is defined as the number of words composing this tag.

Now that we have this training corpus, we will shortly describe the approach upon which our work is built.

2.2.3 Pre-processing of the WSDL files

As we have seen before, a WSDL file contains several element definitions optionally containing a plain-text documentation. The left side of Figure 6.5 shows such a data structure. In order to simplify the WSDL XML representation in a format more suitable to apply text mining techniques, we decided to extract two documents from a WSDL description:

- A set of couples $(type, ident)$ representing the different elements defined in the WSDL. We have $type \in$ *(Service,Port,PortType,Message,Type,Binding)* the type of the element and *ident* the identifier of the element. We call this set of couples the *identifier set*.
- A plain text containing the union of the plain-text documentations found in the WSDL file, called the *global documentation*.

This pre-processing operation is summarized in the Figure 6.5.

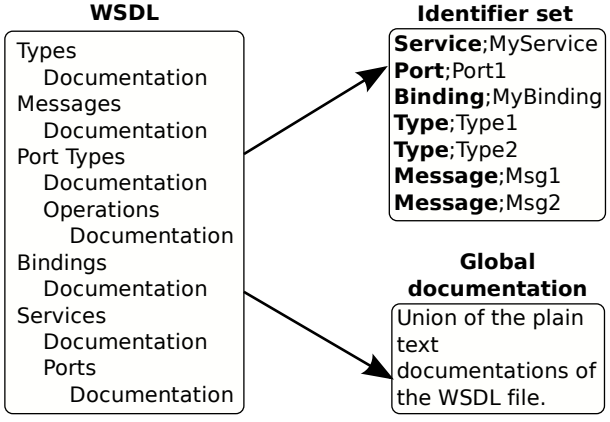


Figure 6.5: WSDL pre-processing

2.2.4 Selection of the candidate tags

As seen in the previous section, we dispose now of two different sources of information for a given WSDL: an *identifier set* and a *global documentation*. Unfortunately, those data are not yet usable to compute meaningful metrics. Firstly because the identifiers are names of the form

MyWeatherService, and therefore are very unlikely to be tags. Secondly because this data contains a lot of obvious useless tags (like the *you* pronoun). Therefore, we will now apply several text-mining techniques on the identifier set and the global documentation.

Figure 6.6 shows how we process the *identifier set*. Here is the complete description of all the performed steps:

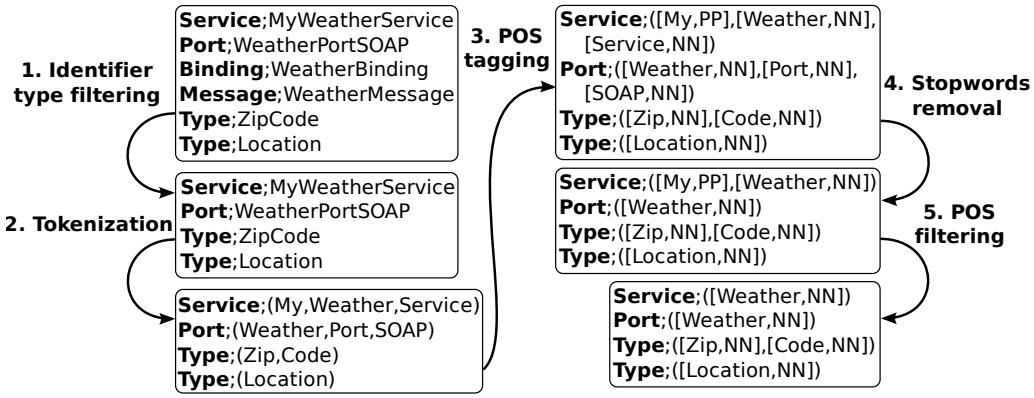


Figure 6.6: Processing of the identifiers

1. **Identifier type filtering**: during this step, the *(type,ident)* couples where *type* ∈ *(PortType,Message,Binding)* are discarded. We applied this filtering because very often, the identifiers of the elements in those categories are duplicated from the identifiers in the others categories.

2. **Tokenization**: during this step, each couple $(type, ident)$ is replaced by a couple $(type, tokens)$. *tokens* is the set of words appearing in *ident*. For instance, *(Service,MyWeatherService)* would be replaced by *(Service,[My,Weather,Service])*. To split *ident* into several tokens, we created a tokenizer that uses common clues in software engineering to split the words. Those clues are for instance a case change, or the presence of a non alpha-numeric character.

3. **POS tagging**: during this step each couple $(type, tokens)$ previously computed is replaced by a couple $(type, ptokens)$. *ptokens* is a set of couples $(token_i, pos_i)$ derived from *tokens* where $token_i$ is a token from *tokens* and pos_i the part-of-speech corresponding to this token. We used the tool *tree tagger* [69] to compute those part-of-speeches. Example: *(Service,[My,Weather,Service])* is replaced by *(Service,[(My,PP),(Weather,NN),(Service,NN)])*. *NN* means noun and *PP* means pronoun.

4. **Stopwords removal**: during this step, we process each couple $(type, ptokens)$ and remove from *ptokens* the elements $(token_i, pos_i)$ where $token_i$ is a *stopword* for *type*. A stopword is a word too frequent to be meaningful. We manually established a stopword list for each identifier type. Example: *(Service,[(My,PP),(Weather,NN),(Service,NN)])* is replaced by *(Service,[(My,PP)(Weather,NN)])* because *Service* is a stopword for service identifiers.

5. **POS filtering**: during this step, we process each couple $(type, ptokens)$ and remove from $ptokens$ the elements $(token_i, pos_i)$ where $pos_i \notin$ *(Noun,Adjective,Verb,Symbol)*. Example: *(Service,[(My,PP),(Weather,NN))* is replaced by *(Service,[(Weather,NN)])* because pronouns are filtered.

Figure 6.7 shows how we process the *global documentation*. Here is the complete description of all the performed steps:

1. **HTML tags removal**: the HTML tags (words begining by $<$ and ending by $>$) are removed from the global documentation.
2. **POS tagging**: similar to the POS tagging step applied to the identifier set.
3. **POS filtering**: similar to the POS filtering step applied to the identifier set.

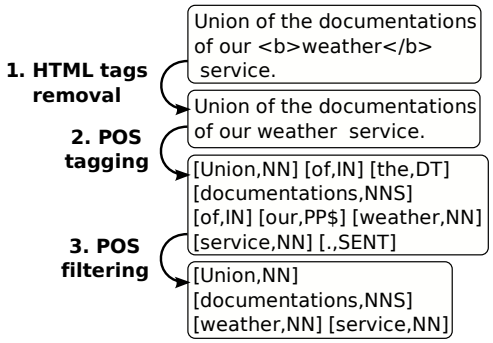


Figure 6.7: Processing of the global documentation

The union of the remaining words in the identifier set and in the global documentation are our candidate tags. When defining those processing operations, we took great care that no correct candidate tags (i.e. a candidate tag that is a real tag) of the training corpus have been discarded. The next section describes how we adapted the features of a technique for keyphrase extraction called Kea [70], for the candidate tags.

2.2.5 Computation of the features

After having applied our text mining techniques on the identifier set and the global documentation, we dispose now of different well separated words. Therefore we can now compute the $tfidf$ feature [71]. But words appearing in documentation or in the identifier names are not the same. We decided (mostly because it turns out to perform better) to separate the $tfidf$ value into a $tfidf_{ident}$ and a $tfidf_{doc}$ which are respectively the $tfidf$ value of a word over the identifier set and over the global documentation. Like in Kea, we used the method in [72] to discretize those two real-valued features.

The *distance* feature still has no meaning over the identifier set, because the elements of a WSDL description are given in an arbitrary order. Therefore we decided to adapt it by defining five different features: $in\_service$, in_port, in_type, $in\_operation$ and $in\_documentation$. Those features take their values in the $(true, false)$ set. A $true$ value indicates that the word has

been seen in an element identifier of the corresponding type. For instance *in_service(weather) = true* means that the word *weather* has been seen in a service identifier. *in_documentation(weather) = true* means that the word *weather* has been seen in the global documentation.

In addition of these features, we compute another feature called *pos*. We added this feature, not used in Kea, because it significantly improves the results. *pos* is simply the part-of-speech that has been assigned to the word during the POS tagging step. If several parts-of-speech have been assigned to the same word, we choose the one that has been assigned in the majority of the cases. The different values of *pos* are: *NN (noun), NNS (plural noun), NP (proper noun), NPS (plural proper noun), JJ (adjective), JJS (plural adjective), VV (verb), VVG (gerundive verb), VVD (preterit verb), SYM (symbol).*

2.2.6 Training and using the classifier

We applied the previously described technique to all the WSDL files of $\mathcal{T}$. In addition to the previously described features, we compute the is_tag feature over the candidates. This feature takes its values in the $(true, false)$ set. $is\_tag(word) = true$ means that $word$ has been assigned as a tag by Seekda users for its service description. We have serialized all those results in an *ARFF* file compatible with the Weka tool [73]. Weka is a machine learning tool that defines a standard format for describing a training corpus and furnish the implementation of many classifiers. One can use Weka in order to train a classifier or compare the performances of different classifiers regarding a given classification problem. Table 6.1 shows an extract of the ARFF file we produce. In this table, words are displayed for the sake of clarity, but in reality, they are not present in the ARFF file. The ARFF file only contains features.

Table 6.1: Extract of the ARFF file

Word	$TFIDF_{id}$	$TFIDF_{doc}$	$IN\_SERVICE$	...	IN_DOC	POS	IS_TAG
Weather	$[0, 0.01]$	$]0.01, 0.04]$	×			NN	×
Location	$]0.03, 0.1]$	$]0.04, 0.15]$			×	JJ	
Code	$]0.03, 0.1]$	$]0.01, 0.04]$			×	VV	

With this ARFF file, we used Weka to train a naive Bayes classifier, shown as optimal for our kind of classification task [74]. This trained classifier can now be used in the tag extraction phase. As previously said, the beginning of this phase is the same as the one of the training phase. It means that the WSDL file goes through the previously described operations (preprocessing, candidates selection and features computation). Only this time, the value of the is_tag feature is not available. This value will be automatically computed by the previously trained classifier.

2.2.7 WordNet for semantically related tags

In our approach, the classifier that we built determines whether a word in a WSDL file is a tag or not. Thus, it extracts the tags appearing inside the WSDL files only. This way, we miss some other interesting tags like associated words or synonyms. In order to solve this issue, we used the WordNet lexical database [75]. In WordNet a word may be associated with many synsets (synonym sets), each corresponding to a different sense of a word.

Each WSDL file of our corpus is assigned two sets of tags: user tags and our automatically extracted tags. Our objective is to enrich each set of tags with semantically similar words extracted from WordNet. Thus, for each tag we identify the possible senses and the synonyms set related to each sense. We add the extracted synonyms to the corresponding set of tags, and we perform some experiments to evaluate the obtained tags.

2.2.8 FCA Classifier

This classifier takes as input a formal context of services characterized by the tags extracted by the automatic tagger. The keywords queries specified in the AWSDL file are inserted into the context. The context is then analyzed by the classifier, in order to generate the corresponding lattice of services and tags.

In our scenario, we suppose retrieving ten services and extracting their tags. The formal context corresponding to these services would be similar to the context in Table 6.2. The lattice corresponding to this context is shown in Figure 6.8.

Table 6.2: Formal context...

	plane	train	ticket	reserve	distance	duration	taxi	car	travel	cancel	meal	airport	cheap	rent
ws1	×		×	×	×	×	×	×	×	×	×	×	×	×
ws2	×		×	×						×		×		
ws3	×			×		×			×	×		×	×	
ws4	×			×						×		×		
ws5		×	×	×	×	×	×	×	×	×	×		×	×
ws6		×	×	×		×				×	×			
ws7		×		×						×	×		×	
ws8		×		×	×	×	×		×	×				
ws9	×	×	×	×		×			×	×			×	
ws10	×	×	×	×	×	×		×		×	×	×	×	×

We want to use this lattice to discover the required services. We navigate a lattice by integrating a query into the context, and classify it into the lattice. The services that answer the specified query are located in the sub-concepts of the lowest concept in the lattice, where the query appears, as explained in Chapter 4. In our scenario, we have two keywords queries: $query1 = \{reserve, plane\}$ and $query2 = \{reserve, train, duration\}$. These queries are inserted into the context as two new lines, as in Table 6.3. This context containing the inserted queries results in the lattice shown in Figure 6.9.

Table 6.3: Formal context...

	plane	train	ticket	reserve	distance	duration	taxi	car	travel	cancel	meal	airport	cheap	rent
query1	×			×										
query2		×		×		×								

2.3 Selection Layer

This lattice is interpreted in the selection layer, which extracts the services satisfying the two queries and they are:

– for $query1$: $ws_1, ws_2, ws_3, ws_4, ws_9$ and ws_{10},

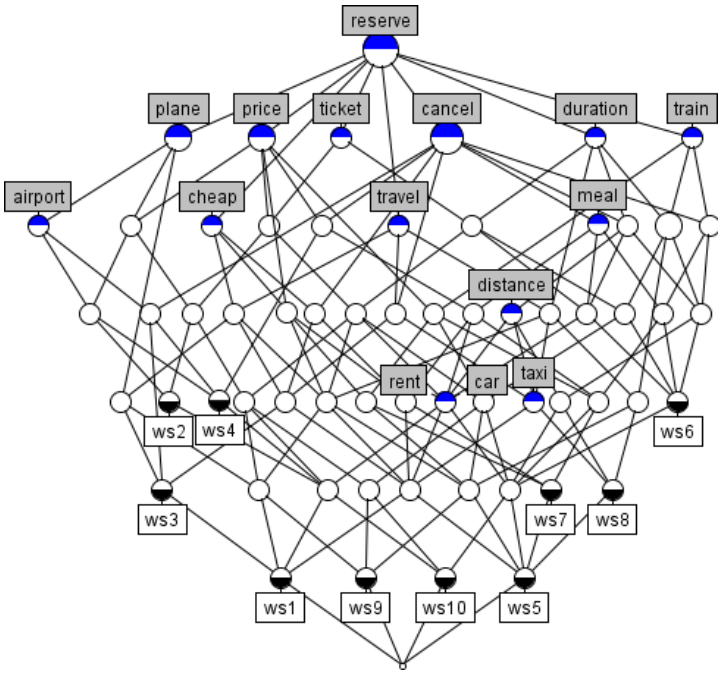


Figure 6.8: The concept lattice for a set of services and their keywords.

– for $query2$: $ws_5, ws_6, ws_8, ws_9, ws_{10}$.

In the next chapter, we present the second use case, in which we explain how to classify the services by the functionality they provide. Using this specialization of the framework, we will be able to further analyse the two sets of services corresponding to the two queries, in our scenario. We apply a similarity measure on the operations according to each set and we identify the groups of similar operations. For example, an operation labelled $reserveF$ in the lattice, represents a group of similar operations for performing a flight reservation. This way, we can build a new context of $service \times operation$ for each set of services. The lattice that corresponds to $query1$ is shown in Figure 6.10 (top), and the one corresponding to $query2$ in Figure 6.10 (bottom).

Using these two lattices, the selection of services offering required operations is straightforward. In our scenario, we need three functionalities as indicated before in Section 1: flight reservation, train reservation (if needed), and calculating the duration needed to travel by train to the aimed destination. By regarding the lattice in Figure 6.10 (top), we notice that all of the services offer an operation for plane reservation. In this case, a service selection might be done regarding the extra operations that the services provide, like for example the operation $rentCar$. When we select a certain service, we can immediately extract the set of backup services that are able to replace it if it fails. In the same way, we can select a service for train reservation with obtaining the duration information. Thus, the composition in Figure 6.1 can be easily achieved and supported with backups, as in Figure 6.10.

Supposing that we selected the service ws_4 named $PlaneWS$, and used its operation $resFlight$ (which is grouped with other similar operations under the name $reserveF$ in the lattice). We

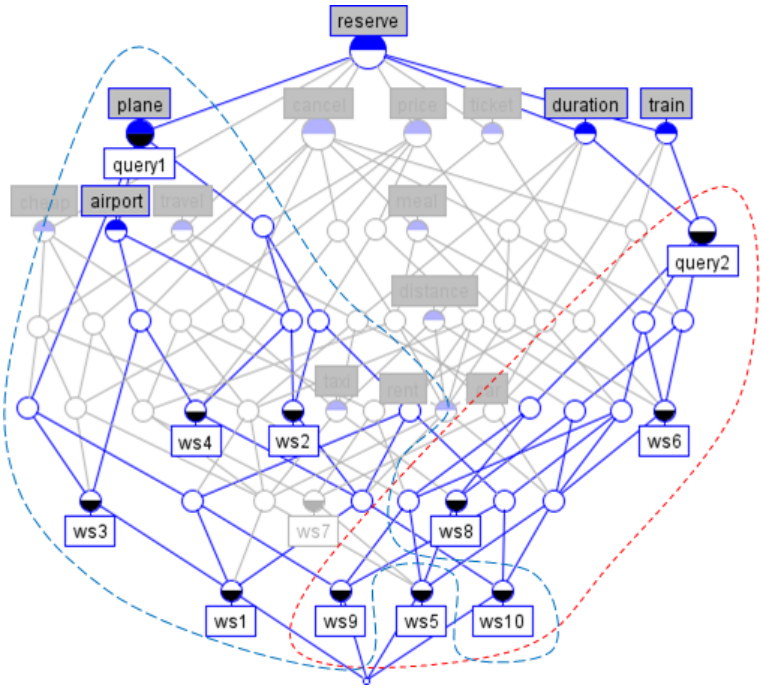


Figure 6.9: Queries as concepts in the *service* × *keyword* lattice.

can notice that for the operation *resFlight*, any other service in the lattice can be a backup for ws_4. In case where all of the operations of ws_4 were used, we notice that only ws_{10} and ws_1 can be backups for ws_4. Similarly, we selected the service ws_6, named $TrainWS$, and used two of its provided operations: *calcDuration* and *resTrain*. We notice that ws_6 can be replaced by the services ws_{10}, ws_8 and ws_5. Thus, we have discovered immediate backups for the service $TrainWS$ as we did for the service $PlaneWS$. We can notice that the service ws_{10} exists in the two lattices, as a backup for both $TrainWS$ and $PlaneWS$. In this case, if both of these services crash, we can replace them both by a single service, which is ws_{10}. This service provides the same functionalities of the two services, with three extra operations that are *priceT*, *rentCar* and *durationF* as can be seen in the lattices in Figure 6.10.

Our approach has enabled us of an easy service discovery and selection, in order to build our aimed scenario. It has also facilitated the discovery of backup services to support service composition and ensure its continuous functionality.

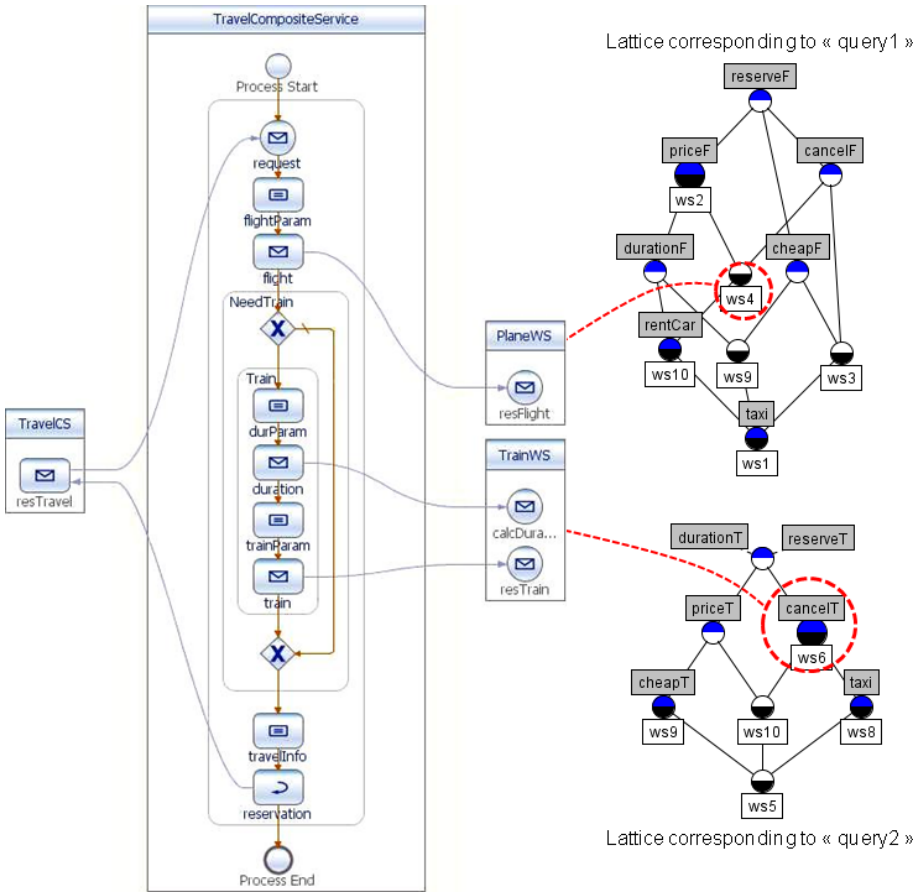


Figure 6.10: The TCS composition with its corresponding backups.

3 Summary

In this chapter, we presented a specialization of our framework that is based on FCA for classifying web services by their tags. In this use case, we showed the utility of our framework for the selection of one or more independent services, along with their possible substitutes. We made use of the keywords specified in the documentation element of each portType inside the AWSDL file, to retrieve services. We retrieve a set of corresponding services and we extract their tags, using our automatic service tagger. The extracted tags are then used to classify Web services, using FCA. The generated lattice offers a browsing mechanism, which enable us to identify a set of services (with their backups) for a set of given keywords. In Chapter 9, we show the experiments that were carried out, in order to validate the current use case.

In the next chapter, we present the second use case of our framework. We consider that the second use case is complementary to the first one. We explain how to extract groups of similar operations, in order to classify Web services by their functionality. We also base our classification on FCA, but using many valued contexts of similarity values, as we shall see next.

Chapter 7

Web Service Selection by Functionality

Contents

1 Introduction

In this chapter, we present the second use case of our framework. In this use case, we enable browsing Web services by their functionality, in order to facilitate the selection of a service offering needed operations, and the identification of its possible backups. We accomplish this by constructing Web service lattices using many-valued contexts of similarity values calculated for each pair of operations. This enables us of extracting groups of mutually similar operations. Each one of these groups represents a functionality, according to which, a new service lattice can be generated. The generated service lattices provide us with browsing and navigation capabilities. This allows the retrieval of more general to more specific sets of services [76, 77]. More general sets have lesser common operations while more specific sets have more common operations. Therefore, applying FCA to Web services provides us with a retrieval mechanism, which facilitates both selection of Web services and identification of their possible substitutes. Accordingly, Web service selection becomes more efficient regardless of the aimed utilization.

We explain this chapter using a scenario for currency conversion, which we call the *Composite Currency Service (CCS)*. Afterwards, we describe the components used in this specialization of our framework, with more concentration on the classification layer.

Scenario

Let us consider the two following services: a currency converter service and a calculation service. The currency converter service offers an operation that returns the exchange rate between two entered currencies. The calculation service offers several operations for calculation, one of them returns the multiplication of two entered numbers. We link the exchange rate operation with the multiplication operation in order to build a composite service of the two mentioned services, we call it the *Composite Currency Service (CCS)*. This new composite service gives us a converted amount from one currency to another. Figure 7.1 shows us an overview of this composite service.

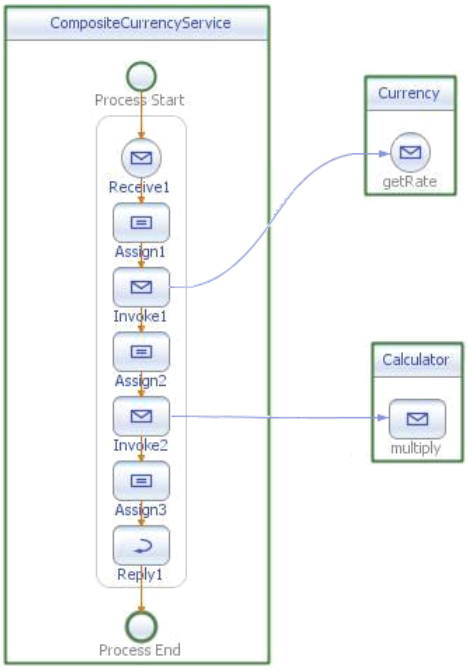


Figure 7.1: The composite currency service.

In this composition, if the service *Calc* crashes, for example, a service offering a multiplication operation equivalent to *mul*() must be searched and discovered. Thus to recover the missing functionality, and ensure the continuity of the composition.

2 The Selection Framework: Use Case 2

In Figure 7.2, we can see the different layers of our framework, which are specialized according to the current use case. We explain this use case on two parts. In the first part, we retrieve a set of WSDL interfaces according to some user requirements specified in an AWSDL file. Nevertheless,

in this use case, we use only the keywords that describe each needed service. From the retrieved WSDL interfaces, we extract a set of service documentations that are passed to the second part.

In the second part, we focus on the classification layer, and we clarify the components that are used to achieve the desired service classification.

2.1 Discovery Layer

In this layer, the AWSDL file gets analyzed, in order to extract the set of keywords describing each needed service. These keywords are located in the documentation element of the portType. In our scenario, the AWSDL file would contain two portTypes: one for currency conversion, and one for calculation. The documentation element of the plane portType would have the keywords {currency, converter, exchange, rate}. The documentation element of the calculator portType would have the keyword {multiply}. Then, the service retriever returns a set of services corresponding to the extracted keywords. The WSDL parser extracts the operation signatures in each retrieved service (WSDL interface), and passes them to the classification layer to be processed as described below.

2.2 Classification Layer

In the classification layer, we aim at classifying the services by their operations using FCA. The formal context that should be defined for such classification is based on the relation: "a service provides an operation". The problem in determining this relation is that Web services may offer similar operations but not necessarily with identical signatures. For example, if we have the following operations: add(x, y) =? addition(a, b), they may provide the same functionality, but without having the exact signature. Therefore, we decided to build the service × operation classification following several steps:

- measuring the similarity between operation signatures for the set of retrieved services;
- generating a many-valued context of operation × operation, containing the similarity values (in the range [0,1]);
- scaling this many-valued context into several binary contexts, according to threshold values;
- generating the operations lattices (a lattice for each threshold), and extracting the groups of mutually similar operations;
- then finally, building the contexts service × operation, according to the extracted groups of operations, and generating the corresponding lattices.

We explain the components of this layer using an imaginary example, for the sake of clarity. This example consists of a set of services illustrated with their signatures in Table 7.1.
We calculate the similarity between each pair of signatures of each pair of distinct services, using the following component.

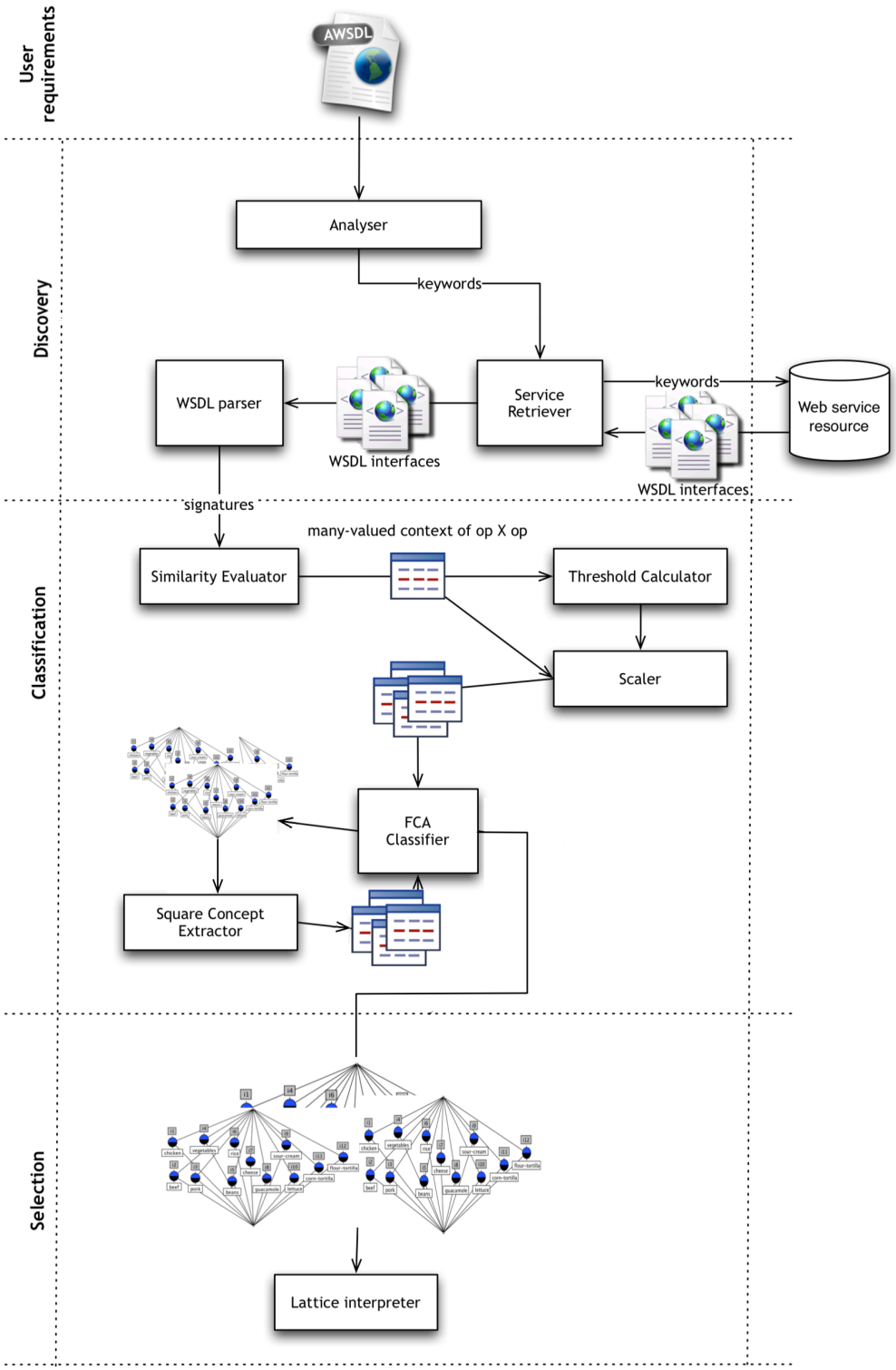


Figure 7.2: The main steps of our approach.

Table 7.1: A set of calculation services with their operations.

Services	Id	Operations	Id
$Calc_1$	ws_1	add(a,b)	op_{11}
		sub(a,b)	op_{12}
$Calc_2$	ws_2	add(a,b,c)	op_{21}
$Calc_3$	ws_3	add(a,b,c,d)	op_{31}
		sub(a,b,c)	op_{32}
		mult(a,b)	op_{33}
		add(a,b,c)	op_{34}

2.2.1 Similarity Evaluator

There are several similarity measures for Web services that evaluate similarity according to syntax and semantics, such as [29, 78, 79]. Similarity is assessed in the form of values in the range [0,1]. If two operations are sufficiently similar, the similarity value will approach 1, otherwise it will approach 0. In this component, we make use of the Jaro-Winkler string distance measure [80] as a similarity measure. This measure calculates the similarity between two strings, and has proved to be efficient and accurate [81]. Although, this is a primary solution for similarity measuring, which we plan to improve in future work. As we mentioned above, we measure the similarity between each pair of signatures of each pair of distinct services. We do not evaluate similarity between operations provided by the same service (we suppose that it is equal to 0), because when a service becomes dysfunctional, all of its operations become dysfunctional too.

Formally, a similarity measure $Sim : \mathbb{O} \times \mathbb{O} \rightarrow [0, 1]$ can be defined as follows:
$\forall\, op_{ij} \in \mathbb{O} \Longrightarrow Sim(op_{ij}, op_{ij}) = 1$ (an operation with itself)
$\forall\, op_{ij}, op_{ik} \in \mathbb{O}, j \neq k \Longrightarrow Sim(op_{ij}, op_{ik}) = 0$ (operations in the same service)
$\forall\, op_{ij}, op_{nm} \in \mathbb{O}, i \neq n \Longrightarrow Sim(op_{ij}, op_{nm}) \in [0, 1]$ (operations in different services)

The calculated similarity values forms a many-valued context, which is a symmetric square matrix that we will call $SimMat$, as shown in Table 7.2. This matrix is of size $n = |\mathbb{O}|$, and its diagonal elements are all equal to 1 (similarity of an operation with itself).

From the similarity matrix $SimMat$, we can extract several binary contexts, by specifying threshold values $\theta \in]0, 1]$. These threshold values are calculated using the following component.

2.2.2 Threshold Calculator

This component implements a statistical technique called BoxPlot++ (Appendix A). It extracts the distinct similarity values in the $SimMat$ and determines several thresholds representing these values.

2.2.3 Scaler

According to the determined thresholds passed from the previous component. The scaler works on converting the $SimMat$ into a binary context. Thus, the values of $SimMat$ that are greater

or equal to the chosen threshold θ are scaled to 1, while other values are scaled to 0. For example, the binary context that corresponds to $\theta = 0.75$ is shown in Table 7.3, we call it $SimCxt$.

Table 7.2: The similarity matrix ($SimMat$).

	op_{11}	op_{12}	op_{21}	op_{31}	op_{32}	op_{33}	op_{34}
op_{11}	1	0	0.75	0.5	0	0	1
op_{12}	0	1	0	0	0.75	0	0
op_{21}	0.75	0	1	0.75	0	0	0.75
op_{31}	0.5	0	0.75	1	0	0	0
op_{32}	0	0.75	0	0	1	0	0
op_{33}	0	0	0	0	0	1	0
op_{34}	1	0	0.75	0	0	0	1

Table 7.3: The context ($SimCxt$) for $\theta = 0.75$.

	op_{11}	op_{12}	op_{21}	op_{31}	op_{32}	op_{33}	op_{34}
op_{11}	×		×				×
op_{12}		×			×		
op_{21}	×		×	×			×
op_{31}			×	×			
op_{32}		×			×		
op_{33}						×	
op_{34}	×		×				×

Formally, the $SimCxt$ context is a triple $(\mathbb{O}, \mathbb{O}, RSim_\theta)$, where $RSim_\theta$ is a binary relation indicating whether an operation is similar to another operation or not.

$$(op_{ij}, op_{nm}) \in RSim_\theta \iff Sim(op_{ij}, op_{nm}) \geq \theta$$

We use the $SimCxt$ context to generate a lattice of operations ($\underline{\mathfrak{B}}(\mathbb{O}, \mathbb{O}, RSim_\theta)$) by the FCA classifier. This lattice is illustrated in Figure 7.3. This lattice helps in discovering groups of similar operations, using the following component.

2.2.4 Square Concept Extractor

In the resulting operation lattice (Figure 7.3), groups of mutually similar operations can be identified by the concepts having equal extent and intent sets. We call such concepts the square concepts, because they form square gatherings on the binary context matrix. We define a group G_{op} of mutually similar operations Op_{Sim} as:

$$G_{op} = \{Op_{Sim} \mid (Op_{Sim}, Op_{Sim}) \in \mathfrak{B}(\mathbb{O}, \mathbb{O}, RSim_\theta)\}$$

The notion of square concepts can be better recognized by performing a mutual column-line interchange in the $SimCxt$. The resulting interchanged context is shown in Table 7.4.

From the lattice in Figure 7.3 as from the interchanged context in Table 7.4, we can identify the groups of similar operations, and they are the following:

Table 7.4: The interchanged ($SimCxt$) context.

	op_{11}	op_{34}	op_{21}	op_{31}	op_{12}	op_{32}	op_{33}
op_{11}	×	×	×				
op_{34}	×	×	×				
op_{21}	×	×	×	×			
op_{31}			×	×			
op_{12}					×	×	
op_{32}					×	×	
op_{33}							×

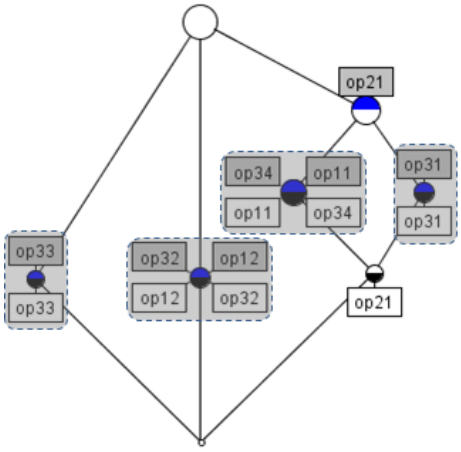


Figure 7.3: The generated lattice for ($SimCxt$) shown in Table 7.3.

– $\{op_{11}, op_{34}, op_{21}\}$ that we label $(11, 34, 21)$;

– $\{op_{21}, op_{31}\}$ labelled $(21, 31)$;

– $\{op_{12}, op_{32}\}$ labelled $(12, 32)$;

– $\{op_{33}\}$ labelled (33).

The groups of similar operations, denoted as $\mathbb{G}$, are used to define the final binary context. This context is a triple $(\mathbb{W}, \mathbb{G}, R)$, in which the relation R indicates whether or not a service offers the functionality represented by the corresponding group of similar operations. We use the labels representing the groups of operations to build the final context, which is shown in Table 7.5. Using this context, the FCA classifier generates the corresponding service lattice, which is shown in Figure 7.4.

Table 7.5: The final services × groups context.

	(11,34,21)	(21,31)	(12,32)	(33)
ws_1	×		×	
ws_2	×	×		
ws_3	×	×	×	×

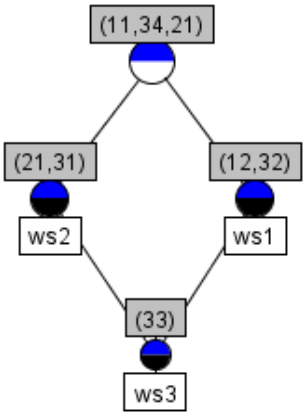


Figure 7.4: The final service lattice with possible backups.

2.3 Selection Layer

From the final generated service lattice, shown in Figure 7.4, we can notice the following:

- ws_1, ws_2, and ws_3 offer the functionality denoted by $(11, 34, 21)$, so they can replace each other for this specific functionality;
- ws_3 can replace ws_1 and ws_2, and it offers an additional functionality (33).

We can also infer immediately which services offer a specific functionality (denoted by a specific label), by considering the indices in the label. For example, the label $(11, 34, 21)$ makes it possible to directly deduce that (11) is provided by ws_1, (34) by ws_3 and (21) by ws_2.

As we shall see later in the experiments (Chapter 9), the resulting service lattices for our scenario can be exploited to build the desired composition, and support it with backup services.

3 Summary

In this chapter, we presented a specialization of our framework that is based on FCA for classifying Web services by their functionality. We achieved this classification by extracting groups of mutually similar operations, identified by what we called square concepts.

The generated lattice reveals the invisible relations between Web services according to their functionality (the operations groups), enabling the identification of substitutable services. This facilitates the selection of needed services and supports them with backups, in order to assure a continuous execution.

The quality of our generated lattices depends on the chosen similarity measure [29, 78, 79] and the similarity threshold. The more accurate the measure is, the more precise the obtained lattice is. The chosen values of threshold will give us a variation of lattices, and they reflect the level of the required adaptations. Thus, a high value of threshold means similar services with a low number of required adaptations.

In Chapter 9, we show the experiments that were conducted, in order to validate the current use case. In the next chapter, we present the third and last studied use case in this thesis. We deal with selection of services considering three levels of user requirements: functionality, QoS, and composition.

Chapter 8

Web Service Selection According to Multi- User Requirements

Contents

1 Introduction

In this chapter, we present the third and final use case of our framework. In this use case, we aim at providing a facility for building business processes transparently, according to user requirements. This means that a user can model his business process in an abstract way, without being aware of the concrete services existing on the Web. This is realized by considering three levels of user requirements: the needed functionality, the accepted QoS levels, and the composition. Then, identifying the services that satisfy these requirements.

We specialize our framework in order to facilitate Web service selection according to user requirements. We use the Relational Concept Analysis (RCA) to characterize the services by their QoS levels and to express the composition relations between them. The generated lattices help in identifying the services that match the specified requirements, with the help of RCA relational queries.

We explain our approach along with a scenario of an abstract process composed of three services, for providing the weather information for a given ip address. We call this process the *WeatherProcess*.

Scenario

Let us consider a process for providing weather information. We call it *WeatherProcess*. It provides weather information for a given ip address. This process is illustrated in Figure 8.1. It orchestrates the invocation of three operations: ipToCity, cityToZipcode, and zipcodeToWeather. The needed services are described inside an abstract WSDL file, which is illustrated in Figure 8.2. It describes three PortTypes (services): CityServicePortType, ZipcodeServicePortType, and WeatherServicePortType. Each service has an expected QoS level for the availability and response time attributes. In Figure 8.2, we expanded the messages description for the operation getWeatherByZipcode. We can see that for the input parameter (zipcode), the user provided a list of four equivalent keywords, which are listed in the WSDL source code. They are as follows: zipcode, zip, postal, postalcode. The parameter type is specified to be string. For the output parameter, we can notice that its name is specified as "any". This means that the user is not asking for a specific parameter name, but he is interested by the complex type, which is in our case Weather. In this case, the user provided a list of 5 equivalent keywords for the Weather type. They are as follows: Weather, WeatherInfo, Forecast, WeatherForecast, and WeatherReport.

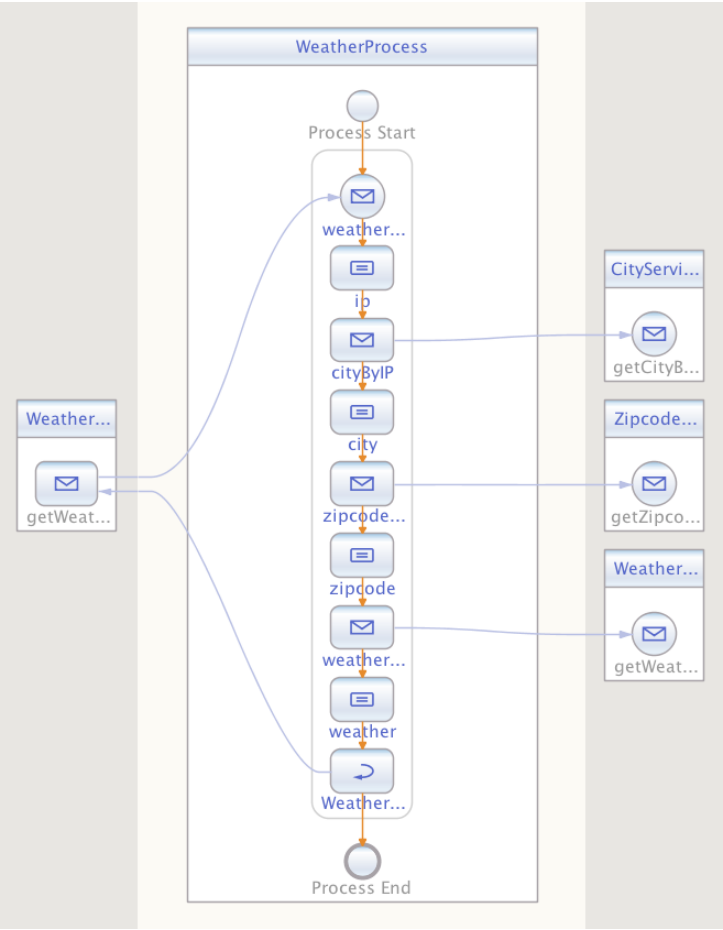


Figure 8.1: The abstract BPEL describing the orchestration of the services in the abstract WSDL.

We use the framework in order to instantiate the needed abstract process with concrete services. We aim at facilitating the selection of the services offering the needed functionality for each task, and satisfying the expected QoS together with the composition links. In the following, we explain our final use case that is studied in this thesis.

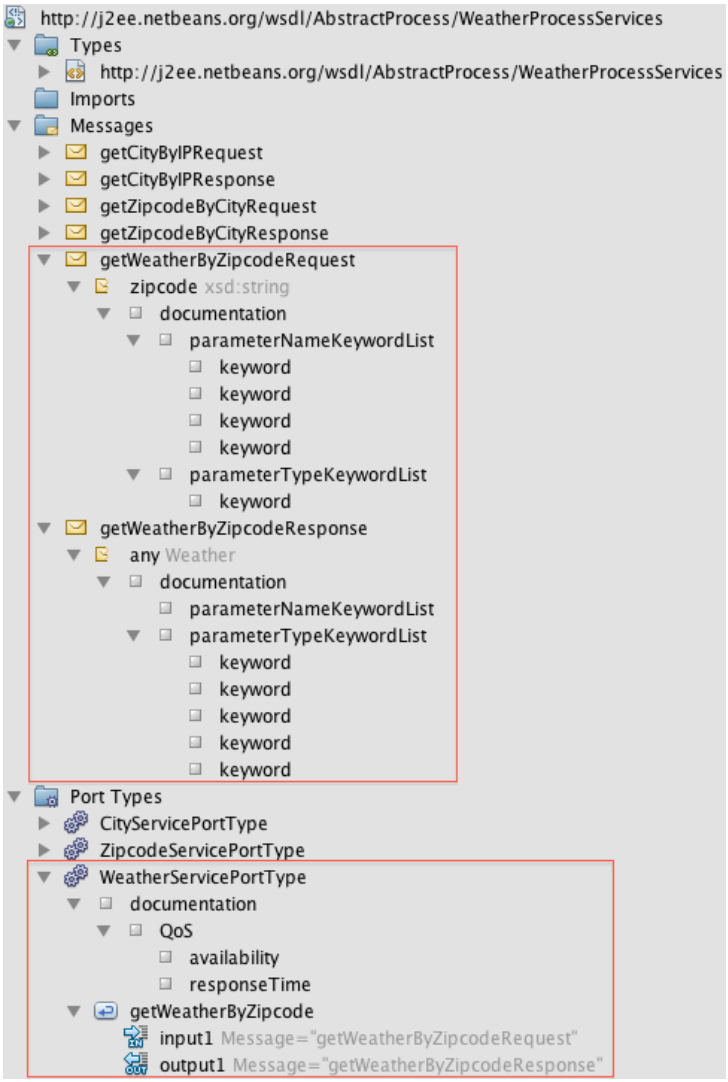


Figure 8.2: The abstract WSDL describing the needed services for the scenario in 1.

2 The Selection Framework: Use Case 3

Using our framework, a user can define an abstract process transparently (without a previous knowledge about the concrete services). The framework works on retrieving the set of services that best match the user's specified requirements, in order to instantiate the described process with concrete services, as well as identify backup services to ensure process continuity.

Below, we explain the framework and its functionality according to the components in each layer (see Figure 8.3).

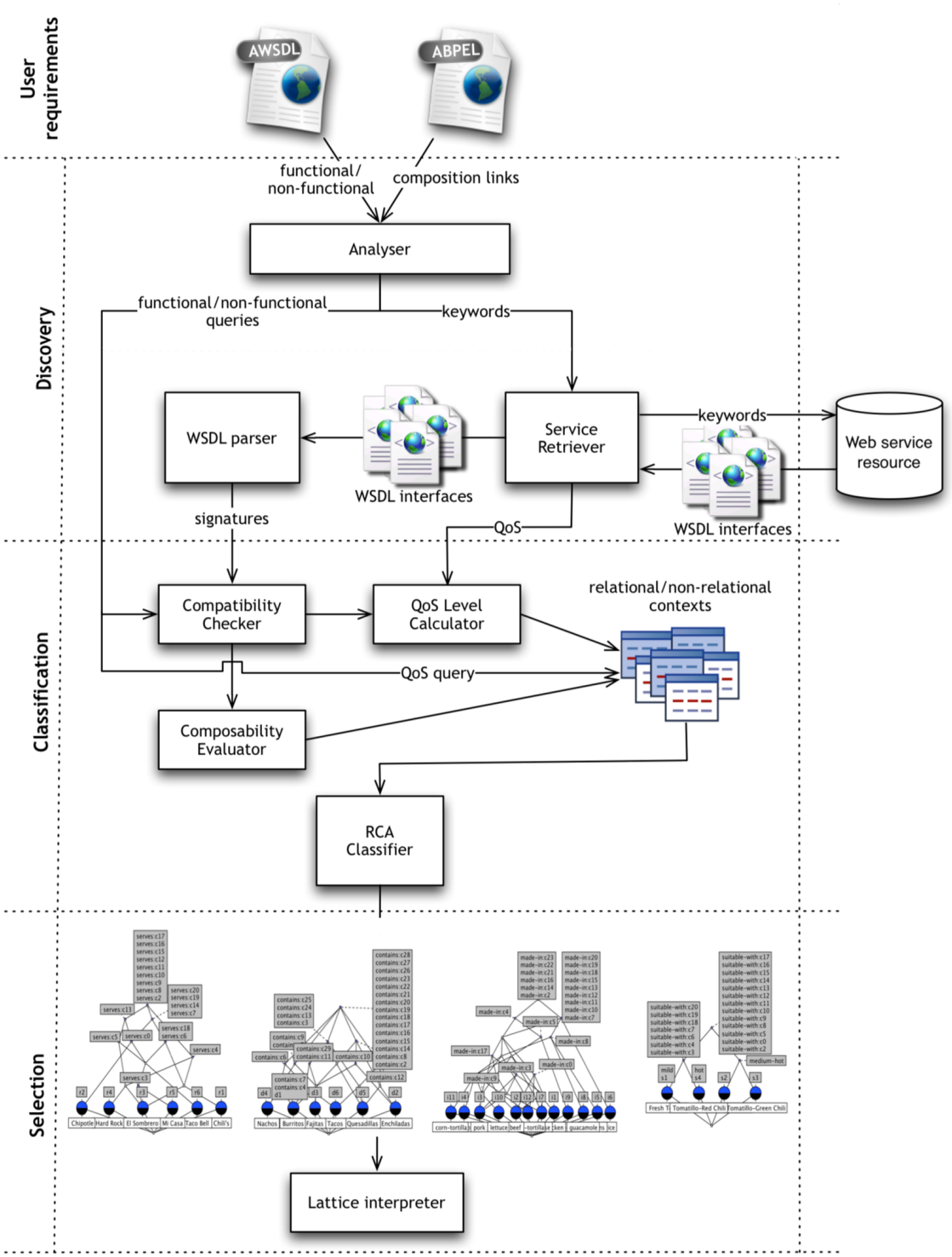


Figure 8.3: An overview of the approach's components.

2.1 Discovery Layer

The user starts by specifying the needed functionality (operations) along with the QoS for each service inside an abstract WSDL file (see Figure 8.2). The composition links are extracted from an abstract BPEL file defined by the user, which specifies the orchestration of the services described in the abstract WSDL (see Figure 8.1).

The analyzer extracts the keywords specifying the required parameters and passes them to the service retriever. It also extracts the QoS required levels and passes them as queries to be integrated into the RCA relational contexts. Concerning the abstract BPEL file, the analyzer works on extracting the composition links between the specified services.

Composing two Web services is finding two operations (one of each) that can be linked. Two operations can be linked when the output parameters of the first (source) matches one or more of the input parameters of the second (target). We can define two composition modes according to the coverage of the input parameters of a target operation, in addition to two other modes according to the needed adaptations. These composition modes are `Fully-Composable (FC)`, `Partially-Composable (PC)`, `Adaptable-Fully-Composable (AFC)`, and `Adaptable-Partially-Composable (APC)` (as described below).

Thus, the composition links represent the assignment of parameters of a source operation to the parameters of a target operation. Actually, each invoke activity has a pre and post assignments of parameters. The composition links can be recognized by regarding the invoke activity, together with the $copy from-to$ structure in the pre- and post- assign activities. The pre-assign describes the composition link of the service to be invoked with its preceding service in the orchestration, while the post-assign describes the composition link with the next service. For example, in Figure 8.4, we can see an invocation of the operation getZipcodeByCity, which has one input parameter $GetZipCodeByCityIn$ and one output parameter $GetZipcodeByCityOut$. From the pre-assign (named "city"), we recognize that the composition between the operation getZipcodeByCity and the operation getCityByIP has an FC mode. The extracted composition links are used afterwards to specify the relational contexts to be used, when generating the services relational lattices, as explained shortly after.

The service retriever receives the list of keywords specified for the parameters of each required operation from the analyzer. Then it retrieves a set of services together with the QoS properties that are supported by the Web service resource. In our case, the retriever gets the availability and response time values for each retrieved service. The retrieved WSDL interfaces are passed to the WSDL parser to be parsed, while the QoS values are passed to the QoS level calculator, which is explained in the classification layer section. The WSDL parser, extracts the operation signatures and passes them to the compatibility checker, which is also explained in the classification layer section below.

```
<invoke name="cityByIP" partnerLink="CityService" operation="getCityByIP"
        portType="tns:CityServicePortType"
        inputVariable="GetCityByIPIn" outputVariable="GetCityByIPOut"/>

<assign name="city">
    <copy>
        <from variable="GetCityByIPOut" part="city"/>
        <to variable="GetZipcodeByCityIn" part="city"/>
    </copy>
</assign>

<invoke name="zipcodeByCity" partnerLink="ZipcodeService" operation="getZipcodeByCity"
        portType="tns:ZipcodeServicePortType"
        inputVariable="GetZipcodeByCityIn" outputVariable="GetZipcodeByCityOut"/>

<assign name="zipcode">
    <copy>
        <from variable="GetZipcodeByCityOut" part="zipcode"/>
        <to variable="GetWeatherByZipcodeIn" part="zipcode"/>
    </copy>
</assign>

<invoke name="weatherByZipcode" partnerLink="WeatherService" operation="getWeatherByZipcode"
        portType="tns:WeatherServicePortType"
        inputVariable="GetWeatherByZipcodeIn" outputVariable="GetWeatherByZipcodeOut"/>
```

Figure 8.4: Pre and post assign for each invoke activity in BPEL.

2.2 Classification Layer

The components of this layer work on filtering the retrieved services, to ensure their compatibility with the requested functionality. Then, the obtained QoS values are processed and the composability between the retrieved services is evaluated, in order to build RCA contexts. From the built contexts, the service are classified into relational concept lattices according to process composition links, and after integrating the QoS queries. The generated lattices are then used to realize an efficient selection of services that best match and satisfy the user specified requirements. In this layer, we have the following components:

2.2.1 Compatibility Checker

This component checks whether a service provides an operation that can satisfy the corresponding task. An operation satisfies a task when it contains the requested input/output parameters names. We verified this by using the Jaro-Winkler string distance measure [80]. This measure calculates the similarity between two strings, and has proved to be efficient and accurate [81]. Although, this is a primary solution for similarity measuring, which we plan to improve in future work. By doing so, we discovered three possible cases:

- *compatible*, there exists one operation at least that satisfies the corresponding task and has the same parameters types; or it may become:
- *adaptable compatible*, meaning that none of the satisfying operations has the same parameter types (either for input, or output, or both), thus type adaptations need to be done; otherwise:
- *incompatible*, the service does not satisfy the corresponding task.

The compatibility checker reduces the number of the retrieved services, by omitting the incompatibles ones, while keeping a detailed list of the compatible ones together with their satisfying operations.

Once we identified the compatible services, we can measure the composition mode and the QoS levels using the two following components.

2.2.2 QoS Level Calculator

This component takes into consideration the QoS values for all the sets of compatible services. It extracts these values from the ones returned by the service retriever, according to the list returned by the compatibility checker. Web services have many QoS attributes and different ranges of numerical values for each one of these attributes. In order to have a better overview of these values, we apply a statistical technique called BoxPlot++ (Annex A) to cluster the convergent values together. The BoxPlot++ is an extension of the original boxplot [82] technique. It takes as input a given set of numerical values, and produces one to seven corresponding levels of values: `L = {BadOutlier, VeryBad, Bad, Medium, Good, VeryGood, GoodOutlier}`[1]. The technique is applied on each QoS attribute. Then, it generates for each set of services a non-relational context having all of its QoS attributes levels. These contexts are exploited afterwards by the RCA classifier, in order to classify the services according to these different QoS levels.

2.2.3 Composability Evaluator

We define four composition modes as mentioned before. We list them again as follows:

- `Fully-Composable (FC)`, when a source operation covers by its outputs all of the expected inputs of a target operation;
- `Partially-Composable (PC)`, when one or more input parameters of a target operation are not covered;
- `Adaptable-Fully-Composable (AFC)`, when the source and target operations have an `FC` mode, but need some type adaptations either for the output of the source or the input of the target, or both of them; and
- `Adaptable-Partially-Composable (APC)`, similar to `AFC` but when having a `PC` mode.

The composability evaluator determines the mode of composition between the services according to the specified orchestration. Then, it generates four relational contexts (corresponding to the composition modes). These contexts are exploited by the RCA classifier to clarify which services can be composed and following which modes.

2.2.4 RCA Classifier

This component takes into consideration the relational contexts of composition modes and the non-relational contexts of QoS levels. It also uses the QoS queries extracted by the analyzer component, which are integrated into the corresponding contexts.

Finally, the RCA classifier [64] generates all the corresponding service lattices and passes them to the final component.

[1] Available online: http://www.lirmm.fr/~azmeh/tools/BoxPlot.html

2.3 Selection Layer

By integrating the non-functional queries into the contexts, they appear inside the concepts of the corresponding lattices. This enables this component to locate the services that satisfy the queries and to navigate between the different solutions. These services are present at the sub-concepts of the queries concepts. This is better illustrated in Section 9-4.2.

In case of multiple possible selections of Web services having approximating levels of QoS, can use the following technique for performing an optimal selection. This technique is an enhancement for the lattice interpreter. We call it optimizing by triangles, and it works as follows: From the extracted services, we can have several combinations that satisfy our desired orchestration. We have to select the combination of services that meet the optimal compromise of QoS levels and composability. In order to meet this issue, we propose to represent the services by vectors. A vector per set of services, on which, services are ordered according to their QoS. We also define a vector for composability levels. Each composition can be regarded as a triangle, having a head corresponding to the composability level, and the two other heads corresponding to the pair of services to be composed, as shown in Figure 8.5.

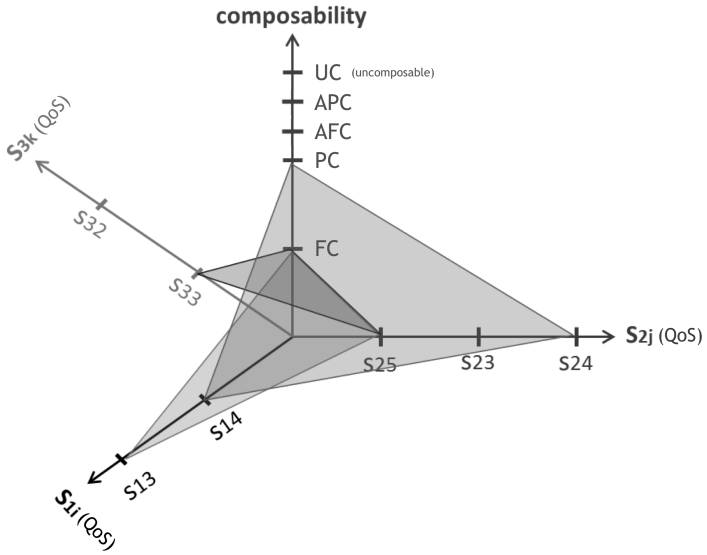


Figure 8.5: Optimizing the service selection using triangles.

The optimal selection would be the triangle that has the minimal area, in case of a two services composition. Otherwise, it will be the minimal sum of the services triangles according to an orchestration. Thus, for example, by regarding the triangles in Figure 8.5, we notice that the triangle (FC, s_{25}, s_{33}) represents an optimized composition between S_{2j} and S_{3k} (corresponding to Task2 and Task3, respectively). Accordingly, if we consider the triangle (FC, s_{13}, s_{25}) that shares an edge with the previous triangle, it represents a composition between S_{1i} and S_{2j} (corresponding to Task1 and Task2, respectively). These two triangles together may be an optimized combination for the required orchestration, after comparing them with all the existing triangles[2].

[2]We haven't shown all the triangles in Figure 8.5, for the sake of simplicity.

3 Summary

In this chapter, we presented the third and final use case of our framework. It facilitates building Web service orchestrations, by enabling a straightforward identification of composable services that satisfy the user's functional and non-functional requirements.

Users can specify their required functionality by an AWSDL file, where they can also specify the needed QoS for each required service. The desired service orchestration can be defined by an ABPEL file, which specifies the interaction between the abstract services inside the AWSDL file. Using our framework, we can analyze these requirements, retrieve compatible services, then classify them using RCA. The resulting RCA-based lattices group services that have common QoS and composition levels. They enable the selection of the required services, by performing lattice navigation by QoS queries (we explained the navigation by query algorithm in Chapter 4). For an optimized selection, we proposed a simple triangles-approach that works on identifying the services offering the best compromise for QoS and composability. We envisage to look at other techniques like skylines operators [83].

In the next chapter, we demonstrate our conducted experiments for the current use case, as well as for previously presented use cases.

Chapter 9

Experimentation

Contents

1 Introduction

In the previous chapters, we presented several theoretical descriptions of our framework through three distinct use cases. In this chapter, we present the experiments that we conducted to validate each of the described use cases. We used for each experiment, real sets of Web services that were retrieved using the *Seekda* [8] and *Service-Finder* [14] Web service search engines. We present each experiment on two parts: a methodology part, where we explain the steps that we followed for conducting the experiment; and a validation part, where we show and discuss the obtained results.

In Section 2, we validate our automatic service tagger, which is a component of the selection of Web services by tags, presented in Chapter 6. In Section 3, we validate the selection of Web services by functionality, presented in Chapter 7. In Section 4, we validate the selection of Web services according to multi- user requirements, presented in Chapter 8.

2 Web Service Selection by Tags

This section provides a validation of our automatic tagger (presented in Chapter 6) using real Web services retrieved from Seekda. This engine allows its users to manually assign tags to its indexed services. Using the service retriever, we retrieved a set of 146 WSDL files together with

their associated tags.

We cleaned the tags of the training corpus, as explained in Section 6-2.2.2. Finally, our corpus $\mathcal{T}$ contained 146 WSDL files and 1393 tags (average of 9.54 tags per WSDL). An analysis of $\mathcal{T}$ showed that about 35% of the user tags are already contained in the WSDL files.

2.1 Methodology

We carried out our experiments on three stages. In the first one, the trained classifier is applied on the training corpus $\mathcal{T}$ and its output is compared with the tags given by Seekda users (obtained as described in Section 6-2.2.2).

After having conducted the first experiment, a manual assessment of the tags produced by our approach revealed that many tags not assigned by the user seemed highly relevant. This phenomenon has also been observed in several human evaluations of Kea [84, 85], that inspired our approach. It occurs because tags assigned by the users are not the *absolute truth*. Indeed, it is very likely that users have forgotten many relevant tags, even if they were in the service description. To show that the real efficiency of our approach is better than the one computed in the first experiment, we perform a second experiment. In this experiment, we manually augmented the user tags of our corpus with additional tags we found relevant and accurate by analyzing the WSDL descriptions of the services. In the final experiment, we enriched the user tags as well as our automatically extracted tags with semantically related tags using WordNet.

Metrics: In the evaluation, we used precision and recall. First, for each web service $s \in \mathcal{T}$, where $\mathcal{T}$ is our training corpus, we consider: A the set of tags produced by the trained classifier, M the set of the tags given by Seekda users and W the set of words appearing in the WSDL. Let $I = A \cap M$ be the set of tags assigned by our classifier and Seekda users. Let $E = M \cap W$ be the set of tags assigned by Seekda users present in the WSDL file. Then we define $precision(s) = \frac{|I|}{|A|}$ and $recall(s) = \frac{|I|}{|E|}$, which are aggregated in $precision(\mathcal{T}) = \frac{\sum_{s \in \mathcal{T}} precision(s)}{|\mathcal{T}|}$ and $recall(\mathcal{T}) = \frac{\sum_{s \in \mathcal{T}} recall(s)}{|\mathcal{T}|}$. The recall is therefore computed over the tags assigned by Seekda users that are present in the descriptions of concerned services. We did not compute the recall for the WordNet extracted tags, because these tags may not be present in the WSDL descriptions.

2.2 Validation

Figure 9.1 (left) gives results for the first experiment where the output of the classifier is compared with the tags of Seekda users, while in Figure 9.1 (right), enriched tags of Seekda users are used in the comparison (curated corpus). In this figure, our approach is called *ate* (*Automatic Tag Extraction*). To clearly show the concrete benefits of our approach, we decided to include in these experiments a straightforward (but fairly efficient) technique. This technique, called *tfidf* in Figure 9.1, consists in selecting, after the application of our text-mining techniques, the five candidate tags with the highest *tfidf* weight.

In Figure 9.1 (left), the precision of *ate* is 0.48. It is a significant improvement compared to the *tfidf* method that achieves only a precision of 0.28. Moreover, there is no significant dif-

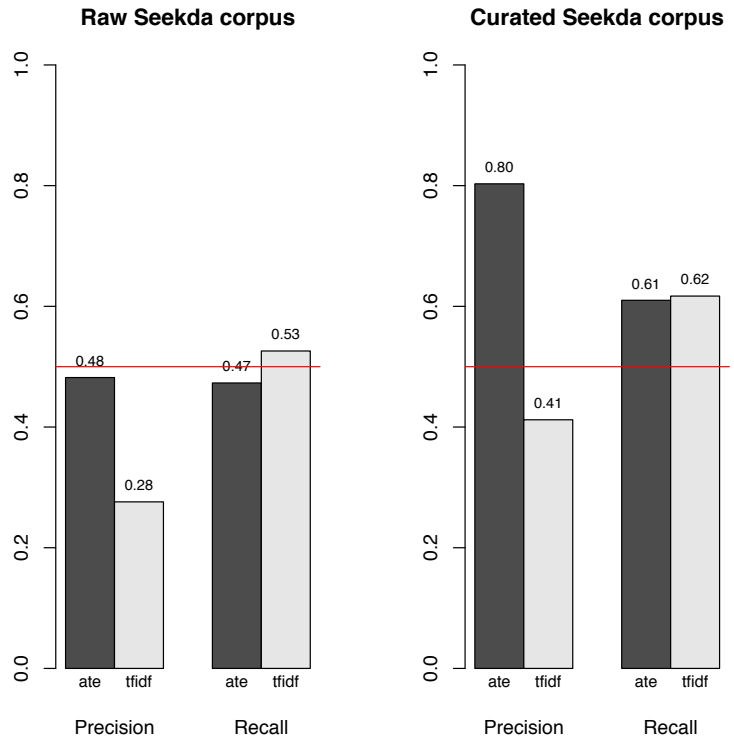


Figure 9.1: Results on the original and manually curated Seekda corpus

ference between the recall achieved by the two methods. To show that the precision and recall achieved by *ate* are not biased by the fact that we used the training corpus as a testing corpus, we performed a 10 folds cross-validation. In a 10 folds cross-validation, our training corpus is divided in 10 parts. One is used to train a classifier, and the 9 other parts are used to test this classifier. This operation is done for every part, and then, the average recall and precision are computed. The results achieved by our approach using cross-validation ($precision = 0.44$ and $recall = 0.42$) are very similar to those obtained in the first experiment.

In Figure 9.1 (right), we see that the precision achieved by *ate* in the second experiment is much better. It reaches 0.8, while the precision achieved by the *tfidf* method increases to 0.41. The recall achieved by the two methods remains similar. The precision achieved by our method in this experiment is good. Only 20% of the tags discovered by *ate* are not correct. Moreover, the efficiency of *ate* is significantly higher than *tfidf*.

WordNet for semantically related tags

The classifier that we built determines whether a word in a WSDL file is a tag or not. Thus, it extracts the tags appearing inside the WSDL files only. This way, we miss some other interesting tags like associated words or synonyms. In order to solve this issue, we used the WordNet lexical database [75]. In WordNet a word may be associated with many synsets (synonym sets), each

corresponding to a different sense of a word.

Our corpus consists of 146 WSDL files, each of which is assigned two sets of tags: user tags and our automatically extracted tags. Our objective is to enrich each set of tags with semantically similar words extracted from WordNet. Thus, for each tag we identify the possible senses and the synonyms set related to each sense. We add the extracted synonyms to the corresponding set of tags, and we perform some experiments to evaluate the obtained tags.

Evaluation after using WordNet

We enriched the tags sets with semantically similar words extracted using the WordNet, as described above. We recalculated the precision value, considering these new sets of enriched user tags and automatically extracted tags. The precision value has increased by 9%, reaching the value of 89% of correctness. Thus, using the WordNet has improved the precision value and enriched the services with tags that are not necessarily present in the WSDL descriptions.

Observation

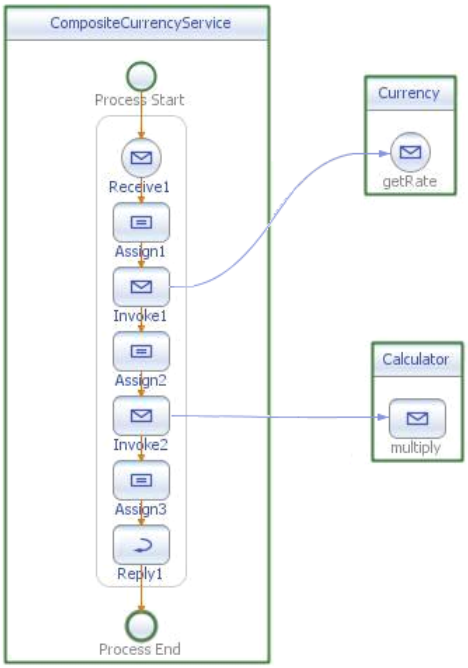
Our experiments use real world services, obtained from the Seekda service search engine. Our training corpus contains services extracted randomly with the constraint that they contain at least 5 user tags. We assumed that Seekda users assign correct tags. Indeed, our method admits some noise but would not work if the majority of the user tags were poorly assigned. In the second experiment, we manually added tags we found relevant by examining the complete description and documentation of the concerned services. Unfortunately, since we are not "real" users of those services, some of the tags we added might not be relevant.

3 Web Service Selection by Functionality

In this section, we validate our scenario presented in Chapter 7. We demonstrate the use of service lattices for both building composite Web services and supporting them with backup services in a real world context.

We consider the *Composite Currency Service (CCS)* (presented in Chapter 7), which is composed of two Web services: a currency converter service *Currency* and a calculation service *Calculator*. The *Currency* service offers an operation that returns the exchange rate between two entered currencies: *getRate(fromCurr,toCurr)*. The *Calculator* service offers an operation that calculates the multiplication of two entered numbers: *mul(a,b)*. We compose these two operations in order to build the composite currency service that converts a given amount from one currency to another. We describe a service composition using the Business Process Execution Language (BPEL) [86]. We use the BPEL editor of *NetBeans IDE* [87] to design and describe the specified *CompositeCurrencyService* as shown in Figure 9.2.

We used the *Seekda* and *Service-Finder* to search for the needed services. We describe this scenario on two parts: first we illustrate the use of the approach, then we validate it.



Recieve1: a start request performed by a requestor of the *CompositeCurrencyService*
Assign1: assigning values to the *getRate* operation's input parameters
Invoke1: invoking the *getRate* operation from the *Currency* service
Assign2: assigning the *getRate* operation's output value & an amount to the *mul* operation
Invoke2: invoking the *mul* operation from the *Calculator* service
Assign3: assigning the *mul* operation's output value to the *CompositeCurrencyService* response
Reply1: returning the final response of the *CompositeCurrencyService*

Figure 9.2: The composite currency service.

3.1 Methodology

We use a set of services for currency conversion shown in Table 9.1 and another set for calculation as shown in Table 9.2. We limit the number of services in this example, in order to simplify it and clearly explain the idea of lattice use.

For dealing with this illustration, we assess manually the similarity for the obtained services' operations of each set. This is achieved by comparing operation signatures (operation names, parameter names and types). Using the operations lattice and its square concepts, we identify the following groups of mutually similar operations for the currency services in Table 9.1:

- $\{op_{11}, op_{21}, op_{31}, op_{51}\}$ that we label $(CR : 11, 21, 31, 51)$;
- $\{op_{32}, op_{42}, op_{52}, op_{61}, op_{73}, op_{82}, op_{91}\}$ labelled $(CC : 32, 42, 52, 61, 73, 82, 91)$;
- $\{op_{33}, op_{81}\}$ labelled $(CS : 33, 81)$;
- $\{op_{41}\}$ labelled $(R : 41)$;
- $\{op_{72}\}$ labelled $(FC : 72)$;
- $\{op_{71}\}$ labelled $(CF : 71)$.

Table 9.1: The set of currency converter services.

Services	Id	Operations	Id
CurrencyConverter	ws_1	GetConversionRate(fromCurrency,toCurrency)	op_{11}
CurrencyConvertor	ws_2	ConversionRate(FromCurrency,ToCurrency)	op_{21}
DOTSCurrencyExchange	ws_3	GetExchangeRate(ConvertFromCurrency,ConvertToCurrency)	op_{31}
		ConvertCurrency(Amount,ConvertFromCurrency,ConvertToCurrency)	op_{32}
		GetCountryCurrency(Country)	op_{33}
CurrencyRates	ws_4	GetRate(CurrencyCode)	op_{41}
		GetConversion(FromCurrencyCode,ToCurrencyCode)	op_{42}
RadixxFlights	ws_5	GetExchange(FromCurrency,ToCurrency)	op_{51}
		ConvertCurrency(Amount,FromCurrency,ToCurrency)	op_{52}
rates	ws_6	Convert(CurrencyFrom,CurrencyTo,ValueFrom)	op_{61}
Conversion	ws_7	CelciusToFahrenheit(fCelsius)	op_{71}
		FahrenheitToCelcius(fFahrenheit)	op_{72}
		Currency(fValue,sFrom,sTo)	op_{73}
CurConvert	ws_8	GetCurrencySign(CountryName)	op_{81}
		ConvertCurrency(FromCountry,ToCountry,Amount)	op_{82}
ConverterService	ws_9	Convert(sourceCurrency,targetCurrency,value)	op_{91}

Table 9.2: The set of calculation services.

Services	Id	Operations	Id
Calc	ws_1	add(a,b)	op_{11}
		div(a,b)	op_{12}
		mul(a,b)	op_{13}
		pow(b,a)	op_{14}
		sub(a,b)	op_{15}
Service	ws_2	add(a,b)	op_{21}
		sqrt(a)	op_{22}
		sub(a,b)	op_{23}
MathService	ws_3	Add(A,B)	op_{31}
		Divide(A,B)	op_{32}
		Multiply(A,B)	op_{33}
		Subtract(A,B)	op_{34}
CalculatorService	ws_4	add(y,x)	op_{41}
		divide(denominator,numerator)	op_{42}
		multiply(y,x)	op_{43}
		subtract(y,x)	op_{44}
CalcService	ws_5	Divide(A,B)	op_{51}
		Multiply(A,B)	op_{52}
		OperationAdd(A,B)	op_{53}
		Subtract(A,B)	op_{54}
Calculate	ws_6	Add(dbl1,dbl2)	op_{61}
		Divide(dbl1,dbl2)	op_{62}
		Multiply(dbl1,dbl2)	op_{63}
		Subtract(dbl1,dbl2)	op_{64}

We extract also the groups of mutually similar operations for the calculation services in Table 9.2, and they are as follows:

- $\{op_{15}, op_{23}, op_{34}, op_{44}, op_{54}, op_{64}\}$ labelled $(sub : 15, 23, 34, 44, 54, 64)$;
- $\{op_{11}, op_{21}, op_{31}, op_{41}, op_{53}, op_{61}\}$ labelled $(add : 11, 21, 31, 41, 53, 61)$;
- $\{op_{13}, op_{33}, op_{43}, op_{52}, op_{63}\}$ labelled $(mul : 13, 33, 43, 52, 63)$;
- $\{op_{12}, op_{32}, op_{42}, op_{51}, op_{62}\}$ labelled $(div : 12, 32, 42, 51, 62)$;
- $\{op_{14}\}$ labelled $(pow : 14)$;

– $\{op_{22}\}$ labelled $(sqrt : 22)$.

These extracted groups of similar operations lead to a binary context for each set of services as shown in Tables 9.3 and 9.4.

Table 9.3: The formal context corresponding to the currency converter services.

	(CR:11,21,31,51)	(CC:32,42,52,61,73,82,91)	(CS:33,81)	(R:41)	(FC:72)	(CF:71)
ws_1	×					
ws_2	×	×				
ws_3	×	×	×			
ws_4		×		×		
ws_5	×	×				
ws_6		×				
ws_7		×			×	×
ws_8		×	×			
ws_9		×				

Table 9.4: The formal context corresponding to the calculator services.

	(sub:15,23,34,44,54,64)	(add:11,21,31,41,53,61)	(mul:13,33,43,52,63)	(div:12,32,42,51,62)	(pow:14)	(sqrt:22)
ws_1	×	×	×	×	×	
ws_2	×	×				×
ws_3	×	×	×	×		
ws_4	×	×	×	×		
ws_5	×	×	×	×		
ws_6	×	×	×	×		

We generate the two corresponding lattices as shown in the right side of Figure 9.3. We can exploit these service lattices to build our composite service as well as to support it with backup services. Thus, we decide to select operation $op_{11} : (CR : 11)$ from service ws_1 for exchange rate (currency lattice), and operation $op_{13} : (mul : 13)$ from service ws_1 for multiplication (calculation lattice). From these lattices (Figure 9.3), we can also extract some backup services for our composite service according to the selected operations. For example, we used operation $op_{11} : (CR : 11)$ from service ws_1, which has 3 equivalent operations: $op_{21} : (CR : 21)$, $op_{31} : (CR : 31)$ and $op_{51} : (CR : 51)$ appearing clearly in the lattice. This means that if service ws_1 breaks down, we can replace it by any of the services ws_2 (equivalent to ws_1 being in the same concept), ws_3 or ws_5 (services introduced in subconcepts).

Moreover, if we go down in the lattice, we get the set of services that provide the operations used together with extra operations, like service ws_5 and service ws_3. They can help if the composite service evolves and needs other operations. In the same way, we can extract the backup services for the calculation service ws_1 that we are using. According to the calculation service lattice, service ws_1 as a whole set of operations cannot be replaced by any service. But, regarding the multiplication functionality, $op_{13}(mul : 13)$, it can be replaced by operations $op_{33} : (mul : 33)$, $op_{43} : (mul : 43)$, $op_{52} : (mul : 52)$, and $op_{63} : (mul : 63)$, which are offered by

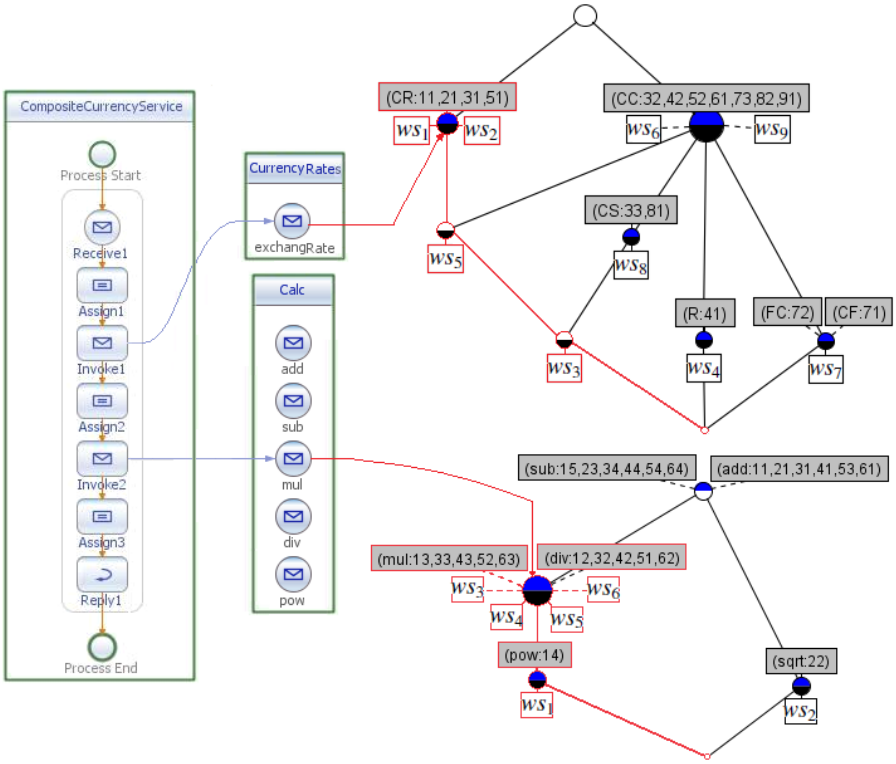


Figure 9.3: The composite currency service, supported by backups from the service lattices.

services ws_3, ws_4, ws_5, and ws_6 respectively. This gives us a replacement possibility in case of unavailability of ws_1 in the framework of the composite currency service.

3.2 Validation

We validate our approach using the entire number of retrieved *Calculator* and *Currency* services[1] from *Service − Finder* and *Seekda*. We queried *Service − Finder* to collect service endpoints (addresses), then we downloaded the corresponding WSDL interfaces via *Seekda*. For the *Calculator* service, we searched using *multiply* as keyword. This returned a set $WS1$ of 29 services, among which we found one unrelated service.

For the *Currency* service, we used a combination of the following keywords *exchange*, *rate*, *currency*, *converter*. After eliminating the repeated services, we found a set of 81 services. From this set, we also eliminated the services that we were unable to parse. This resulted in a set $WS2$ of 64 services.

We parsed each service of the two sets (WSDL parser[2]), to extract its operation signatures. The $WS1$ set has a total of 142 operations, while $WS2$ has 935 operations.

[1] Retrieved services: http://www.lirmm.fr/~azmeh/icfca11/CaseStudy.html
[2] Available online: http://www.lirmm.fr/~azmeh/tools/WsdlParser.html

In order to calculate the $SimMat$ (explained in Section 7-2.2.1) for both sets of services, we make use of $Jaro - Winkler$ [80] similarity measure, to assess the similarity between the extracted signatures according to each set. This metric gave convenient similarity values that were calculated efficiently, compared to another tested technique that used a combination of syntactic and semantic metrics. After a number of experiments, we found that a relatively pertinent similarity value starts from 80%. By applying this threshold on the $SimMat$, we obtained the binary $SimCxt$ corresponding to each set.

We tried to compute the lattices corresponding to each $SimCxt$ using Galicia [64]. The lattices could not be generated due to an "out of memory" error (explosion of the number of generated concepts). Therefore, we computed the Galois Sub Hierarchy (GSH) (see Chapter 4), (order induced by attribute and object concepts). Using GSH, we obtained a suborder of 155 concepts for $SimCxt$ (142×142) and another suborder of 1724 concepts for $SimCxt$ (935×935). The second suborder may be reduced depending on the functionality filtering techniques.

Hereby, we show our analysis for $WS1$. From the GSH calculated for $SimCxt$ (142×142), we extracted 65 square concepts. Among these 65 square concepts, we had 13 non-trivial concepts and 52 concepts reduced to one operation. Each square concept represents a functionality, for example: $c82$ represents the *multiply* functionality. It contains $\{op15.2, op18.3, op2.3, op6.3, op8.2\}$, which are mutually substitutable operations for calculating the multiplication of two numbers.

Afterwards, we constructed the lattice of services (as objects) and these square concepts (as attributes). The generated lattice is shown in Figure 9.4, and contains 21 concepts. By regarding the right half of the lattice, we can notice services that can be entirely replaced by other ones. For example: if we consider $ws15$, it contains the *multiply* functionality (being a subconcept of $c82$). This service can be replaced by three other services: $ws18$, $ws6$ and $ws2$.

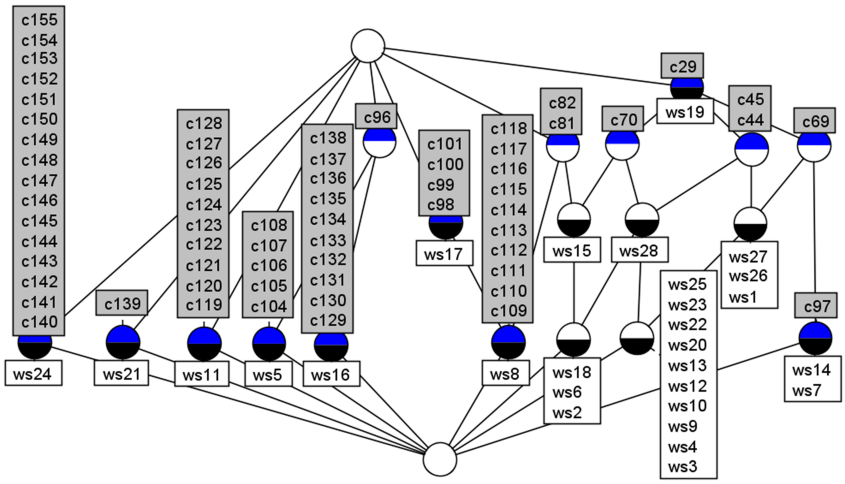


Figure 9.4: The lattice corresponding to the *Calculator* services set.

In the same way, we analyze the GSH calculated for $SimCxt$ (935×935). We could not take a screen shot of this GSH, because it was huge and could not fit. Although, we show the corresponding service lattice in Figure 9.5.

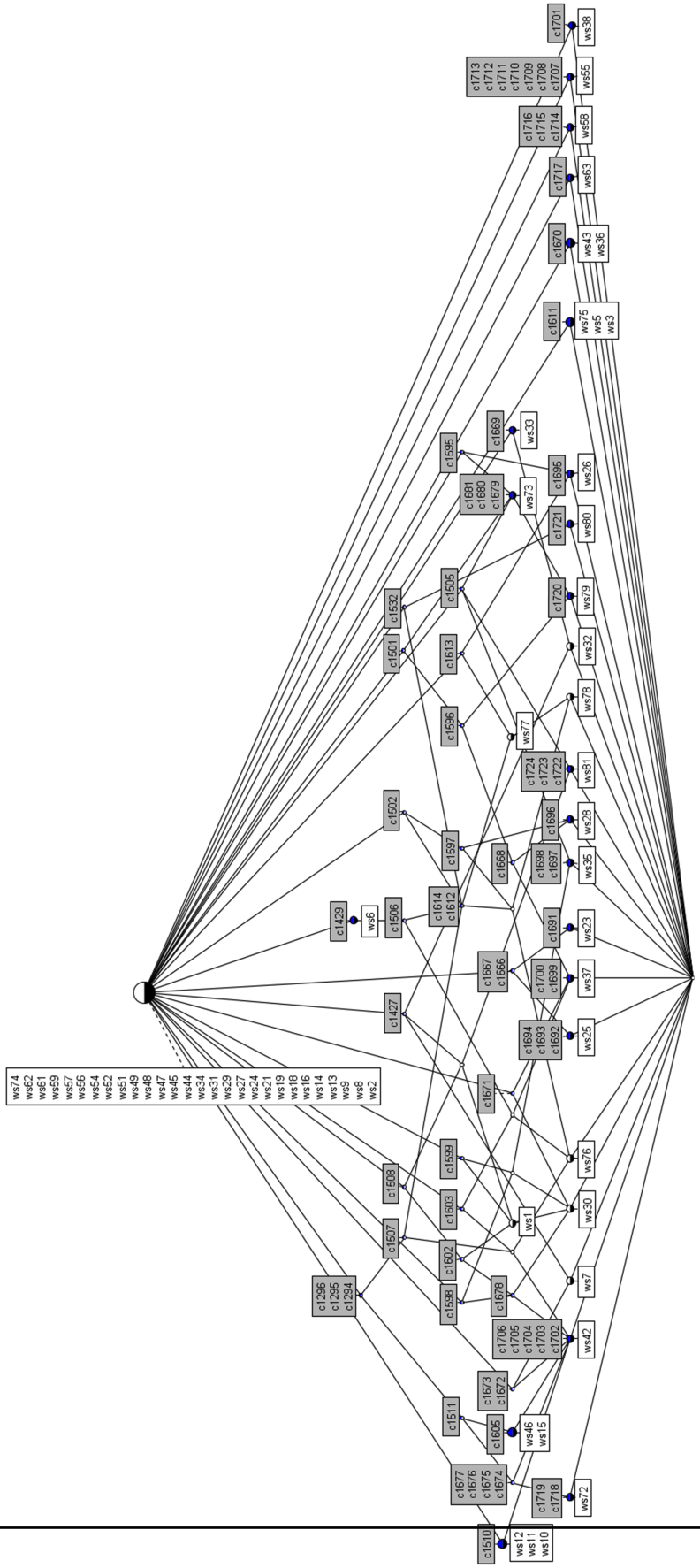


Figure 9.5: The lattice corresponding to the *Currency* services set.

4 Web Service Selection According to Multi- User Requirements

In this section, we validate our final use case, which is presented in Chapter 8. We reconsider the scenario about the abstract weather process (Figure 9.6) that we presented before. The considered abstract weather process is supposed to provide the weather information for a given ip address.

It orchestrates the invocation of three operations: ipToCity, cityToZipcode, and zipcodeToWeather. The needed services are described inside an abstract WSDL file, which is illustrated in Figure 9.7. It describes three PortTypes (services): CityServicePortType, ZipcodeServicePortType, and WeatherServicePortType. Each service has an expected QoS level for the attributes availability and response time. In Figure 9.8, we can see the description specified for the CityServicePortType. We can notice its operation getCityByIP, which is specified by it input/output keywords list.

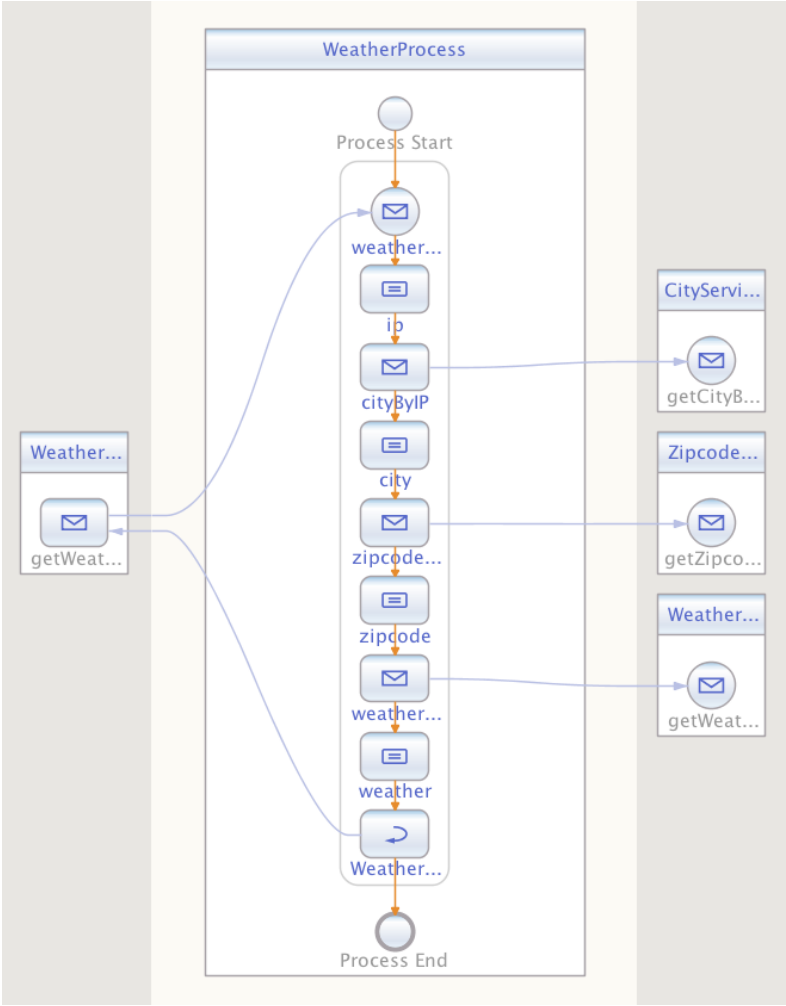


Figure 9.6: The abstract BPEL describing the orchestration of the services in the abstract WSDL.

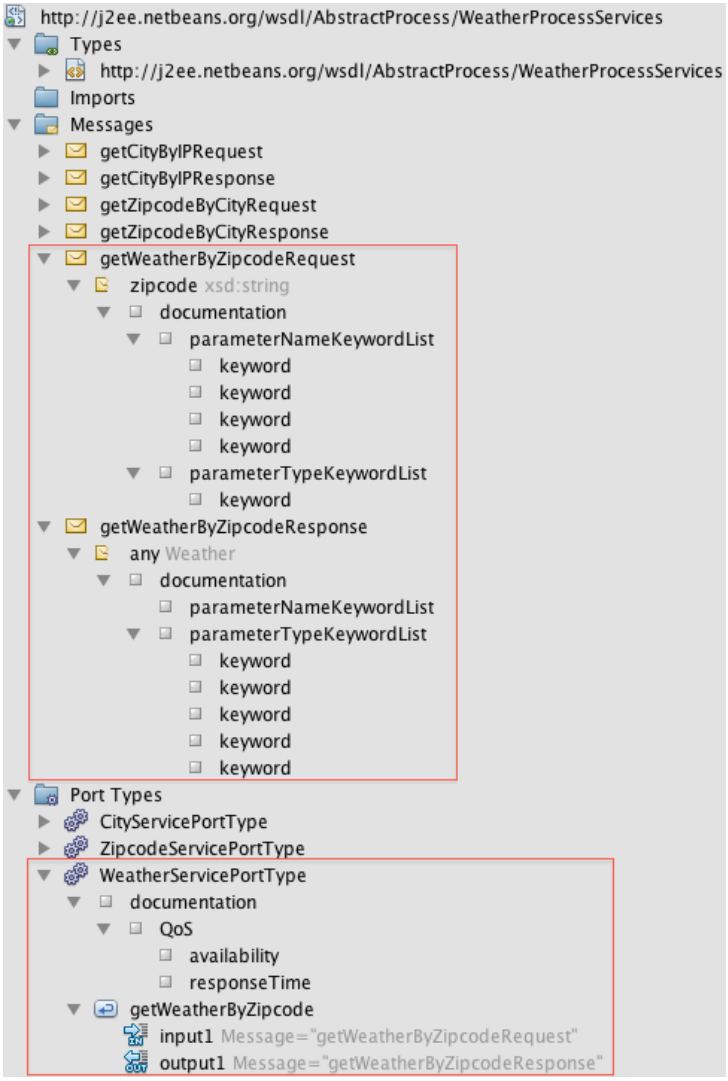


Figure 9.7: The abstract WSDL describing the needed services for the scenario in 1.

4.1 Methodology

From this abstract WSDL file, the experiments are conducted on four steps, according to the framework's layers, as follows:

1. collecting the services: we use the set of keywords describing each operation to search and retrieve sets of corresponding Web services. We make use of Service-Finder to collect a set of corresponding endpoints (WSDL addresses). This engine also provides us with values of two QoS attributes: availability `(A)` and response time `(RT)`. We download the corresponding WSDL files after omitting repeated and invalid endpoints;

2. filtering the services: in this step, we parse the WSDL files using the WSDL parser (Section 8-2.2.2) and remove the invalid ones (Filter1). Then we calculate the compatible ones (Filter2) using the Compatibility checker (Section 8-2.2.1);

```
<message name="getCityByIPRequest">
    <part name="ip" type="xsd:string">
        <documentation>
            <parameterNameKeywordList>
                <keyword value="ip" />
                <keyword value="ipAddress" />
            </parameterNameKeywordList>

            <parameterTypeKeywordList>
                <keyword value="string" />
            </parameterTypeKeywordList>
        </documentation>
    </part>
</message>

<message name="getCityByIPResponse">
    <part name="city" type="xsd:string">
        <documentation>
            <parameterNameKeywordList>
                <keyword value="city" />
                <keyword value="cityName" />
            </parameterNameKeywordList>

            <parameterTypeKeywordList>
                <keyword value="string" />
            </parameterTypeKeywordList>
        </documentation>
     </part>
</message>

<portType name="CityServicePortType">
    <documentation>
        <QoS>
            <availability level="Good" />
            <responseTime level="Good" />
        </QoS>
    </documentation>
    <operation name="getCityByIP">
        <input name="input1" message="tns:getCityByIPRequest"/>
        <output name="output1" message="tns:getCityByIPResponse"/>
    </operation>
</portType>
```

Figure 9.8: The CityServicePortType description inside the abstract WSDL.

3. calculating the composition modes and the QoS levels: using the Composability evaluator (Section 8-2.2.3) and the QoS level calculator (Section 8-2.2.2), for the compatible sets of services. Then generating the corresponding contexts;

4. generating the corresponding lattices using the RCA classifier (Section 8-2.2.4): by taking the contexts formed in the previous step and integrating the QoS queries.

4.2 Validation

1. Collecting Services: We show in Table 9.5 each described operation together with its keywords, the number of obtained endpoints, the number of retrieved WSDL files, and the sets identifiers. In this step, we make use of the requirements analyzer and service retriever components (Section 8-2.2.1).

Table 9.5: Summary of the retrieved services.

Task	Keywords	#Endpoints	#Services	SetID
1	{ip,ipAddress}, {city,cityName}	94	94	WS1.i
2	{city,cityName}, {zip,zipcode,postal,postalcode}	768	760	WS2.j
3	{zip,zipcode,postal,postalcode}, {weather,weatherInfo,forecast, weatherForecast,weatherReport}	39	37	WS3.k

2. Filtering the Services: In Table 9.6, we can see the resulting number of filtered services for each set.

Table 9.6: The number of filtered services for each set.

	WS1.i	WS2.j	WS3.k
Filter1 (Valid)	94	748	37
Filter2 (Compatible)	17	96	21

3. Composability and QoS: The calculated composition modes for the compatible sets of services as well as their QoS levels are shown in Table 9.7.

Table 9.7: Number of services per composition mode.

	WS1.i'	WS2.j'	WS3.k'
# FC services	12	3	12
# PC services	4	89	4
# AFC services	2	1	11
# APC services	0	3	2

The resulting composition modes and QoS levels are organized into non-relational and relational contexts, and are used to classify the services in the next step. In Table 9.8, we show the non-relational context of services corresponding to the first required service specified by *CityServicePortType*, described by their QoS levels. In Table 9.9, we show the relational context representing the fully composable (FC) mode, between services of WS1 and services of WS2.

4. RCA-Based Classification: The queries that we choose in this scenario are specified to be `Good A` and `Good RT` levels for each task in the process. They are integrated into the contexts formed in the previous step. We also require an `FC` composition mode for both `(Task1,Task2)` and `(Task2,Task3)` couples. The generated lattices are illustrated in Figure 9.9.

These lattices are finally interpreted by the lattice interpreter (Section 8-2.3), considering two rules:

- in each lattice, the services satisfying the corresponding query (QoS and composition) appear in the sub-concepts of the concept where the query appears. Example: the services in `c0 (WS1.i)` satisfy `Query1`;
- the services located closer to the bottom of a lattice offer better QoS levels than the farther ones, for example: in the lattice `(WS2.j)`, the service `WS2.8` is better than service `WS2.198`

WS1.i	BadOutlier A	VeryBad A	Bad A	Medium A	Good A	VeryGood A	GoodOutlier A	BadOutlier RT	VeryBad RT	Bad RT	Medium RT	Good RT	VeryGood RT	GoodOutlier RT
WS1.3	×	×	×	×	×	×		×	×	×	×	×		
WS1.4	×	×	×	×	×	×		×	×	×	×	×		
WS1.5	×	×	×	×	×	×		×	×	×	×	×		
WS1.7	×							×						
WS1.22	×							×						
WS1.26	×							×						
WS1.27	×							×						
WS1.31	×							×						
WS1.41	×							×						
WS1.52	×	×	×	×	×	×		×	×	×	×			
WS1.53	×							×						
WS1.58	×	×	×	×	×	×		×	×	×				
WS1.59	×	×	×	×	×	×		×	×	×	×	×		
WS1.79	×							×						
WS1.80	×							×						
WS1.82	×							×						
WS1.94	×							×						
Query1	×	×	×	×	×			×	×	×	×	×		

Table 9.8: The non-relational context of WS1 services described by their QoS levels.

FC_ WS1.i_ WS2.j	WS1.3	WS1.4	WS1.5	WS1.7	WS1.22	WS1.26	WS1.27	WS1.31	WS1.41	WS1.52	WS1.53	WS1.58	WS1.59	WS1.79	WS1.80	WS1.82	WS1.94	QueryWS1.i
WS2.1																		
WS2.2																		
WS2.3																		
WS2.4																		
WS2.8	×	×	×	×	×	×	×	×	×		×	×	×	×	×	×	×	
WS2.9																		
WS2.11																		
WS2.12																		
WS2.24																		
WS2.26																		
WS2.27																		
WS2.28																		
WS2.32																		
WS2.35																		
WS2.70																		
WS2.71																		
WS2.90																		
WS2.92																		
WS2.93																		
WS2.97																		
WS2.106																		
WS2.127																		
WS2.132																		
WS2.134																		
...																		
...																		
QueryWS2.j																		×

Table 9.9: The relational context representing the FC composition between services of WS1 and services of WS2.

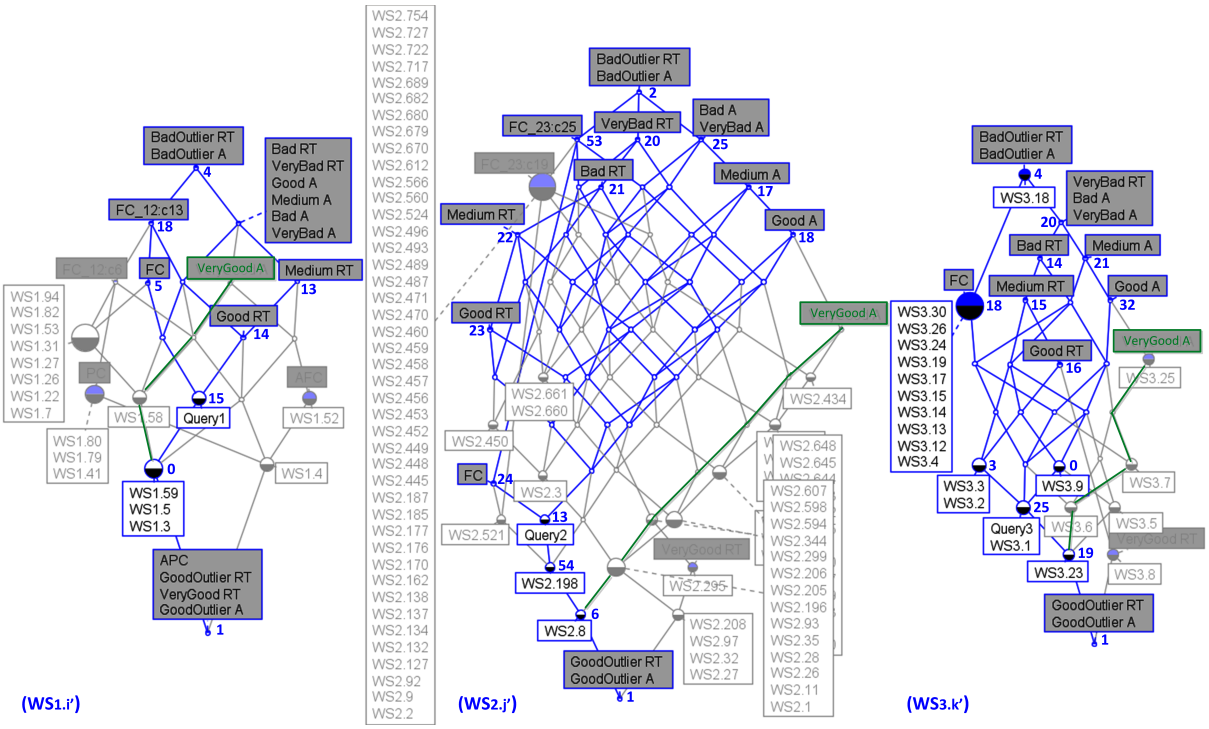


Figure 9.9: The concept lattices for the compatible sets of services with `Good A`, `Good RT` queries, and `FC` mode.

because it has a `VeryGood A` (an inherited attribute). On the other hand, the services located in a same concept have convergent QoS levels.

Following the previous rules, the lattice interpreter extracts the following services to be the best choice regarding the specified requirements:

- `{WS1.59,WS1.5,WS1.3}` for `Task1` because they all appear in the same concept `(c0)`;
- `{WS2.8}` for `Task2` because it is better than `{WS2.198}`;
- and `{WS3.23}` for `Task3` because it is better than `{WS3.1}`.

If we verify the actual services, we get the information in Table 9.10. These service lattices offer a browsing mechanism that facilitates services selection according to user requirements. In each lattice, services are classified by their QoS levels as well as their composition modes with services in a following lattice. A lattice reveals two relations between the services regarding QoS: *hasSimilarQoS* when services are located in a same concept and *hasBetterQoS* when a service is a descendant of another service(s). Services having a *hasSimilarQoS* or *hasBetterQoS* relations with a selected service, are considered to be its alternatives. User requirements (queries) can be expressed as new services to be classified in the corresponding lattices. This locates the part of the lattice that meets the user requirements and thus represent an efficient lattice navigation mechanism. In the case where no services could be found for the specified requirements, the query mechanism enables the user to identify the next best service. This service will have lesser

Table 9.10: Information about the services satisfying the queries with the selected ones (highlighted).

WS	Name	Operation	A(%)	RT(ms)
1.59	Ip2LocationWebService	IP2Location	100	257
		(in) IP:string (out) CITY:string,..		
1.5	GeoCoder	IPAddressLookup	100	328
		(in) ipAddress:string (out) City:string,..		
1.3	IP2Geo	ResolveIP	100	798
		(in) ipAddress:string (out) City:string,..		
2.198	MediCareSupplier	GetSupplierByCity (in) City:string (out) Zip:string,..	85	304
2.8	ZipcodeLookupService	CityToLatLong	100	439
		(in) city:string (out) Zip:string,..		
3.1	USWeather	GetWeatherReport (in) ZipCode:string (out) WeatherReport:string	85	384
3.23	Weather	GetCityForecastByZIP	100	237
		(in) ZIP:string (out) ForecastReturn:complex		

QoS than the requested and it represents the direct ascendant of the query concept. For example, in the third lattice of Figure 9.9, we could take `WS3.2` or `WS3.3` in concept `c3` to be the next best selection. They have a `Medium RT` and a `Good A`.

In this experiment, we had several functional and non-functional requirements needed to build a simple process of three tasks as described previously. The Service-Finder enabled us to find a total of `901` (Table 9.5) Web service addresses, among which we had to identify and retrieve the services meeting our requirements. Using our approach, we efficiently identified out of 901 endpoints a set of five services that best match our requirements (Table 9.10). The total time required to extract these services is equal to `103` sec, starting from the `WSDL Parser` (component `C`) till the end. This was calculated by `NetBeans (6.9.1)` on a machine equipped with an `Intel Core 2 Duo (1.80GHz)` processor and a memory of `(2.00 GB)`.

5 Summary

In this chapter, we presented a number of experiments to validate our framework. The experiments have proven the effectivity of our framework for:

- associating descriptive tags to their corresponding services, which enable a quick understanding of a service's functionality;
- assisting a user in selecting a needed service by its provided operations, as well as identifying its potential backups;
- assisting a user in building a desired service orchestration, by selecting sets of composable services. This assistance is based on three levels of user requirements: functionality, QoS, and composition.

In general, we showed that our framework has assisted the construction of business processes, by facilitating the selection of services and supporting them with backups to assure a continuous functionality.

Part III

Conclusion and Perspectives

Chapter 10

Conclusion

Contents

1 Conclusion

We started this dissertation with the following question:
How can a user select a suitable service to use, according to some desired requirements?

As we have seen, this question imposes several other questions and issues, which we summarize as follows:

- **Issues related to Web service discovery:**
  Web service discovery consists of describing and publishing a Web service to a public registry, to enable consumers to find it, understand its functionality, then finally select it.

  Web service description issues come from the WSDL standard, which is restricted to syntactic information. The only available semantic information can be extracted from the names of each parameter, which may not always be possible because of unclear parameter names. Inside a WSDL description, it is possible to add a textual documentation for each element to help users understand the functionality more easily, but since it is optional, it is often left empty.

  WSDL interfaces do not provide QoS information, which is especially important with the ever increasing number of functionally similar Web services. QoS helps determining the usability and utility of a Web service, to decide whether to select it or not.

  Web service publishing issues come from the deficiency of UDDI standard, which was proposed for building Web service registries. Current service registries are embodied in Web service search engines, which index services by the keywords located in their WSDL descriptions. This does not represent an efficient mechanism, because of the lack of textual

documentation inside WSDL interfaces, as we mentioned before. Using these search engines, we can search for Web services by keywords, which returns a fairly large list of Web services. Inside this returned list, a hard and time-consuming inspection for the needed operation must be carried out by the user. This inspection may consider multiple criteria, like the needed QoS, the composability of the selected service with a previously selected one, or maybe the substitutability of a broken service with the selected one. No current discovery mechanism enables the verification of such criteria, it is all left on the user's shoulder.

- **Issues related to the dynamic nature of Web services**
  Dealing with Web services, either for an independent invocation or for a composition, is threatened at any moment by failing services. Web services have a dynamic nature coming from the unpredictable Internet nature. Thus, a broken service must be replaced by another equivalent one to recover the missing functionality. This necessitates facing again all the difficulties related to Web service discovery.

Our answer to this question is embodied in the framework we are proposing. This framework can assist the construction of business processes, by facilitating the selection of real Web services and supporting them with backups to assure a continuous functionality.

Our framework works according to four layers:

- User requirements layer: in which, we enable users to express their needed functionality by an abstract WSDL interface (AWSDL) that contains a description of all the needed services (several portTypes), together with the expected QoS for each service. Users can also express a desired service orchestration by an abstract BPEL (ABPEL) file, which describes the interaction between the services described in the AWSDL interface. These two files are analyzed in the discovery layer.

- Discovery layer: in which, the AWSDL and BPEL files are analyzed. Then according to the needed services (AWSDL), a set of services can be retrieved, filtered and parsed. The services that offer the needed functionality are passed together with their information to the classification layer, to be classified into concept lattices.

- Classification layer: in which, the retrieved functionally-compatible services together with their QoS information are processed by FCA and RCA, then are classified into concept lattices. A lattice is a structure that reveals relations between services according to the considered classification criteria (keywords, functionality, QoS, and composition). It offers a navigation facility that enables an easier straightforward selection of needed services, in addition to identifying its potential backups (supporting service continuity). Resulting lattices at this layer are passed to be interpreted in the selection layer.

- Selection layer: in which, the generated lattices are interpreted, in order to identify the services that match the user requirements, together with their backups. The lattice interpreter works according to a navigation by query algorithm, which we proposed for RCA-based lattices to extract the concepts of interest. A query represents a new concept in the lattice, and the services that offer the minimum required functionality represent the

concepts that are closest to the query concept, while further situated services offer extra functionalities. In the lattice, when selecting a service, a sub-lattice that is descendant from this service can be extracted. This sub-lattice contains the possible backups that can replace this service to ensure a recovered functionality.

After generating the lattices, whenever new services are retrieved, they can be classified into the existing lattices using incremental lattice generation algorithms, without regenerating the whole lattices.

We presented a number of experiments to validate our framework. The experiments have proven the effectivity of our framework for:

- associating descriptive tags to retrieved services to enable a quick understanding of a service's functionality;
- assisting a user in selecting a needed service by its provided operations, as well as identifying its potential backups;
- assisting a user in building a desired service orchestration, by selecting sets of composable services. This assistance is based on three levels of user requirements: functionality, QoS, and composition.

In order to find our position between the works presented in Chapter 3, we reconsider the criteria table (Table 3.1), and we integrate our work in it (Thesis GOAL), as we can see in Table 1.

Work	Design-time	Run-time	Semantic WS	Discovery	Functionality	Composability	QoS		Backups
							per service	global	
Thesis GOAL	✓	×	×	✓	✓	✓	✓	×	✓

We generate the corresponding lattice, shown in Figure 10.1, which clarifies our position according to the other works. We can notice that for our fixed objectives, there is not any work that meet them all together (no concepts below Thesis GOAL). We notice also that we still miss supporting semantic Web services, being dynamic, and calculating a global QoS for a whole composition. These points draw some of our future work, as we shall see in the next section.

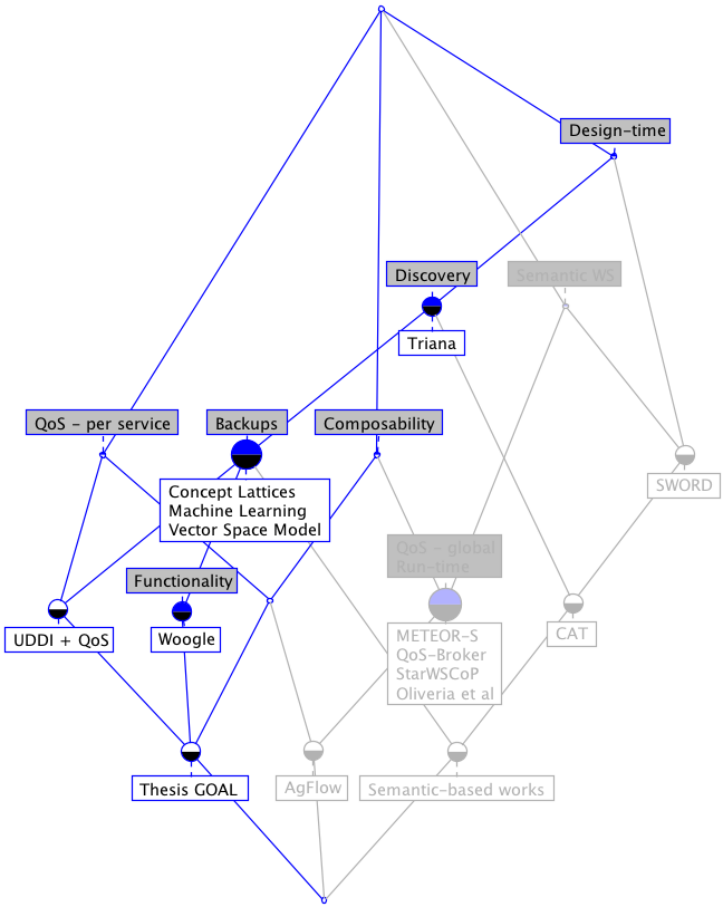


Figure 10.1: Our position in the lattice of works.

2 Perspectives

We have several perspectives for our proposed framework. We envisage first to improve the components and the used techniques for each layer in our framework. Then, we envisage evolve our framework towards defining a smart Web service registry for achieving an assisted WSOA.

2.1 Improving our Framework

We list below the envisaged improvements for each layer of our framework:

– User requirements layer: in our framework, we supposed that user's functional requirements can be specified by an AWSDL file. Inside this AWSDL, each operation parameter is specified by a set of equivalent keywords for its name, and its type. In future work, we plan to simplify the user's job, by allowing him to specify one keyword per parameter, and enhance our similarity measure using semantics (WordNet ontology for example). We may even extend the ABPEL to enable users of specifying the needed functionality inside it, without having to define an AWSDL.

Further more, we envisage dealing with semantic Web services. Like this, a user who needs certain services must semantically annotate his AWSDL file with an ontology, to enable a better understanding of user functional requirements.

– Discovery layer: the first thing we would like to do is do define our own service registry, like this, there is no need to search for services elsewhere. We plan to organize services into categories, according to their functionality, with the help of the WordNet ontology. We plan also to enhance our similarity measuring techniques to group services with similar operations together. We have started to study techniques from data mining to cluster similar signatures together, for a better retrieval of service backups.

– Classification layer: in this layer, we have to face the challenge of dynamic updating of concept lattices, for efficiently adding, removing a service, or modifying the QoS levels.

  For now, we are dealing with adding new services to the lattices without recalculating them, using the incremental lattices building algorithms. When services disappear, it might not be a good idea to immediately dismiss its indexing information, as services might be temporarily unavailable (because of a crashed web server for example). We still have to evaluate if a Web service disappearance should be handled as immediate removals (or, maybe, as lazy removals, based on their being unavailable for too long). This dynamic aspect is an interesting field for future research.

– Selection layer: in this layer, we proposed a lattice interpreter that works on identifying interesting concepts, by navigating lattices with queries. We also defined several operators for enriching the formal contexts with relational attributes. As a future work, we aim at enable lattice querying according to several scaling operators. This way, users can express a query like: I need services provided only by *provider*1, and for each service, there must exist at least one backup, which can be fully-composable with all the services having a very good availability.

2.2 Towards Defining a Structure of a Smart Web Service Registry

According to the experiments that we conducted and the results that we obtained, we find it really encouraging to go forward defining a smart Web service registry. This registry will include all the improvements that we listed above, in addition to keeping track of consumers, and monitoring the services they are invoking. This enables the registry to send service status and QoS updates for the consumers, as well as providing them with backups, upon the failure of a service in their compositions. We also plan to store all the performed queries, with the achieved compositions, for avoiding the recalculations for similar lattices, as well as implementing a learning mechanism.

This registry should also memorize the manual or automatic adaptations [88], which are necessary when a broken service is replaced by another. This information may be used to optimize future substitutions for better efficiency [89]. In [90], dynamic service generation is proposed, which would support the continuity of service compositions.

An interesting point that we can see in [9], in which, the authors propose an approach to determine Web service substitutability and composability by observing actual executions of Web services. The algorithm works on two phases. The first phase aligns invocations of different services. In the second phase, input and output parameter values are compared syntactically and matchings are deduced. Approximately, the first phase indicates composability, and the second phase computes substitutability. This can be regarded as another point of view for supporting the precision of Web services similarity measuring.

With the help of the previous smart registry, we are also planning to achieve dynamic composition and substitution, for dynamic user environments. This enables facing the dynamic changing of Web services, like when a business provides newer services or when old services are replaced by other ones. Like this, service processes will be able to transparently adapt to environment changes and requirements [91] with minimal user intervention. We plan also to enhance the specification of abstract compositions, to support the development and execution of service composition that are able to reconfigure themselves at run-time [92, 93].

Bibliography

[1] Michael P. Papazoglou. *Web Services - Principles and Technology.* Prentice Hall, 2008. (Cited on pages 1, 11 and 15.)

[2] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Mike Champion, Christopher Ferris, and David Orchard. Web services architecture. World Wide Web Consortium, Note NOTE-ws-arch-20040211, February 2004. (Cited on pages 1 and 12.)

[3] Nicolai M. Josuttis. *SOA in Practice: The Art of Distributed System Design.* O'Reilly, Beijing, 2007. (Cited on pages 1, 2, 11, 12 and 18.)

[4] Michael N. Huhns and Munindar P. Singh. Service-oriented computing: Key concepts and principles. *IEEE Internet Computing*, 9(1):75–81, 2005. (Cited on page 1.)

[5] Web Services Description Language (WSDL) 1.1, http://www.w3.org/tr/wsdl. (Cited on pages 1 and 12.)

[6] UDDI Version 3.0.2, http://www.uddi.org/pubs/uddi_v3.htm. (Cited on pages 1 and 16.)

[7] Simple object access protocol (soap) - version 1.2. (Cited on pages 1, 17 and 18.)

[8] Seekda, web service search engine. (Cited on pages 4, 17 and 109.)

[9] Michael D. Ernst, Raimondas Lencevicius, and Jeff H. Perkins. Detection of web service substitutability and composability. In *WS-MaTe 2006: International Workshop on Web Services — Modeling and Testing*, pages 123–135, Palermo, Italy, June 9, 2006. (Cited on pages 4, 29 and 136.)

[10] Eric Newcomer and Greg Lomow. *Understanding SOA with Web Services.* Addison-Wesley Professional, 2004. (Cited on page 5.)

[11] Daniel Austin, W. W. Grainger, Abbie Barbir, Christopher Ferris, and Sharad Garg. Web services architecture requirements. W3C Working Group Note 11, W3C, Feb. 2004. (Cited on page 12.)

[12] Yutu Liu, Anne H. Ngu, and Liang Z. Zeng. Qos computation and policing in dynamic web service selection. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, WWW Alt. '04, pages 66–73, New York, NY, USA, 2004. ACM. (Cited on page 15.)

[13] Liangzhao Zeng, Boualem Benatallah, Anne H. H. Ngu, Marlon Dumas, Jayant Kalagnanam, and Henry Chang. Qos-aware middleware for web services composition. *IEEE Trans. Software Eng.*, 30(5):311–327, 2004. (Cited on pages 15, 27 and 30.)

[14] Service-finder, web service search engine. (Cited on pages 17 and 109.)

[15] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services: Concepts, Architectures and Applications.* Springer, Berlin, 2004. (Cited on page 18.)

[16] Haiyan Sun, Xiaodong Wang, Bin Zhou, and Peng Zou. Research and implementation of dynamic web services composition. In Xingming Zhou, Stefan Jähnichen, Ming Xu, and Jiannong Cao, editors, *APPT*, volume 2834 of *Lecture Notes in Computer Science*, pages 457–466. Springer, 2003. (Cited on pages 18, 27 and 30.)

[17] Chris Peltz. Web services orchestration and choreography. *IEEE Computer*, 36(10):46–52, October 2003. (Cited on page 18.)

[18] Matjaz B. Juric. A hands-on introduction to bpel. (Cited on page 19.)

[19] Stephen A. White. Business Process Modeling Notation (BPMN) Version 1.0. 2004. (Cited on page 19.)

[20] Saartje Brockmans, Michael Erdmann, and Wolfgang Schoch. Research report about current state of the art of deliverable d4.1 - research report about current state of the art of matchmaking algorithms. Technical report, 2008. (Cited on page 24.)

[21] Holger Lausen and Nathalie Steinmetz. Survey of current means to discover web services. Technical report, STI Innsbruck, 08 2008. (Cited on page 24.)

[22] Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. (Cited on page 24.)

[23] Christian Platzer and Schahram Dustdar. A vector space search engine for Web services. In *Web Services, 2005. ECOWS 2005. Third IEEE European Conference on*, page 9 pp., nov. 2005. (Cited on page 24.)

[24] Yiqiao Wang and Eleni Stroulia. Semantic structure matching for assessing web-service similarity. In Maria E. Orlowska, Sanjiva Weerawarana, Mike P. Papazoglou, and Jian Yang, editors, *ICSOC*, volume 2910 of *Lecture Notes in Computer Science*, pages 194–207. Springer, 2003. (Cited on page 24.)

[25] Marco Crasso, Alejandro Zunino, and Marcelo Campo. Query by example for web services. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, pages 2376–2380, New York, NY, USA, 2008. ACM. (Cited on page 24.)

[26] Marco Crasso, Alejandro Zunino, and Marcelo Campo. Awsc: An approach to web service classification based on machine learning techniques. *Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial*, 12(37):25–36, 2008. (Cited on page 24.)

[27] Andreas Heß and Nicholas Kushmerick. Learning to attach semantic metadata to web services. In Dieter Fensel, Katia P. Sycara, and John Mylopoulos, editors, *International Semantic Web Conference*, volume 2870 of *Lecture Notes in Computer Science*, pages 258–273. Springer, 2003. (Cited on page 24.)

[28] Jiangang Ma, Yanchun Zhang, and Jing He. Efficiently finding web services using a clustering semantic approach. In Quan Z. Sheng, Ullas Nambiar, Amit P. Sheth, Biplav Srivastava, Zakaria Maamar, and Said Elnaffar, editors, *CSSSIA*, volume 292 of *ACM International Conference Proceeding Series*, page 5. ACM, 2008. (Cited on page 24.)

[29] Xin Dong, Alon Y. Halevy, Jayant Madhavan, Ema Nemes, and Jun Zhang. Similarity Search for Web Services. In *VLDB*, pages 372–383, 2004. (Cited on pages 24, 30, 93 and 96.)

[30] Lerina Aversano, Marcello Bruno, Gerardo Canfora, Massimiliano Di Penta, and Damiano Distante. Using concept lattices to support service selection. *Int. J. Web Service Res.*, 3(4):32–51, 2006. (Cited on page 24.)

[31] Marcello Bruno, Gerardo Canfora, Massimiliano Di Penta, and Rita Scognamiglio. An Approach to support Web Service Classification and Annotation. In *EEE*, pages 138–143, 2005. (Cited on page 24.)

[32] Dunlu Peng, Sheng Huang, Xiaoling Wang, and Aoying Zhou. Management and retrieval of web services based on formal concept analysis. In *CIT*, pages 269–275. IEEE Computer Society, 2005. (Cited on page 25.)

[33] Dunlu Peng, Sheng Huang, Xiaoling Wang, and Aoying Zhou. Concept-based retrieval of alternate web services. In Lizhu Zhou, Beng Chin Ooi, and Xiaofeng Meng, editors, *DASFAA*, volume 3453 of *Lecture Notes in Computer Science*, pages 359–371. Springer, 2005. (Cited on page 25.)

[34] Giuseppe Fenza and Sabrina Senatore. Friendly web services selection exploiting fuzzy formal concept analysis. *Soft Comput.*, 14(8):811–819, 2010. (Cited on page 25.)

[35] Hongliang Lai and Dexue Zhang. Concept lattices of fuzzy contexts: Formal concept analysis vs. rough set theory. *Int. J. Approx. Reasoning*, 50(5):695–707, 2009. (Cited on page 25.)

[36] Stéphanie Chollet, Vincent Lestideau, Philippe Lalanda, Diana Moreno-Garcia, and Pierre Colomb. Heterogeneous service selection based on formal concept analysis. In *SERVICES*, pages 367–374, 2010. (Cited on page 25.)

[37] Stéphanie Chollet, Vincent Lestideau, Philippe Lalanda, Pierre Colomb, and Olivier Raynaud. Building FCA-based Decision Trees for the Selection of Heterogeneous Services. In *Proceedings of International Conference on Services Computing - Application and Industry Track (SCC 2011)*, July 2011. (Cited on page 25.)

[38] Maha Driss, Naouel Moha, Yassine Jamoussi, Jean-Marc Jézéquel, and Henda Hajjami Ben Ghézala. A requirement-centric approach to web service modeling, discovery, and selection. In Paul P. Maglio, Mathias Weske, Jian Yang, and Marcelo Fantinato, editors, *ICSOC*, volume 6470 of *Lecture Notes in Computer Science*, pages 258–272, 2010. (Cited on page 25.)

[39] Shuping Ran. A model for web services discovery with qos. *SIGecom Exchanges*, 4(1):1–10, 2003. (Cited on page 25.)

[40] Mohamed Adel Serhani, Rachida Dssouli, Abdelhakim Hafid, and Houari A. Sahraoui. A qos broker based architecture for efficient web services selection. In *ICWS*, pages 113–120. IEEE Computer Society, 2005. (Cited on page 25.)

[41] Tao Yu and Kwei-Jay Lin. Qcws: an implementation of qos-capable multimedia web services. *Multimedia Tools Appl.*, 30(2):165–187, 2006. (Cited on page 25.)

[42] Tian Gramm Naumowicz, M. Tian, A. Gramm, T. Naumowicz, H. Ritter, and J. Schiller. A concept for QoS integration in web services. In *Proceedings of the Fourth international conference on Web information systems engineering workshops*, WISEW'03, pages 149–155, Washington, DC, USA, 2003. IEEE Computer Society. (Cited on page 25.)

[43] Min Tian, A. Gramm, Hartmut Ritter, and Jochen H. Schiller. Efficient selection and monitoring of qos-aware web services with the ws-qos framework. In *Web Intelligence*, pages 152–158. IEEE Computer Society, 2004. (Cited on page 25.)

[44] Wei-Tek Tsai, Raymond A. Paul, Zhibin Cao, Lian Yu, Akihiro Saimi, and Bingnan Xiao. Verification of web services using an enhanced uddi server. In *WORDS*, pages 131–138. IEEE Computer Society, 2003. (Cited on page 25.)

[45] Yamine Aït Ameur. A semantic repository for adaptive services. In *SERVICES I*, pages 211–218, 2009. (Cited on page 26.)

[46] Nabil Belaid. *Modélisation de services et de workflows sémantiques à base d'ontologies de services et d'indexations. Application à la modélisation géologique.* PhD thesis, Ecole Nationale Supérieure de Mécanique et d'Aérotechnique, 2011. (Cited on page 26.)

[47] Xia Wang, Tomas Vitvar, Mick Kerrigan, and Ioan Toma. A qos-aware selection model for semantic web services. In *4th International Conference on Service Oriented Computing (ICSOC)*, pages 390–401. Springer, 2006. (Cited on page 26.)

[48] Dumitru Roman, Uwe Keller, Holger Lausen, Jos de Bruijn, Rubén Lara, Michael Stollberg, Polleres, Cristina Feier, Cristoph Bussler, and Dieter Fensel. Web service modeling ontology. *Applied Ontology*, 1(1):77–106, 2005. (Cited on page 26.)

[49] Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, and Katia P. Sycara. Semantic matching of web services capabilities. In Ian Horrocks and James A. Hendler, editors, *International Semantic Web Conference*, volume 2342 of *Lecture Notes in Computer Science*, pages 333–347. Springer, 2002. (Cited on page 26.)

[50] Boualem Benatallah, Mohand-Said Hacid, Alain Léger, Christophe Rey, and Farouk Toumani. On automating web services discovery. *VLDB J.*, 14(1):84–96, 2005. (Cited on page 26.)

[51] Shalil Majithia, Matthew S. Shields, Ian J. Taylor, and Ian Wang. Triana: A graphical web service composition and execution toolkit. In *ICWS*, pages 514–. IEEE Computer Society, 2004. (Cited on pages 26 and 30.)

[52] Jihie Kim and Yolanda Gil. Towards interactive composition of semantic web services. *National Conference on Artificial Intelligence*, 2004. (Cited on pages 26, 27 and 30.)

[53] Shankar R. Ponnekanti and Armando Fox. Sword: A developer toolkit for web service composition. In *Proceedings of the 11th International WWW Conference (WWW2002)*, Honolulu, HI, USA, 2002. (Cited on pages 26, 27 and 30.)

[54] The web services toolkit (wstk), http://www.alphaworks.ibm.com/tech/webservicestoolkit. (Cited on pages 26 and 28.)

[55] Alberto Martínez, Marta Patiño-Martínez, Ricardo Jiménez-Peris, and Francisco Pérez-Sorrosal. Zenflow: A visual web service composition tool for bpel4ws. *Visual Languages and Human-Centric Computing, IEEE Symposium on*, 0:181–188, 2005. (Cited on page 26.)

[56] Idir Aït-Sadoune and Yamine Aït Ameur. Stepwise design of bpel web services compositions: An event_b refinement based approach. In Roger Y. Lee, Olga Ormandjieva, Alain Abran, and Constantinos Constantinides, editors, *SERA (selected papers)*, volume 296 of *Studies in Computational Intelligence*, pages 51–68. Springer, 2010. (Cited on page 26.)

[57] Frederico G. Alvares de Oliveira Jr. and José M. Parente de Oliveira. Qos-based approach for dynamic web service composition. *J. UCS*, 17(5):712–741, 2011. (Cited on page 27.)

[58] Thomas Weise, Steffen Bleul, Diana Elena Comes, and Kurt Geihs. Different Approaches to Semantic Web Service Composition. In Abdelhamid Mellouk, Jun Bi, Guadalupe Ortiz, Kak Wah (Dickson) Chiu, and Manuela Popescu, editors, *Proceedings of The Third International Conference on Internet and Web Applications and Services (ICIW'08)*, pages 90–96. IEEE Computer Society Press: Los Alamitos, CA, USA, 2008. (Cited on page 27.)

[59] Tao Yu, Yue Zhang, and Kwei-Jay Lin. Efficient algorithms for web services selection with end-to-end qos constraints. *TWEB*, 1(1), 2007. (Cited on pages 28 and 30.)

[60] Martin MOSER, Dusan P. JOKANOVIC, and Norio SHIRATORI. An algorithm for the multidimensional multiple-choice knapsack problem. *IEICE Trans. Fundamentals*, Vol.E80-A(No.3):582–589, March 1997. (Cited on page 28.)

[61] Rohit Aggarwal, Kunal Verma, John A. Miller, and William Milnor. Constraint driven web service composition in meteor-s. In *IEEE SCC*, pages 23–30. IEEE Computer Society, 2004. (Cited on pages 28 and 30.)

[62] Conexp, the concept explorer tool. (Cited on page 29.)

[63] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer, Berlin/Heidelberg, 1999. (Cited on page 33.)

[64] GaLicia, "galois lattice interactive constructor", 2002. (Cited on pages 36, 105 and 117.)

[65] Erca framework for formal and relational concept analysis. (Cited on page 38.)

[66] Marianne Huchard, Mohamed Rouane Hacene, Cyril Roume, and Petko Valtchev. Relational concept discovery in structured datasets. *Ann. Math. Artif. Intell.*, 49(1-4):39–76, 2007. (Cited on page 37.)

[67] Sergei O. Kuznetsov and Sergei A. Obiedkov. Comparing performance of algorithms for generating concept lattices. *J. Exp. Theor. Artif. Intell.*, 14(2-3):189–216, 2002. (Cited on page 58.)

[68] Dean van der Merwe, Sergei A. Obiedkov, and Derrick G. Kourie. Addintent: A new incremental algorithm for constructing concept lattices. In Peter W. Eklund, editor, *ICFCA*, volume 2961 of *Lecture Notes in Computer Science*, pages 372–385. Springer, 2004. (Cited on page 58.)

[69] Helmut Schmid. Probabilistic part-of-speech tagging using decision trees. In *Proceedings of the International Conference on New Methods in Language Processing*, Manchester, UK, 1994. (Cited on page 81.)

[70] Eibe Frank, Gordon W. Paynter, Ian H. Witten, Carl Gutwin, and Craig G. Nevill-Manning. Domain-specific keyphrase extraction. In *IJCAI '99: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 668–673, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. (Cited on page 82.)

[71] Gerard Salton. Developments in automatic text retrieval. *Science*, 253:974–979, 1991. (Cited on page 82.)

[72] Usama M. Fayyad and Keki B. Irani. Multi-interval discretization of continuous-valued attributes for classification learning. In *IJCAI*, pages 1022–1029, 1993. (Cited on page 82.)

[73] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, October 1999. (Cited on page 83.)

[74] Pedro Domingos and Michael J. Pazzani. On the optimality of the simple bayesian classifier under zero-one loss. *Machine Learning*, 29(2-3):103–130, 1997. (Cited on page 83.)

[75] Christiane Fellbaum, editor. *WordNet: An Electronic Lexical Database*. MIT Press, Cambridge, MA, 1998. (Cited on pages 83 and 111.)

[76] Robert Godin, Guy Mineau, Rokia Missaoui, and Hafedh Mili. Méthodes de classification conceptuelle basées sur les treillis de Galois et applications. *Revue d'Intelligence Artificielle*, 9(2):105–137, 1995. (Cited on page 89.)

[77] Claudio Carpineto and Giovanni Romano. A lattice conceptual clustering system and its application to browsing retrieval. *Machine Learning*, 24(2):95–122, 1996. (Cited on page 89.)

[78] Eleni Stroulia and Yiqiao Wang. Structural and semantic matching for assessing web-service similarity. *Int. J. Cooperative Inf. Syst.*, 14(4):407–438, 2005. (Cited on pages 93 and 96.)

[79] Natalia Kokash. A Comparison of Web Service Interface Similarity Measures. Technical report, University of Trento, 2006. (Cited on pages 93 and 96.)

[80] William W. Cohen, Pradeep Ravikumar, and Stephen E. Fienberg. A comparison of string distance metrics for name-matching tasks. In Craig Knoblock and Subbarao Kambhampati, editors, *Proceedings of IJCAI-03 Workshop on Information Integration*, pages 73–78, Acapulco, Mexico, August 2003. (Cited on pages 93, 104 and 117.)

[81] Oktie Hassanzadeh. Benchmarking declarative approximate selection predicates. *CoRR*, abs/0907.2471, 2009. informal publication. (Cited on pages 93 and 104.)

[82] Wayne A Larsen and JW Tukey. Variations of box plots. volume 32, pages 12–16, 1978. (Cited on pages 105 and 147.)

[83] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. The skyline operator. In *Proceedings of the 17th International Conference on Data Engineering*, pages 421–430, Washington, DC, USA, 2001. IEEE Computer Society. (Cited on page 107.)

[84] Steve Jones and Gordon W. Paynter. Human evaluation of kea, an automatic keyphrasing system. In *JCDL*, pages 148–156. ACM, 2001. (Cited on page 110.)

[85] Steve Jones and Gordon W. Paynter. Automatic extraction of document keyphrases for use in digital libraries: Evaluation and applications. *JASIST*, 53(8):653–677, 2002. (Cited on page 110.)

[86] Web Services Business Process Execution Language Version 2.0, http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html. (Cited on page 112.)

[87] NetBeans IDE, http://www.netbeans.org/. (Cited on page 112.)

[88] MOREAU Aurélien. *Mise en oeuvre automatique de processus métier dans le domaine des architectures orientées services*. PhD thesis, LIP6 - Université Pierre & Marie Curie, April 2009. (Cited on page 135.)

[89] Gabriela Arévalo, Zeina Azmeh, Marianne Huchard, Chouki Tibermacine, Christelle Urtado, and Sylvain Vauttier. Component and service farms. In Eric Cariou, Laurence Duchien, and Yves Ledru, editors, *Actes des deuxièmes journées nationales du GDR Génie de la Programmation et du Logiciel — Défis pour le Génie de la Programmation et du Logiciel*, pages 281–284, Pau, France, Mars 2010. (Cited on page 135.)

[90] Clement Jonquet and Stefano A. Cerri. The strobe model: Dynamic service generation on the grid. *Applied Artificial Intelligence*, 19(9-10):967–1013, 2005. (Cited on page 135.)

[91] Gabriel Hermosillo, Lionel Seinturier, and Laurence Duchien. Creating context-adaptive business processes. In Paul P. Maglio, Mathias Weske, Jian Yang, and Marcelo Fantinato, editors, *ICSOC*, volume 6470 of *Lecture Notes in Computer Science*, pages 228–242, 2010. (Cited on page 136.)

[92] Stéphanie Chollet and Philippe Lalanda. An extensible abstract service orchestration framework. In *ICWS*, pages 831–838. IEEE, 2009. (Cited on page 136.)

[93] Samir Dami, Jacky Estublier, and Mahfoud Amiour. Apel: A graphical yet executable formalism for process modeling. *Autom. Softw. Eng.*, 5(1):61–96, 1998. (Cited on page 136.)

[94] Pawan Lingras, Rui Yan, and Chad West. Comparison of conventional and rough k-means clustering. In Guoyin Wang, Qing Liu, Yiyu Yao, and Andrzej Skowron, editors, *RSFDGrC*, volume 2639 of *Lecture Notes in Computer Science*, pages 130–137. Springer, 2003. (Cited on page 150.)

[95] Carlos D. Martínez-Hinarejos, Alfons Juan, and Francisco Casacuberta. Generalized k-medians clustering for strings. In Francisco J. Perales López, Aurélio C. Campilho, Nicolas Pérez de la Blanca, and Alberto Sanfeliu, editors, *IbPRIA*, volume 2652 of *Lecture Notes in Computer Science*, pages 502–509. Springer, 2003. (Cited on page 150.)

Part IV

Appendices

Appendix A

BoxPlot++

1 Definition

A boxplot [82] is a statistical tool that represents graphically the distribution of a set of numerical data. It splits a data set into quartiles by calculating five numbers:

- the median (Q2): the value separating the higher half of a sample from the lower half;
- the upper quartile (Q3): the median of the higher half of the data set;
- the lower quartile (Q1): the median of the lower half of the data set;
- the minimum value;
- and the maximum value.

The length of the box is represented by the inter quartile (IQ), which is the difference between the upper and the lower quartiles. The inter quartile tells how spread out the "middle" values are; it can also be used to tell when some of the other values are "too far" from the central value. These "too far" points are called "outliers", because they "lie outside" the range in which we expect them. An outlier is any value that lies more than one and a half times the length of the box from either end of the box. That is, if a data point is below Q1 $1.5 \times$ IQ or above Q3 + $1.5 \times$ IQ, it is viewed as being too far from the central values to be reasonable. In Figure A.1[1], we see a graphical representation of a boxplot with its five numbers.

2 Utilization

Creating a boxplot starts by ordering the data. Then, finding the 3 medians (Q1, Q2, and Q3). When finding a median number, if the data set has an even number of values, then the median is the average of the two middle values. If we have a data set of an odd number of values, then the median is the middle value.

Having the following set of numbers: {1, 1, 1, 2, 3, 4, 5, 10, 15, 54, 60, 70, 75, 88, 90, 93}, we find the following results:
median (Q2) = 12.5;
lower quartile (Q1) = 2.5;
upper quartile (Q3) = 72.5;
inter quartile (IQ) = 70;
min bound = -102.5;
max bound = 177.5.

[1] Image taken from http://www.cms.murdoch.edu.au/areas/maths/statsnotes/samplestats/images/

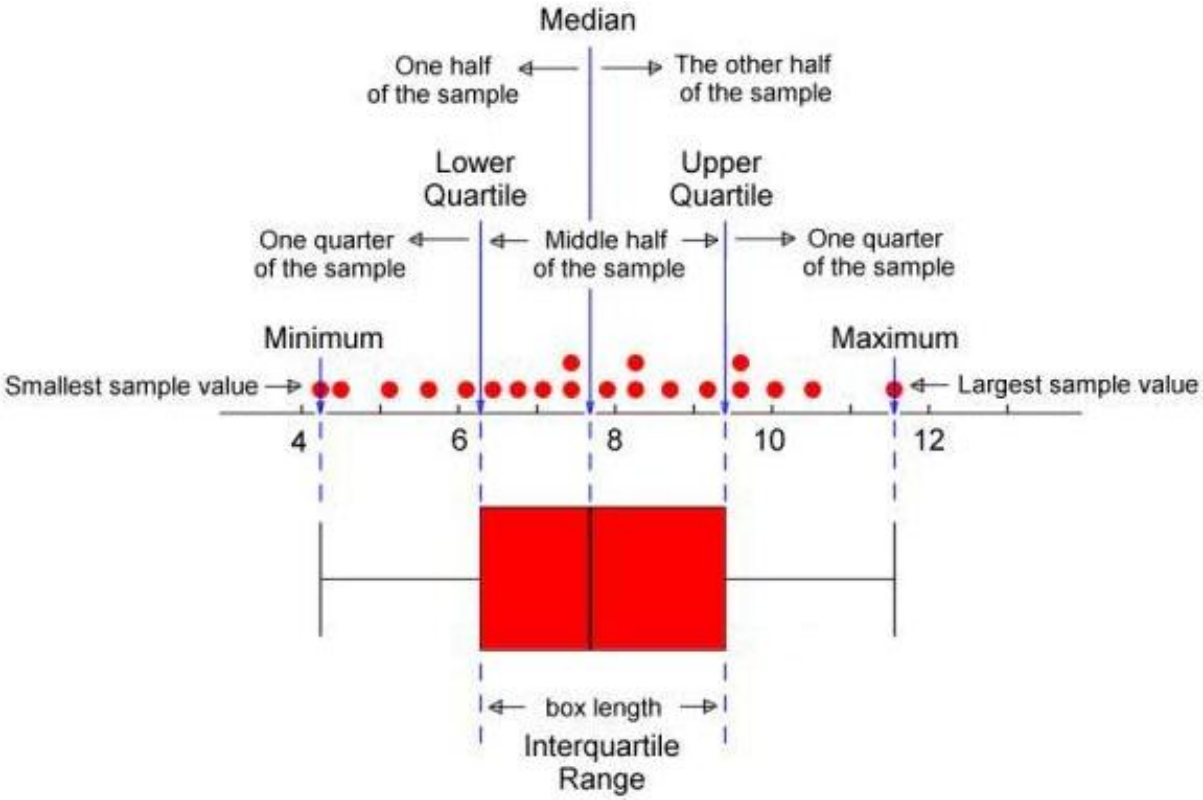


Figure A.1: A boxplot graphical representation.

The resulting boxplot is shown in Figure A.2[2]. Like this, the values that are higher than Q3 are considered as high values, and they are:
{75, 88, 90, 93};
the values that are lower than Q1 are considered as low values, and they are:
{1, 1, 1, 2};
the values in the IQ range are considered to be middle values, and they are:
{3, 4, 5, 10, 15, 54, 60, 70};
there are no high outliers, because the max bound = 177.5 and all the values are lower than it. In the same way, we find that there are no low outliers because the min bound = -102.5 and all the values are higher than it.

By regarding the middle values set, we notice high differences between its values, like 3, 4, 5, 10, 15 and 54, 60, 70. This does not give us a precise representation of data distribution. Therefore, in the next section, we present our proposition named BoxPlot++, which is based on calculating distances between values and medians.

[2]We used http://www.shodor.org/interactivate/activities/BoxPlot/ to generate the graphical representation of the boxplots.

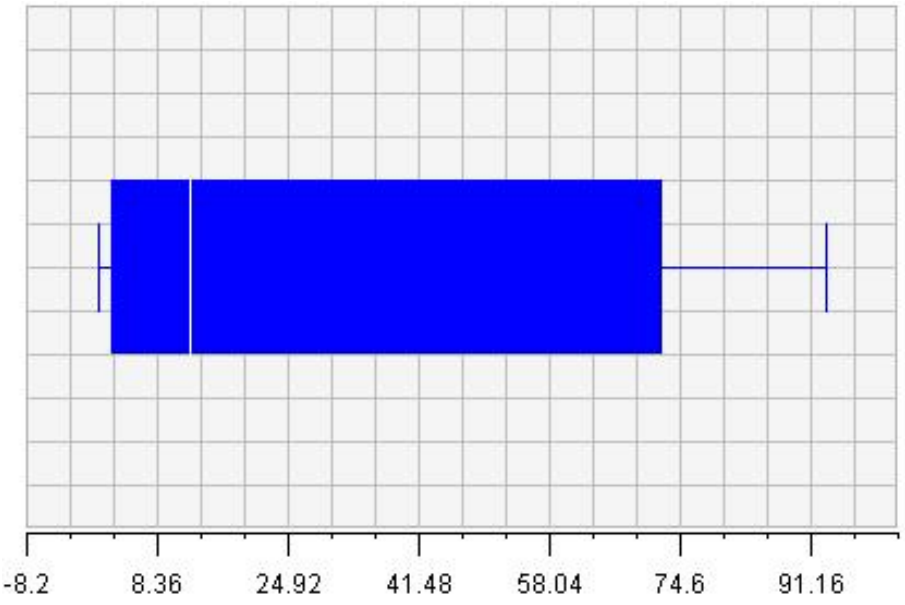


Figure A.2: The boxplot corresponding to the input data set.

3 BoxPlot++

We propose to extend the original boxplot, in order to have a more precise distribution of values, especially the middle ones.

Our idea is to measure the distance between the data points and the clusters' centers. We consider a cluster's center to be its median. Thus, we measure the distances of each point from its two adjacent medians.

We begin the calculation by omitting the repeated values. Then, we apply the original boxplot technique. Afterwards, we take each resulting set and calculate the distance of each point of it from its adjacent medians.

Thus, taking the same previous example set: {1, 1, 1, 2, 3, 4, 5, 10, 15, 54, 60, 70, 75, 88, 90, 93}, we get the following results:
median (Q2) = 34.5;
lower quartile (Q1) = 4;
upper quartile (Q3) = 75;
inter quartile (IQ) = 71;
min bound = -102.5;
max bound = 181.5.
The resulting boxplot is shown in Figure A.3.

In our approach, we have added two more levels:
- lower values, which are more close to the min value than to the lower quartile;
- higher values, which are more close to the max values than to the upper quartile.
Like this, the higher values are: {88, 90, 93};
the high value: {60, 70, 75}
the low values are: {2, 3, 4, 5, 10, 15};
the lower values are: {1};

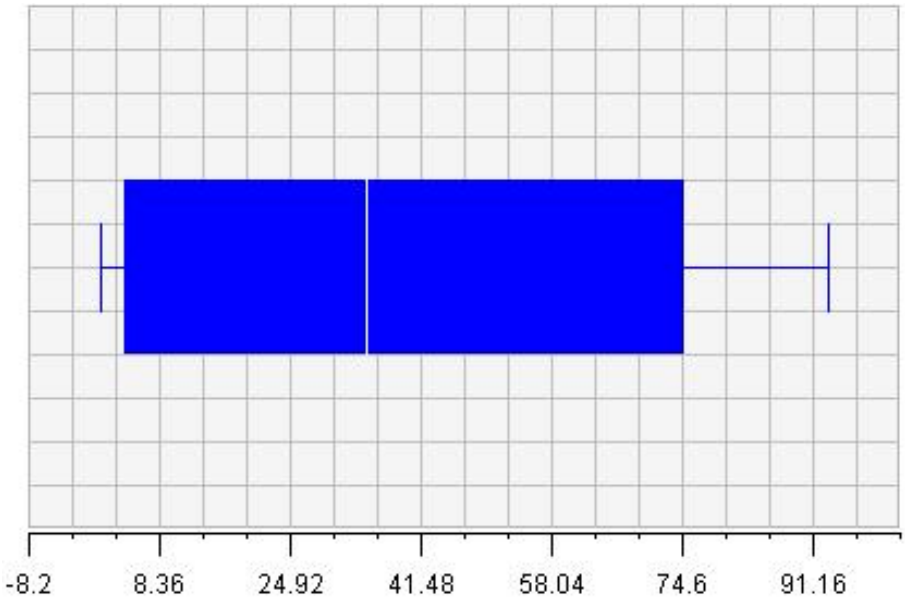


Figure A.3: The boxplot corresponding to the input data set.

the values in the IQ range are: {54};
and there are neither low outliers nor high outliers.

The tool is put for online test at the address:
http://www.lirmm.fr/~azmeh/tools/BoxPlot/BoxPlot.html

4 Related work

In the literature we find similar techniques to cluster sets of points according to center points. We mention two of them: k-means [94] and k-medians [95].

4.1 *k*-means clustering

K-means aims at partitioning a set of data points into k clusters in which each point belongs to the cluster with the nearest mean.

4.2 The *k*-medians clustering

k-medians clustering is a variation of k-means clustering where instead of calculating the mean for each cluster to determine its centroid, one instead calculates the median.

The definition of k-median is as follows: Given a data set N of nodes, a distance function d : N2 $\longrightarrow$ R, and an integer k; find a k element subset of N as medians such that sum of distances from each node to its nearest median is minimal. The nodes that are closer to a median form a cluster. For any node, the node is said to be assigned to its nearest median.

5 Conclusion

We proposed the BoxPlot++ as an extension of Tukey's boxplot. We improved the resulting data values distribution by removing the repeated values and by calculating distances between the points and the nearest median. The values in the resulting cluster show more precision than the original boxplot approach.