FACHHOCHSCHULE **HOCHSCHULE FÜR**
STUTTGART **TECHNIK**

STUTTGART UNIVERSITY OF APPLIED SCIENCES

Department of Geomatics, Computer Science and Mathematics

Conceptualization and Realization of an API for Semantic Discovery of Web Services

By

Alexandre Victor Rodrigues Lins

Master Thesis

Master of Science (M.Sc.) in Software Technology

Submission date: January 30th, 2004

Thesis Committee:

Prof. Dr. Gerhard Wanner
Dipl. Inf. Thomas Schlegel

Hereby I declare that I have written this work independently and used none other than the indicated aids. References to the work of others are clearly documented.

Full name: Alexandre Victor Rodrigues Lins

Signature:

Stuttgart, January 30th, 2004.

I would like to thank Doris Janssen and Thomas Schlegel for the opportunity to develop this work at the Fraunhofer Institute, and for your orientation and support.

Table of Contents

List of Figures

List of Diagrams

List of Tables

List of Code Fragments

1 Introduction

This thesis presents an approach for the development of applications for the semantic discovery of web services. The work examines the standards, technologies and tools currently available in the field, and proposes the definition of an application programming interface that supports the development of applications based on the OWL-S semantic descriptions in the Java programming language.

1.1 Motivation

The web services technologies have gained broad interest in the software industry in the last few years. These technologies are expected to provide applications with a level of interoperability not available so far in the software industry. This interoperability is guaranteed by a set of basic standards that support the description, search and remote call of these programs, independent of language or platform they have been developed.

On the other side, semantic web technologies have also turned into a growing research field in the past years. These technologies are expected to provide software applications with a common language that is able to express data and rules about this same data in an unambiguous form, allowing for the development of intelligent applications that can actually communicate and interact with each other.

In fact, the two technologies described above complement each other, as the web services paradigm focuses on standards and tools to support interoperability while the semantic web technologies concentrates on languages, standards and tools that allow for the development of intelligent applications based on a common language. Combining both technologies, the creation interoperable intelligent software becomes possible.

But despite the growing interest and years of research, the semantic web technologies find themselves still in an early stage. In fact, the work on languages and standards is quite advanced, but the shortage of tools to support the development of applications based on these technologies has slowed down their adoption. In regard to the combination of semantic web and web services technologies, this fact is even more real.

In this last case, even the languages and standards are still in an early stage. Moreover, great part of the work has focused on developing building block technologies and tools that address very specific areas such as the development of semantic registries and frameworks for the composition of services. Even though this work is important, there is still a lack of more general tools that can be used to develop semantic web applications.

This work proposes a different approach by defining a basic API for the development of applications based on web services and semantic web technologies. The API defines a set of design constructs upon which applications in the Java programming language can be developed.

1.2 Scope

As discussed above, a major part of the research work in the combined area of semantic web and web services has focused on specific areas and problems. A main problem today though, is the lack of a framework upon which developers can build and execute their applications. The main goal of the work presented here is to define a general tool that can assist on the development of these applications.

More specifically, this work defines an API that allows for the creation of semantic descriptions based on OWL-S and the discovery of services on remote registries. The API should provide the necessary functionality to perform operations on registries based on semantic web technologies, abstracting the developer from implementation details, so that he can focus on the business rules. In addition, it will serve as a basis for a broader OWL-S toolkit to be developed in the future, that will provide support for service invocation and composition.

Furthermore, a reference implementation of the API and a proof of concept prototype are developed as part of this work. In the lack of a registry based on semantic web technologies, the implementation is based on a UDDI registry, and so it is unable of performing searches that rely on semantic matching. To allow for semantic matching, some of the functionality is hard coded in the API implementation for demonstration and evaluation purposes.

1.3 Organization

This thesis is organized in five chapters. This chapter gives a brief introduction to the work. Chapter 2 gives an overview of the technologies, standards, languages, and tools that are currently under development in the combined area of web services and semantic web technologies. To be more specific, it provides an overview of the standards and languages in both areas and then focuses on the OWL-S ontology and semantic registry tools available today.

The third chapter provides the definition of the service discovery API. It discusses the architectural issues and the decisions that were taken on the design of the tool. Chapter 4 presents a reference implementation of the API and a proof of concept prototype that uses the API to perform operations on a registry. A critical evaluation is carried out based on this prototype and advantages and shortcomings are pointed out. The final chapter presents the conclusions about the work.

2 Overview

This chapter introduces the technologies, standards and tools related to web services and the semantic web, and discuss how these technologies can be applied to improve the integration of applications. A special focus is given on the DAML-S and OWL-S ontologies and the applications available today that are based in these standards. The chapter starts with a problem definition where an example application scenario is described on which these technologies could be applied to solve a problem.

2.1 Problem Definition

To expose how web services and semantic web technologies could be applied to improve the integration of software applications that are being developed today, one needs to place the use of these technologies into a context. This section presents an application scenario and describes briefly the limitations that exist today, and how the technologies mentioned above could help solve these problems.

This work has chosen the transportation services industry as the example application scenario. Transportation companies have software applications that automate important processes inside the company, reducing the cost of performing these operations using human labor. Two examples are programs that query information about services offered and applications responsible for placing order of shipments.

Programs that query information about services offered by the company can reduce costs by giving the power to the clients to operate queries directly on the companies database without any intervention of human resources from the company. The client places a query asking, for example, what services are available to transport a shipment from a place to another in an specific date, and get as response the list of services and prices from that company. Figure 2-1 shows this arrangement.

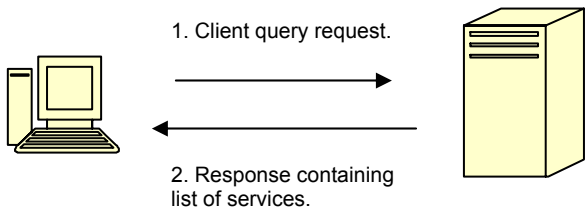

Figure 2-1: Client-Server integration.

The same applies for the shipment application, where the client can place orders of shipment and perhaps even pay for them electronically without human intervention. In fact, these applications are available today on the web sites from logistics companies, where clients can register themselves and perform operations electronically. These are rather simple examples of applications in the logistics field.

Even though simple, these applications still can be improved and help reduce costs for both clients and service providers. A very simple way of improving these applications is by connecting client programs to the programs lying on the transportation company server. This fact does not necessarily eliminates the need for human intervention on the client side, but it can be used to connect processes in the client side to the operation on the server.

2.1.1 Integration Problems

The field of connecting different applications together is known as enterprise application integration. Integrating applications can help reduce costs by connecting the processes that are isolated on each side and that usually require human intervention. But integration of applications post some problems, though, as the programs being integrated usually present different characteristics. In fact, in most cases these programs were not developed to work together.

Some of the programs have been developed using different programming languages and to run on different platforms. Other programs use different names and data structures to represent the same concept. Some have been developed to work under a particular communication or security protocol. The list of problems related to enterprise application integration is in fact large and does not end here.

Indeed, application integrators have to deal with a lot of these problems when faced with the task of putting programs to work together. An even bigger difficulty related to integration comes from the fact that usually one needs to adapt its program to each application it wants to work with. And this usually leads to the development of programs that deals with different technologies, protocols, domain models, and that are hard to extend and maintain. Figure 2-2 shows how a client application needs to be adapted to work with applications developed with different technologies.

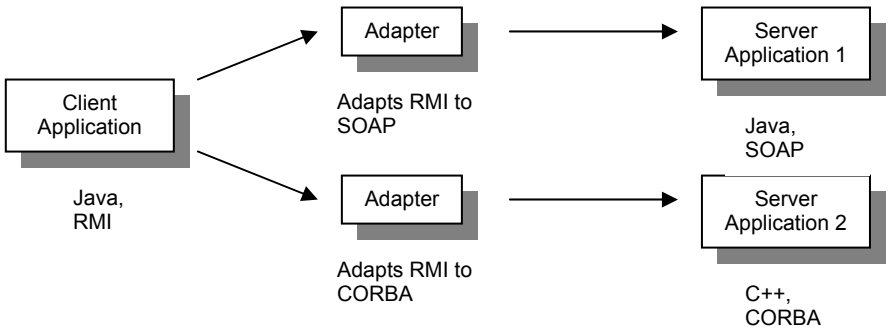

Figure 2-2: Adapting clients for integration.

The problems regarding the integration of applications are old, and the software industry has been working on it for quite a time. In fact, some tools exist today that makes the integration effort easier to handle. Middleware applications and messaging frameworks such as the Java Messaging Services [1], JMS, provide a common platform upon which applications can be integrated. In this case, the programs need to be adapted

on both sides to work with a single technology. Figure 2-3 shows how applications using JMS integrate.

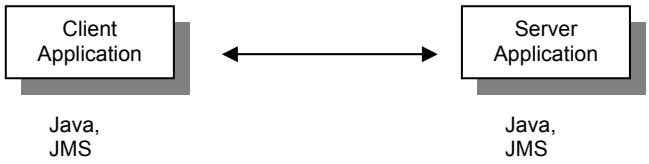

Figure 2-3: Using JMS for integration.

These products offer a good platform and functionality for integration, but they also present some problems. In effect, they usually rely on a particular programming language or operating system platform that the software being adapted must follow. This makes them a good choice for integration inside the enterprise where a single platform for development can be established. But if one needs to integrate outside the enterprise, these solutions might not fit.

Taking the example of the transportation services industry, a client or a transportation company cannot force their business partners to adopt a particular technology, as this would probably force them to adopt a particular platform or programming language. The partners could even get to an agreement, but if the client, for example, decided to work with other service providers, it would still need to deal with integration problems.

So what is needed at this point is a technology that is able to establish a common standard upon which applications can be integrated. This technology would need to cope with the problems of differences among programming languages, operating platforms, communication protocols, and the representation and semantic of the data being exchanged. Once such a technology becomes available, at least part of the problems related to integration would be solved.

2.1.2 Interoperability

The web services are a set of technologies currently under development that can be used to support the integration of applications. It particularly tackles the problem of establishing a common platform for performing remote calls that is independent of programming languages or operating platforms. This is done by defining a set of standards that are discussed more ahead in this work.

In the transportation industry example, the transportation companies would adapt their applications to work as web services. The query and order of shipments applications, for example, would be transformed into web services and exposed in a server in the company for clients to access. The clients, on the other side, would also need to adapt their own programs to place remote calls to these services. Figure 2-4 shows an example where applications are adapted to work with the web services SOAP protocol.

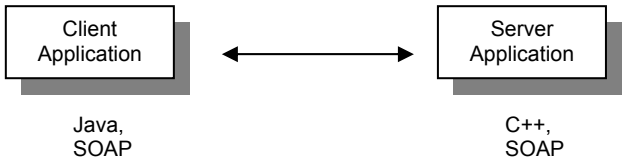

Figure 2-4: Using web services for integration.

Problems regarding data representation and semantic must be solved in advance between the partners, though. For example, if the web service represents a location using a code, while the client application uses a location name, then even though this data is defined as being of type String on both programs, still the integration here would not work. Here the client must be adapted to convert location names into codes, and this is only possible if the partners agree previously about the representation.

To cope in part with these problems related to representation of data, standards have been defined that can be used to describe web services characteristics and technical specifications. This data is stored on registries that can be accessed by the clients to retrieve information about the services and determine what needs to be adapted and how they can connect to the service. These standards are discussed ahead in more detail.

Here one could imagine a scenario where different transportation companies could use the registries to expose their services. On the other side, clients would query these registries to gather information about the services, choose the services they want to call, and perhaps adapt their programs to cope with the representation problems described above. Figure 2-5 shows such an arrangement.

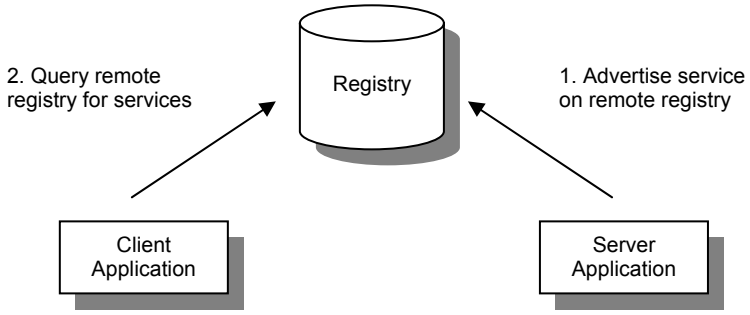

Figure 2-5: Exposing services on remote registries.

In the transportation example, the companies could offer a service that lists the prices for transporting for a determined location. The client would search for these services in the registry and choose the one it wants. But the location parameter for some companies is defined by a type of code, while other companies use different types. The client application uses a location name, though, and so it needs to be adapted to work without a problem.

2.1.3 Data Representation

There is not an easy solution to the problem of the need to adapt client applications. But the fact is that this problem somehow restricts the integration of applications, specially in the web services model. Indeed, if there were no need to adapt applications, their integration could be made automatically. The client programs themselves would be able to search for service on the registries, choose the ones they need, and connect to them to perform operations, without any human intervention.

The impact could be even bigger as service providers could be able to delete, modify and insert new services into the registry. Clients that used a determined service that was deleted the day before would automatically choose another service from the list. New services also would be automatically identified, and since there is no need to adapt the client program to work with them, they could be used without a problem by these applications.

Moreover, clients would be able to compose single services into more complete services. As an example, if a company transports only from the origin to a certain region, and another company transports from this region to the destination, then the clients could place orders to both companies and compose their services. In case a new service that transports directly to the destination was created, it would be automatically identified, and the client would be able to choose between two services now.

As it was mentioned above, no easy solution to this problem is currently available. A set of technologies is under development that addresses this problem, though. They have been identified under the name of semantic web technologies, and they will be discussed in more detail ahead in the coming sections. But these technologies do not focus on adapting programs. Instead, the focus is on providing a common language that programs can use to represent similar concepts.

In the case of transportation services, for example, a common representation for identifying locations could be determined, and this representation would be used by both clients and services. In fact, clients and services could even use a different representation inside their code, as long as they could be programmed to adapt their own representation of a location to the common representation of this data. There is still the need to adapt the programs, but now there are common concepts that are shared by all applications, instead of different representation to each one of them.

One must say this is a rather easy example and it somehow simplifies the effort that is needed to make this sort of integration work. In effect, behind the scenes a lot of technologies and tools are needed, and are actually currently under development, that support this vision. But once these tools and technologies become available, application integration can be taken to a next level. These technologies and tools are present and discussed in more detail in the following sections.

2.2 Web Services

Web services [2] are applications based on a set standards that can be accessed by other client applications over the Internet. The services advertise their characteristics on

registries that client applications can use when looking for a determined web service. The client then uses the data provided by the registry to select and call the service that is going to perform the operation and return the results.

2.2.1 Web Services Standards

A set of three basic standards establish the rules for web services operation. The Simple Object Access Protocol [3], SOAP, is perhaps the most important of them. This standard determines a common way to describe remote calls from the client application to the web service. Software vendors provide the APIs for SOAP messaging and the components known as SOAP engines that manage the communication between client and service.

On the other hand, the Universal Discovery, Description and Integration standard [4], UDDI, defines a common manner for describing service capabilities. Service developers can use registries based on the UDDI standard to store and advertise their services capabilities. Vendors provide APIs that application developers can use to create service descriptions and perform update and query information on the registries.

Finally, the Web Services Definition Language [5], WSDL, is responsible for describing details such as addressing and type information that are needed by the client to perform a remote call. This information is usually integrated somehow on UDDI service descriptions and provide the additional information necessary to perform remote calls [6]. Here also APIs exist that enable the creation and manipulation of WSDL documents.

Together, these standards define the rules that govern the web services paradigm and allows for the interoperability between software applications. In fact, this interoperability is the greatest advantage of web services, as it allows applications that might have been developed in different languages and run in different platforms to work together. These qualities make web services a great choice for enterprise application integration.

Using the transportation services industry as an example, transportation companies would be able to develop their applications in any language and platform they choose. They would advertise their services on UDDI registries providing additionally the necessary WSDL information for making the connection. A SOAP engine lying on the server would be responsible for receiving the remote calls and passing them on to the service. Figure 2-6 shows this arrangement.

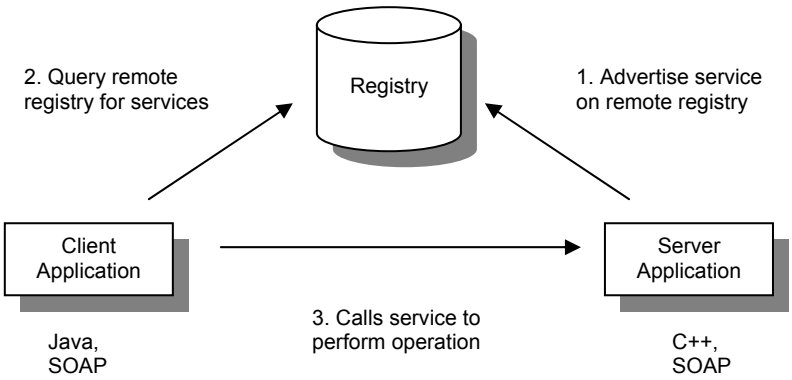

Figure 2-6: Web services operation example.

On the other side, the clients of these companies could use UDDI registries to search for services, and get WSDL data containing connection information. They would then use this data to develop their own applications using another programming language or platform. This is a fair simple example of how enterprise applications could be easily connected by using web services.

2.2.2 Critics on Web Services

The use of web services presents some shortcomings, though. In fact, the SOAP standard presents some problems regarding interoperability, as different implementations of SOAP engines seem to have problems working together. That means that if clients and services use different engines, the call to the service might not work. Initiatives such as the Web Services Interoperability [7], WS-I, are under development to cope with these problems.

Another weak point regarding the technology comes from the fact that the standards described above do not provide support to issues such as transaction support, workflow composition and security issues. Indeed, these turn out to be very important issues in the point of view of enterprise application integration. Complementary standards are currently under development to fill up this gap such as WSCI [8], BPEL4WS[9], WS-Transaction [10], and WS-Security [11].

A final problem that limits the application of web services rises from the fact that applications need to know in advance the meaning of the data they are exchanging and processing. If clients and services had a common language they could understand, then they could be programmed to process the meaning of this data. This means that clients would be able to find, select and call the services they need without human intervention. A set of technologies is under development that deals with these problems. They are discussed in the next section.

2.3 Semantic Web Technologies

The semantic web technologies address the problem of providing a common language and supporting tools that can be used to express and process the semantic of the data

being exchanged between two applications [12]. Such language must be unambiguous and powerful enough to express both data and rules about this same data, so that applications can perform reasoning operations upon it.

Once such languages and tools become available, the implementation of a wide range of new applications is going to become possible. In the case of the transportation services, for example, the process of finding and selecting a service could be made automatically, as the client could program their applications to reason about the data that is being advertised using a semantic web language and choose which service is appropriate to perform an operation. This section discusses some of the languages currently available for the development of semantic web applications.

2.3.1 RDF

The first language to be presented is the Resource Description Framework [13], RDF. This language defines constructs that enables one to represent properties about an object in a unambiguous manner. Moreover, it is based on XML, fact that qualifies it for application integration projects.

The constructs defined in RDF allows the application designer to structure information in a subject, verb, object triple form and relies on Uniform Resource Identifiers [14], URI, to cope with ambiguities. In the transportation services example, one could use a RDF document to define that a country is part of a determined region. Code Fragment 2-1 illustrates how this could be done.

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/TR/WD-rdf-syntax#"
  xmlns:location="http://www.iao.fhg.de/Location#">
  <rdf:Description about="http://www.iao.fhg.de/Location#France">
    <location:PartOf>http://www.iao.fhg.de/Location#France_and_Germany</location:PartOf>
  </rdf:Description>
</rdf:RDF>
```

Code Fragment 2-1: RDF code example.

One application could then use the information defined in the RDF document and reason that if the company operates on a region, then it must operate inside a particular country as well. The URIs that are used to define country and region guarantee that the applications are talking about the same concepts and that these concepts are unique. And now that the application knows that the company operates in the country it needs, it can use the services provided by this transportation company.

Indeed, the applications that can derive from the use of RDF are various, but it is clear from the example above that it could be used to enhance the description, discovery and selection of web services. On the other hand, even though RDF can be used to represent a range of meaningful facts, as the complexity of the application domain increases, it becomes harder to represent the complex relations between objects using the constructs provided by the language.

If more complex applications are to be developed that process semantics, the developer will need to be able to represent these more complex relations somehow. He will need

to represent information in terms of classes and subclasses, express the relationships between them, such as equivalence relations, and provide means to represent restrictions such as cardinalities. The constructs of RDF are not able to provide such expressivity, though.

2.3.2 DAML and OWL

The DARPA Agent Markup Language [15], DAML, and the Ontology Inference Layer [16], OIL, are two research initiatives working on the development of more powerful languages that are able to express more complex semantics in a computer interpretable form. They have joined their efforts a few years ago to create a common language, DAML+ OIL [17], and develop tools [18] that provide support to the language.

In fact, DAML+OIL builds upon XML and RDF to define a set of new constructs that are able to express the concepts mentioned above. The DAML+OIL constructs enable one to define classes and properties, subclasses that extend the concepts from the parent class, relations between classes and the cardinality of these relations, as well as data types and range for the properties. The Code Fragment 2-2 shows the definition of a class Location and of a property of this class, locationName, specifying that a Location can have only one locationName.

```
<daml:Class rdf:ID="Location">
  <rdfs:label>Location</rdfs:label>
  <rdfs:comment>Definition of Location</rdfs:comment>
</daml:Class>

<daml:DatatypeProperty rdf:ID="locationName">
  <rdfs:domain rdf:resource="#Location" />
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string" />
</daml:DatatypeProperty>

<daml:Class rdf:about="#Location">
  <rdfs:comment>A location can have only one name</rdfs:comment>
  <rdfs:subClassOf>
  <daml:Restriction daml:cardinality="1">
  <daml:onProperty rdf:resource="#locationName" />
  </daml:Restriction>
  </rdfs:subClassOf>
</daml:Class>
```

Code Fragment 2-2: DAML code example.

Application designers can thus use the constructs provided by DAML+OIL to build ontologies. Ontologies are formal specifications of concepts and relations about a specific domain that can be shared and processed by applications [19]. Business partners could, for instance, define a common ontology for their business domain and so be able to plug their applications together by using the concepts defined formally in the ontology.

The applications would make use of the concepts defined in the ontology and perform reasoning operations to connect to each other. In effect, even before DAML+OIL a set of ontology languages and reasoning tools already existed. What makes this language unique is the range of constructs it provides and the fact that is based in XML, which enables it to be used to integrate applications developed in different languages and running in different platforms.

Here the Web Ontology Language [20], OWL, needs to be mentioned. This language is being developed as an alternative to DAML+OIL. In fact, the language is based in DAML+OIL and provides the same sort of constructs. The difference that needs to be pointed out is that the constructs in OWL were designed to be more condensed as the ones in DAML+OIL, fact that makes it easier to implement. [20] and [21] discuss the differences between OWL and DAML.

2.3.3 Semantic Web Services

For applications based solely on semantic web technologies, some of the problems regarding integration remain, though, as for example, how to perform remote calls in a common way, or how to provide transaction support and security. In fact, as already mentioned in section 2.2.2, these and other problems are being currently addressed by the web services research community. So, it can be said that web services technologies somehow complement the development of applications based on the semantic web.

On the other side, web services applications lack some of the functionality that is provided by the semantic web technologies. Indeed, these technologies could be used to improve different aspects of the web services applications ranging from service discovery and selection to composition and execution. They will provide these applications with the common language and reasoning capabilities they do not have to perform business integration.

In the end of the line, developers will be able to design applications that can find and select the service they need, perhaps composing them in more complex services, and call these services to perform the needed operations. But to reach this final goal, a lot of work still has to be done on both sides. To be more specific, the tools, standards and languages that support these technologies need to be improved and integrated.

In the particular case of integrating web services and semantic web technologies, the DAML Services [22] initiative, is an effort that has to be mentioned. The DAML Services work builds upon the DAML language to define the DAML-based Web Services Ontology [23] [24], DAML-S, and an OWL-based Web Services Ontology [25], OWL-S, that are ontologies that can be used to provide a complete semantic description of the web services. These ontologies and their main concepts are presented and discussed in the next section.

2.4 Semantic Description

The DAML Services is initiative founded by a group of industry and research partners to develop an ontology for the semantic description of web services. As discussed above, the DAML-S and OWL-S ontologies build upon DAML and OWL constructs and expressivity to provide more complete, semantic descriptions of the web services being described, improving the process of discovery, composition and execution of the services.

An important remark must be made here, that by the time this work is being written, no final version of the ontologies has been released. Both ontologies are currently in

version 0.9, and the group is working on the final release of version 1.0. In fact, according to the group, they have decided to concentrate on OWL-S and there will be no DAML-S 1.0 version. For this reason, this work will focus the discussion upon the OWL-S ontology.

The OWL-S ontology defines a set of classes to represent concepts that can be used to describe web services from different point of views. The top class of the ontology is the class Service. In addition, OWL-S defines the classes ServiceProfile, ServiceModel and ServiceGrounding, that represent different types of descriptions of the service. A service can be related to many instances of ServiceProfile and ServiceGrounding, and only one instance of a ServiceModel. Figure 2-7 shows this arrangement.

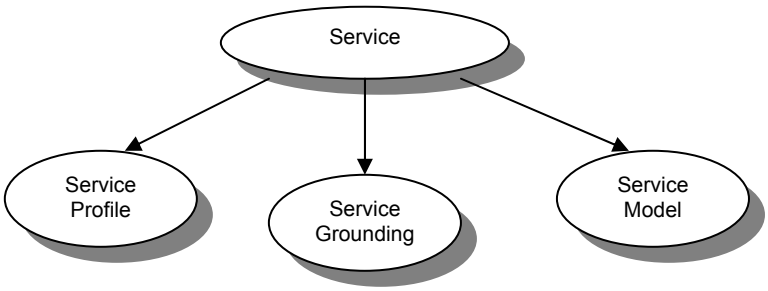

Figure 2-7: OWL-S top ontology classes.

2.4.1 Service Profile

As discussed above, these classes represent different point of views of the service. The class ServiceProfile provides a description of the service functionalities and attributes. In this way, it is very similar to the UDDI descriptions that are stored in registries. The basic difference is that UDDI does not provide for semantic description of the aspects that define a service, and so they provide more restricted search capabilities.

The ontology does not define the attributes in the ServiceProfile class itself, but it defines a Profile class as a subclass of ServiceProfile that specifies the functionalities and attributes of a service as properties. In fact, the class Profile defines a more complete description that includes aspects such as quality of service, that are not defined in UDDI descriptions, even though UDDI could be extend to do so. The Table 2-1 describes the properties defined in the OWL-S profile.

Property	**Description**
serviceName	The name of the Service.
textDescription	Text description of the service.
contactInformation	Contact information of the service provider.
has_process	Link to the process model that describes how to execute this service.
serviceCategory	External category ontologies that can be used to classify a service.

serviceParameter	Properties that characterize the execution of the service, such as response time and geographic radius.
qualityRating	Industry based ratings, such as the stars classification used in the hotel industry.
input	The inputs to the service.
output	The outputs of the service.
precondition	The preconditions to the execution of the service.
effect	The effects resulting from the execution of the service.

Table 2-1: Properties of an OWL-S Profile.

2.4.2 Service Model and Grounding

The class ServiceModel provides a description of how the service can be executed. In fact, the ontology defines a ProcessModel as a subclass of the ServiceModel. This class is responsible for describing how services are composed and how they can be executed as a single one. It provides constructs that allow the application designer to define services as atomic or composed, and the simple services that form the composed one.

The idea behind the ServiceModel is to provide a complementary description of the service, in the case the profile description is not enough. Suppose a service provider have a service that needs to make different calls to execute a single task. In this particular case, the client needs some information about how to execute the service, as for example which functions must be executed first. The profile cannot provide this information, so a process model description is needed at this point.

In the transportation services example, the transportation company would advertise its service using a profile. But the service would be composed of two different functions that must be executed in sequence. A first function would return information as price and time for the client to select, and the second function would actually place the order of transportation. This information about how to run the service is described using the process model.

The final top class the OWL-S ontology provides for service description is the ServiceGrounding. This class provides information about how a client can actually connect to the service, including addressing information, protocols and types being used. In this way, it is very similar to WSDL documents. Here, one more time, the idea is to provide a complementary description of the service, by defining data that cannot be described in a profile.

In the transportation services application example, the client application have already chosen the service it wants to execute. In addition, it also knows what functions it must call to execute the service and in which sequence. Still, it does not know how it is suppose to perform the call. The details of how to connect to service and protocols to be used are not defined anywhere. It is the service grounding description that provides this information.

2.4.3 Critics on DAML-S and OWL-S

As one might notice, OWL-S provides a rather complete model for the description of web services. Besides describing the functionalities provided by the service, it also provides a description of how services should be executed and how to actually connect to them. This kind of description collected together in a single model is not provided by the standards presented before and is one of the greatest advantages of using this ontology.

In addition, one must remember the ontology allows for the semantic description of the service characteristics. This means that the descriptions are computer interpretable and can be semantically processed. Indeed, a number of applications already discussed in the past section become possible, such as more powerful discovery mechanisms, and automatic selection and execution of the services.

Although the ontology presents a number of advantages, it also presents some weaknesses. One of the critics is that the links between the different types of description are not formally defined [26]. An example is the relation between the service groundings and process models. Even though it is clear that for each atomic process in the service model, a service grounding should exist, there is no constraint in the ontology specifying it.

A clear side effect here is the specification of erroneous service descriptions, as there are no constraints to guarantee their soundness. In fact, as no formal specification exists, tools that cope with such problems cannot be developed, unless they rely on such assumptions as the one made above. There are other such constraints in the model that are not formally specified and might lead to restrictions on the practical use of the technology.

Another critic is that the mapping between WSDL and OWL-S service grounding is not straightforward. This means that one would probably need to make changes to the OWL-S description to make the mapping to WSDL possible, but limiting the OWL-S expressivity on the other hand. Moreover, there is still very little tool support for the ontology, fact that might slow down its adoption.

Most of these critics though, come from the fact that OWL-S is a rather immature ontology. The expectation is that as the ontology becomes more mature and start to be applied on more projects, most of the weak points that exist today are going to be worked out. In addition, as more tools become available, the implementation of systems based on the technology are expect to increase. Next section takes a look at some applications based on DAML-S.

2.5 Semantic Web Applications

This section gives a brief introduction to some applications based on semantic web technologies that are currently under development. The systems presented in the following subsections implement registries based on DAML-S descriptions that are used for the discovery of web services.

2.5.1 DAML-S Matchmaker

The first application to be presented is the DAML-S Matchmaker [27]. As already mentioned above, this system can be described as a web services discovery system based on DAML-S descriptions. It uses DAML-S descriptions to enhance the discovery process by performing reasoning operations upon the information provided in the service description. The system is accessible through a web page where clients and providers can perform operations.

The model of operation is quite simple. Service providers use the system to include their service descriptions on the registry. On the other side, clients provide service descriptions according to the services they are looking for. Whenever a client submits a query, the system compares the description provided by the client with the descriptions on its database and returns a list of the services that match the request.

The real power of the Matchmaker comes from the fact that the semantic descriptions that are used to describe the services provide for a much more qualitative matching process [28]. Unlike UDDI registries, that base their searches on keyword matches, the Matchmaker allows for reasoning operations upon the descriptions lying on its database. Reasoning is possible only because semantic web ontologies are being used provide a common language for it.

Here the transportation industry example can be mentioned again to explain how these operations actually work. Supposing a client is searching for a company that provides transportation services inside a specific European country. In the registry there is no such specific description, but a company has advertised a transportation services that work Europe-wide. In this case, the registry can reason that the country is inside Europe and return the service as result.

The architecture of the Matchmaker is quite simple and could be used as a benchmark for other registry implementations. The Figure 2-8 shows the architectural organization. It includes a communication model for handling requests, a UDDI-to-DAML translator for interaction with UDDI registries, and a matching engine. The translator is needed because the system does not store information about how to connect the services and uses UDDI registries for this purpose [29].

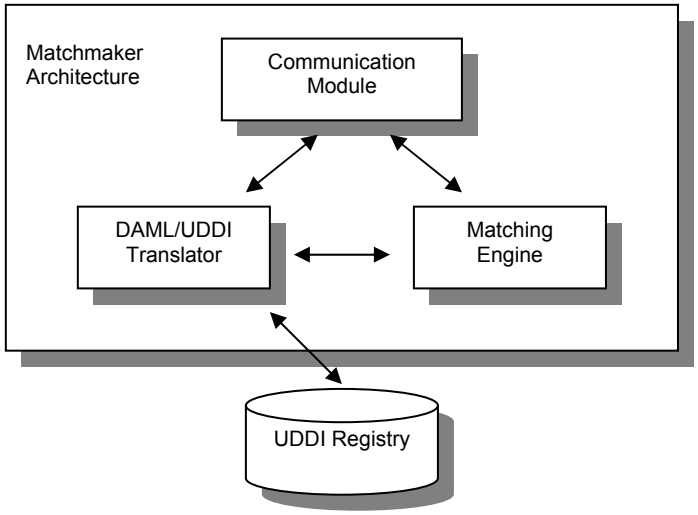

Figure 2-8: DAML-S Matchmaker architecture.

The matching engine is responsible for the matching process and can be considered as a system on itself. The architecture of the engine is shown in Figure 2-9. A main module is in charge of controlling the whole matching process and interacts with an advertisement database that stores the descriptions, and a reasoning module that performs the real matching process with the help of the ontology database.

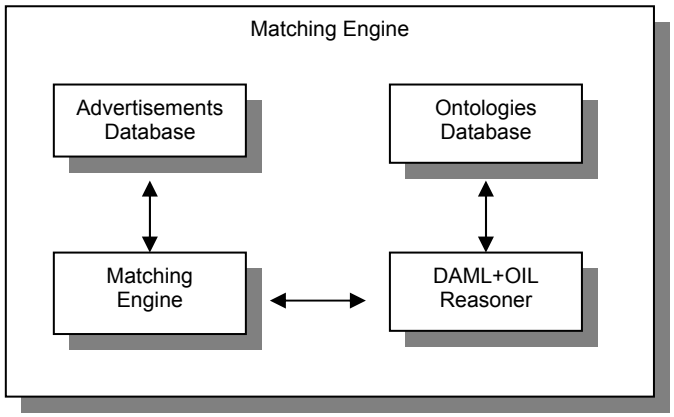

Figure 2-9: DAML-S Matchmaker matching engine architecture.

Service providers wishing to use the system can send a request to the communication module that first stores the information on the UDDI registry and then passes the information to the engine that will store the description on its database. Clients also submit queries through the communication module, that forwards it to the engine. In case a match is found, the system queries its UDDI registry to get the information on how to connect the service.

The weak point of the Matchmaker is that, as it was mentioned above, it is not able to store information on how to connect to the services, the already mentioned grounding

information. This way, the system needs to rely on a UDDI database that only slows down the whole process. Once the system becomes able to store DAML-S process and grounding information, it will turn out to be a very useful tool for automating the execution of services.

2.5.2 Semantic Discovery System

The other system that is presented is the Semantic Discovery System [30]. This system can be defined as a platform for automatic composition of web services based on the DAML-S ontology and the Business Process Execution Language for Web Services, BPEL4WS. The system allows for the definition of business processes using the language and uses a business process engine to manage the execution of the operation.

The benefit of the system is to provide an environment for dynamic composition of services using BPEL4WS, as the business process engines available today provide binding to web services only at design time. In addition, the system is able to perform discovery of services based on semantic description. The architecture of the system is shown in the Figure 2-10 and the components are discussed below.

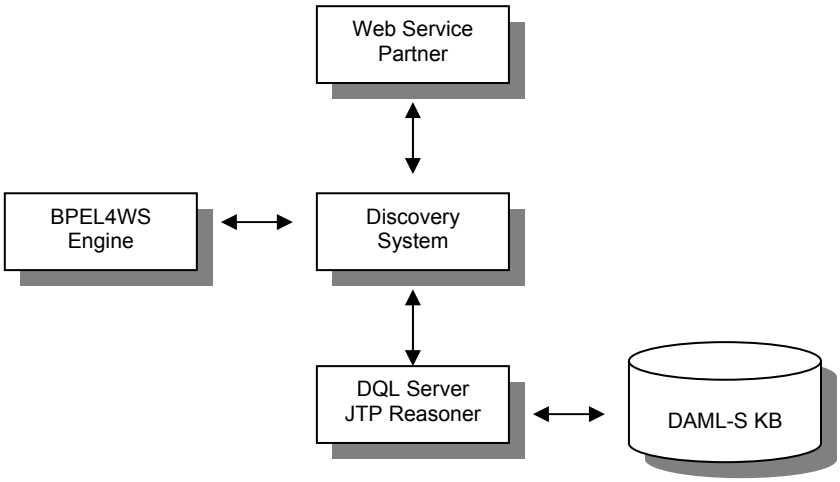

Figure 2-10: Service Discovery System.

The knowledge base is a database that stores the services descriptions. A DAML Query Language Service is the component responsible for performing the queries on the base. It is also in charge of composing the services in sequence, a process known as semantic translation. The discovery system is responsible for selecting the services and performing the calls. Finally, the process engine controls the execution of the process defined in BPEL4WS.

The process engine selects the service it needs to call and sends a request to the discovery system, that queries the DQL Server to find the services and select the ones the best fits its needs. Once the services are select, the discovery system is also responsible for performing and managing the call to these services and finally return the results to the process engine, which will then choose another service to operate.

As discussed above, to perform queries on the knowledge base, the discovery system must send a request to the DQL Server. The server is implemented over a Java Theorem Prover engine, JTP, that allows for reasoning operations over a knowledge base. Using JTP, the server is able to perform semantic queries on the registry and return the results to the discovery system that is responsible now for ranking and selecting the services.

The DQL Server and JTP are also responsible for a process called semantic translation, in which service descriptions that lie on the knowledge base can be combined in sequence in a way that they can match the actual service description request. In this case, the server returns this sequence in the result and the discovery system is responsible for managing the execution, performing the calls the right sequence described.

The weak points of this system fall upon the fact that the discovery system actually does the same type of job the business process engine is supposed to do, resulting in some repeated work. But one must take into account that this architecture is actually a workaround to allow for dynamic composition using a business process engine that does not allow to do so. Once more robust engines become available, the discovery system will be free to manage service discovery and remote calls.

2.6 Conclusion

This chapter presented the standards, technologies and systems currently under development in the areas of web services and semantic web. More specifically, the chapter started with a brief definition of application scenario on which some limitations exist today, and that could be improved by the combined application of web services and semantic web technologies. This problem definition section sets the starting point for a deeper discussion of these technologies.

The next sections discussed the basic standards of web services, and the basic concepts and languages regarding the semantic web. More ahead, the more specific OWL-S ontology is presented and discussed. Finally, a pair of applications for discovery of web services based on DAML-S are presented. As a result of the research done above, the work takes some conclusions that are discussed below.

In first place, as it can be noticed in the fair simple example provided in the problem definition, the joint application of web services and semantic web technologies can solve some limitations that exist today on enterprise application integration. In this case, one must say that the application of these technologies must be considered as a real alternative and studied more deeply.

Second, as it could be seen in the sections regarding web services and the semantic web, a lot of the work seems to be done in the area of basic standards and languages, and so the research must focus now on the development of the standards that deal with more complex issues, and perhaps even more important, on the development of tools to support the application of these technologies on real software projects.

Third, the OWL-S ontology presents itself as an important work, specially by the initiative of combining the semantic web and web services technologies. Even though

the ontology is still in an early stage, one must say that it provides a rather complete description of services. As it becomes more mature, it might turn out to be a powerful tool. Here the lack of tool support is also a problem, restricting the adoption of the technology.

Finally, the systems presented in the last section are a proof that systems based on semantic web and web services, and more specifically on DAML-S and OWL-S, can be developed and improve the applications that exist today. On the other side, they can be seen as applications that focus on solving very specific problems in some restricted environments. This means that other tools must be developed to cover the areas that are missing.

This way, from the facts exposed above, one can conclude that joint application of semantic web and web services promises great rewards, but that the lack of some specific standards and tools are holding back their adoption. This work focuses on the latter issue, by defining a Java API for discovery of services in OWL-S-based registries. The next chapter discusses the requirements and design issues regarding the tool and presents the definition of the API.

3 Service Discovery API

This chapter presents the specification of a Java API for discovery of web services based on OWL-S descriptions. The next sections are going to present a discussion about the requirements and design issues related to the development of this tool. In addition, each of the components of the API is presented, following a discussion about architectural issues and how these components can be represented. Next section discusses the general requirements for the API.

3.1 Requirements

This section discusses briefly the requirements for an API aimed at providing an interface between Java programs and remote registries based on OWL-S ontologies. Simply put, the API must provide the developer with functionality for performing service discovery in these registries. In addition, the API must provide means for the developer to perform other maintenance operations such as insertion, update and deletion of service descriptions.

3.1.1 Object Model

As discussed in section 2.4, OWL-S descriptions are XML files that use specific constructs to describe services and their characteristics. These constructs must be represented somehow inside the Java program so that they can be used to perform operations on the registry. The first requirement for the API is to provide an object model that represents the design constructs of OWL-S and allows the developer to manipulate these constructs.

The model should provide constructs that reflect the ones defined in OWL-S to make it easier for the developer to work with the tool. As an example, the model should define classes or interfaces that represents concepts such as services, profiles, service categories and parameters, and the association between them, so to make it easier to understand and manipulate these constructs.

The provision of an object model is also an important aspect because it abstract the developers from details regarding OWL-S constructs and lets the developer concentrate on the points of the description that really matter. For example, a class representing a service will abstract the programmer from details such as XML tags and attributes, and focus on the properties and methods that are actually important for manipulating the constructs.

3.1.2 File Processing

Another requirement for the API is to provide functions regarding the processing of the OWL files into the constructs defined in the object model and vice versa. To be more specific, the API must provide design constructs that define methods for reading and writing services, service profiles, service groundings and service models objects from and to OWL files lying on the file system.

Providing this functionality is important as it abstracts the developer from the details related to parsing XML files into the objects defined in the object model. This way, OWL files generated by different tools can be read into the Java program without much work and the developer can focus on the business rules. The write operations also abstracts the developer from implementation details and can be used whenever writing to files is need.

3.1.3 Registry Operations

The main objective of the API is to provide the functionality to perform service discovery and maintenance operations on remote registries. A requirement for the API is then to provide design constructs and functions that allow the developer to perform insertion, update, deletion and query operations on a remote registry. The API must provide operations that are based on the object model to be defined as part of the framework.

An important point that must be mentioned regards the granularity of the operations. A requirement here is that the API must provide methods that allow the developer to perform insertion, update and deletion operations for services, service profiles, service groundings and service models on remote registries. Query operations that return collections of services, profiles, groundings and models must also be provided by the API.

The definition of operations for these types of objects is important as they are the main design constructs in OWL-S and in fact encapsulate the other constructs defined in the ontology. This means that by performing an operation to save a profile, the program is automatically saving the associated parameters and categories, for example. Providing methods that perform operations based on more fine grained objects is not a requirement at the moment, though.

The provision of design constructs to perform operations on remote registries is important because it abstracts the developers from implementation details such as communication with the remote registry, the processing of the objects to be sent in a request, and the processing of the results received from the registry. Using these methods means that the developer can focus on the business rules while the API takes care of these implementation details.

3.1.4 Openness

As it was discussed in chapter 2, a number of tools and technologies for the semantic web are currently under research and development, including registries based on semantic web technologies. For this reason, a requirement for the API is that it should be open enough to provide implementations for the different types of OWL-S-based registries that are to become available in the future.

Openness is usually achieved by the specification of interfaces that define a basic set of design constructs and associated operations from which different implementations can be derived. Openness also means to provide for the configurability of the tool, allowing the developer to use design constructs to configure the specific points of the implementation it wants to use without needing to change the program.

As a result of this requirement, the API will be able to support different technologies of registries and abstract the application developer from the implementation details that distinguish these tools. On the other hand, the developer is going to be able to configure aspects regarding the different API implementations without the necessity to change the program.

3.1.5 Extensibility

The OWL-S specification defines a basic ontology for the semantic description of services. In fact, the ontology, although very complete, is meant solely as a basis upon which new classes and relationships can be created or derived to extend the constructs already provided to fit the specific needs of an application. It should be possible, for example, to define new types of service categories and parameters that can be used in a service profile.

A requirement for the API then is that it must be extensible enough to provide support to these new types and relationships to be created in the future. Extensibility can be achieved by the provision of general objects from which the new types can be derived, and the specification of methods based on these objects that can be overridden to support the new constructs that are to be created.

3.1.6 Security

Security is an important issue regarding applications that interact with remote parties. Remote registries should implement mechanisms that allow them to verify the identity of a client application and the operations they are allowed to perform. On the other side, the clients must be able to provide the authentication and authorization information that the registries need to authorize the operations to be performed.

The API must define constructs that support the configuration and processing of authentication and authorization information. The constructs will abstract the developer from the implementation details regarding these issues and let him concentrate on the business rules. An important requirement, though, is that these design constructs should be extensible enough to provide support to the different authentication mechanisms supported by the registries.

3.1.7 Other Issues

As the API provides functionality to perform operations on remote parties, two other aspects that need to be mentioned are performance and reliability. Performance plays an important role on performing remote functions as the network lying between the parties slows down the execution of the operation. Depending on the type of operation, the server side also plays a role in the overall performance of the system.

Also reliability is an important aspect on remote operations. In fact, it is important for the system to guarantee that messages sent between client and server do not get lost in the network for the operations to get completed. The API does not provide support in its design constructs to deal with aspects regarding performance and reliability, though.

Instead, it is up to the API implementer to guarantee that these requirements are met according to the quality of the services they wish to provide in their implementation.

Caching and message timeout mechanisms are examples of how performance and reliability can be supported by API implementations. Even though the API does not provide support for this functionality in its design constructs, still the configuration of such mechanisms should be made possible by the same type of constructs that allow for the configurability of the API implementation during program execution.

3.2 Architecture

This section introduces an overview of the architecture of the API. It presents briefly the main design constructs of the API and how they are organized. The details regarding design decisions supporting this architecture are discussed in the following sections. Figure 3-1 presents the general organization of a service discovery application that uses the API to perform operations on the registry.

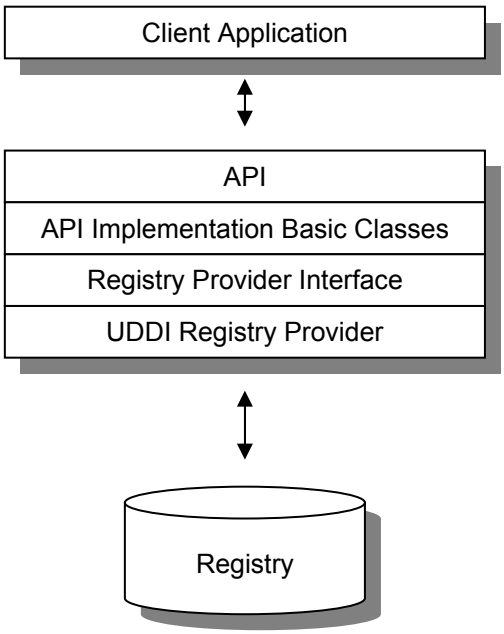

Figure 3-1: General organization of the API.

The top component in the figure is the client application that is going to use the API to interact with the registry. This component can be either a Java stand alone application or a Java enterprise component. Below the client application is the API, a set of interfaces and associated methods that the client application will use to interact with the registry. The methods provided by the API abstract the client from details regarding registry operations.

Below the API are the API Implementation Basic Classes, that implement the functionality defined in the interface. The API implementation classes delegate the task of performing operations on the registry to the Registry Provider Interface. This interface is implemented by a UDDI Registry Provider. The API Implementation Basic Classes, the Registry Provider Interface and the UDDI Registry Provider compose what is known as the API implementation.

The provider implementations, such as the UDDI Registry Provider defined above, are the components that actually carry out communication with the registry, establishing connection, sending requests to perform operations and processing the responses. Clients, API and API implementation must lie in the same Java Virtual Machine. The last component presented on the figure is the remote registry itself that is an application lying on somewhere on the network.

The set of classes and interfaces provided by the API is presented in the Diagram 3-1. The diagram shows a ConnectionFactory class that is responsible for creating connection objects. Connection objects represent a connection with the remote registry and provide access to the RegistryService, that is considered the main access point to the functionality provided by the remote registry.

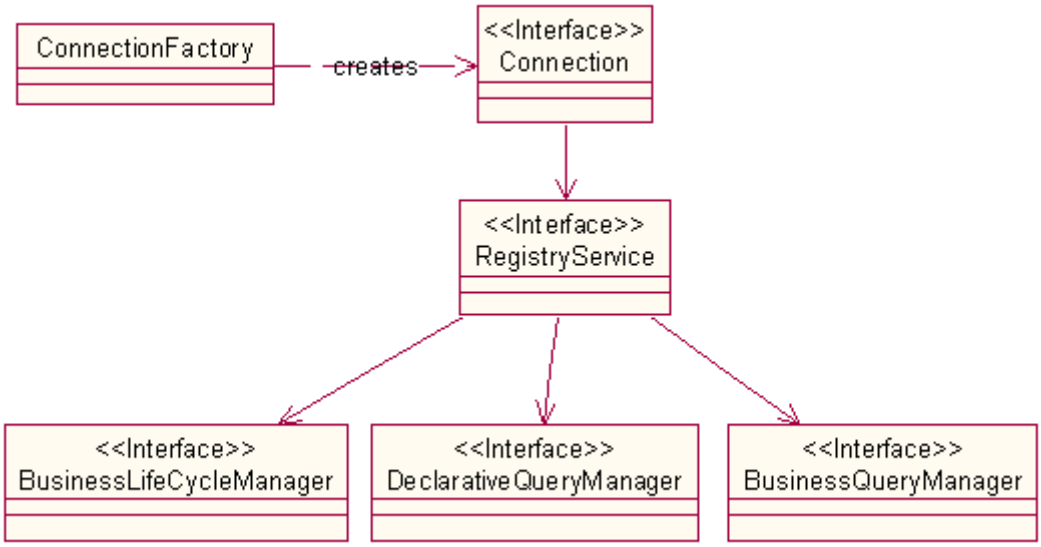

Diagram 3-1: General architecture of the API.

On the other side, the RegistryService provides access to objects that deal with specific types of functionality. The BusinessLifeCycleManager defines a set of save and delete operations. The BusinessQueryManager specifies a set of find operations that are used to query information on the registry. Finally, the DeclarativeQueryManager defines operations to that deal with performing declarative queries.

In addition to these constructs, the API defines a set of interfaces know as information model that represent the classes defined in the OWL-S ontology. These interfaces and the design decisions regarding their specification are discussed in detail in the following section. As it will be discussed in the next sections, this organization is strongly inspired

on the JAXR API [31] architecture and that is used as a benchmark and modified at some points to deal with specific aspects regarding OWL-S and the particular requirements stated in Section 3.1.

3.3 Information Model

As a first step on the design of the API, an object model needs to be defined that is able to represent the constructs of OWL-S inside a program. The application developer is going to use this model to manipulate OWL-S constructs and perform operations on the registry. This section examines the constructs defined in the ontology and presents an object model in the Java programming language. The interfaces and classes defined here are located in the package `de.fhg.iao.ws.owl.infomodel`.

3.3.1 Element and Resource

Understanding that the objects that compose the model might have properties in common, an interface Element is defined in first place. Other interfaces in the model extend this interface to inherit the methods defined on it that represent these common properties. In this point, Element resembles the RegistryObject interface defined in JAXR. In the present moment, only two properties are actually being defined, the ID of the element and an associated Resource. The Diagram 3-2 presents the model.

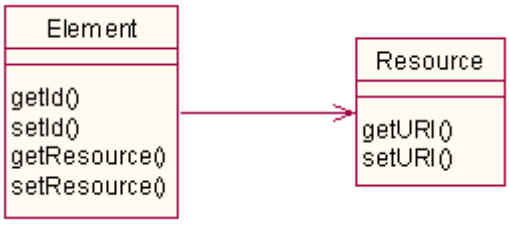

Diagram 3-2: Object model for Element and Resource

Resources are interfaces that define an URI property that can be used to identify objects uniquely. In fact, the URI defined in the resource is the concatenation of the namespace and the ID of the Element being identified. In the context of service discovery they can be used as a key for the object inside the registry. Resource and ID can be used interchangeably and the developer should decide which one to use depending on the availability of the namespace information.

3.3.2 Document

The OWL-S descriptions are usually divided into different OWL files and these files have in common a RDF root and ontology tags. To represent these constructs an interface Document is defined in the information model that contains methods to access ontology and RDF objects. These objects are instances of the Ontology and Header interfaces that are defined to represent the respective tags. The Diagram 3-3 shows the object model for these constructs.

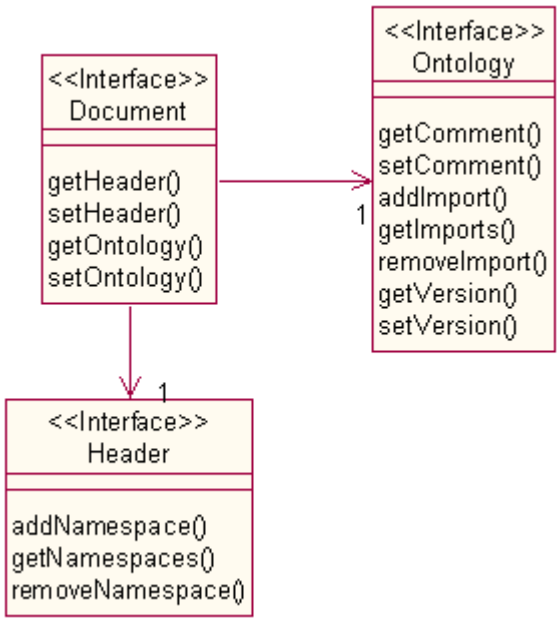

Diagram 3-3: Object model for OWL-S document information.

3.3.3 Service

The first OWL-S construct that needs to be represented is the Service class. As defined in the OWL-S specification, this class contains links to service profiles, model and groundings defined as properties. In order to represent this construct, an interface Service is defined in the information model. This interface provides methods that allow to define links to profile, grounding and model objects. This interface also extends the Document interface defined above. Diagram 3-4 presents the Service interface specification.

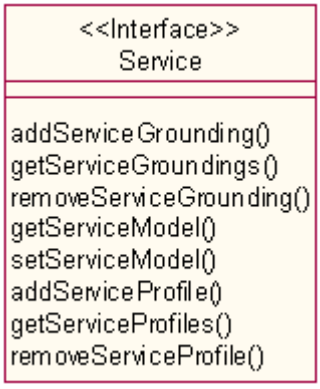

Diagram 3-4: Object model for service.

The classes ServiceModel and ServiceGrounding defined in the OWL-S specification provide additional information that can be used by the client program to execute a determined service. These classes do not play an important role in the discovery of the services, though. For this reason, no design constructs have been defined in the information model at the moment to represent them.

3.3.4 Service Profile

Two other important constructs of the OWL-S specification that need to be represented are the classes ServiceProfile and Profile. The ServiceProfile, as already discussed in the OLWS section is an empty class that can be used as a top class from which other profiles can be extended. The Profile, on the other hand, is a subclass of ServiceProfile that defines a set of properties a basic profile should have.

To represent these constructs, an empty interface ServiceProfile could be defined together with another interface Profile that extends from it. Instead, the ServiceProfile interface is defined with a single method representing the link to the Service it is associated to. In the OWL-S specification, this Service is defined as a property of the Service itself, but in an object model, this construct is better represented as a method inside the ServiceProfile class. Diagram 3-5 presents the model for service profiles.

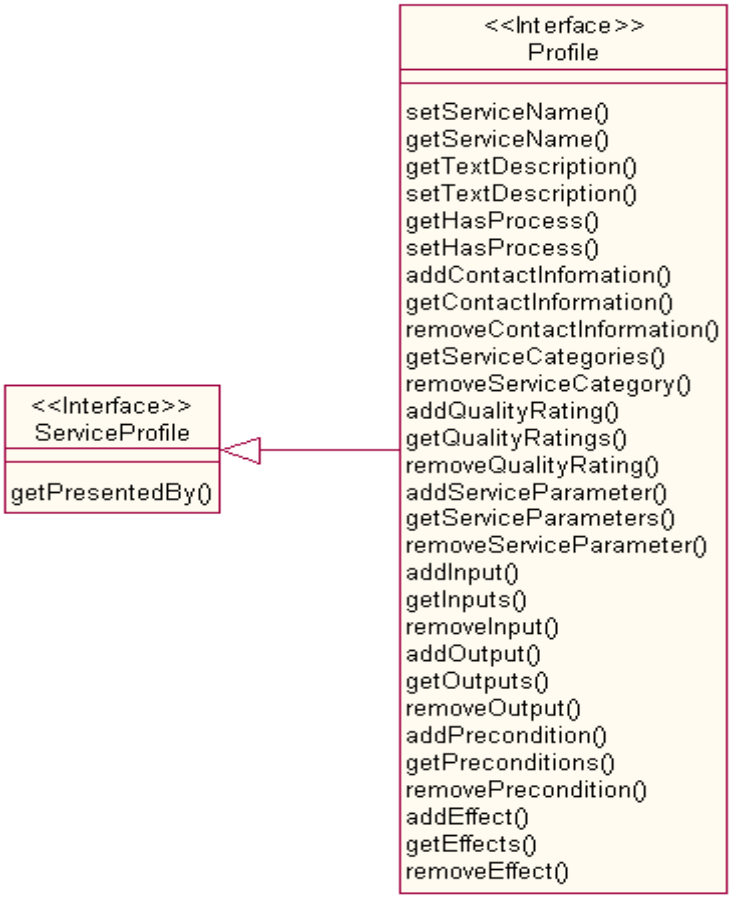

Diagram 3-5: Object model for service profile.

The Profile interface then extends the ServiceProfile by providing methods to manipulate the properties defined in the class Profile. These properties, presented in the Section 2.3, and the interfaces defined here to represent them make up the core of the information model. As some of the concepts defined in OWL-S do not map clearly into

an object model, some design decisions are discussed in order to define the constructs to be used.

3.3.5 Contact Information

The first construct to be discussed is the contactInformation property, that is defined as a property of the Profile class, having as range the class Actor. This means that the OWL-S specification itself ties the contactInformation to values of type Actor, or perhaps subclasses of Actor, and this property cannot be represented by any other type. An instance of the class Profile can contain more than one instance of contact information.

The class Actor itself is specified by a set of properties of range String and can be defined here as an interface containing a set of methods to access these properties. Here an alternative to represent the contactInformation property is to define methods in the Profile interface to access and manipulate objects of type Actor. This way, if a new interface that extends Actor is defined, one can still use the methods in Profile to manipulate contact information. This arrangement is shown in Diagram 3-6.

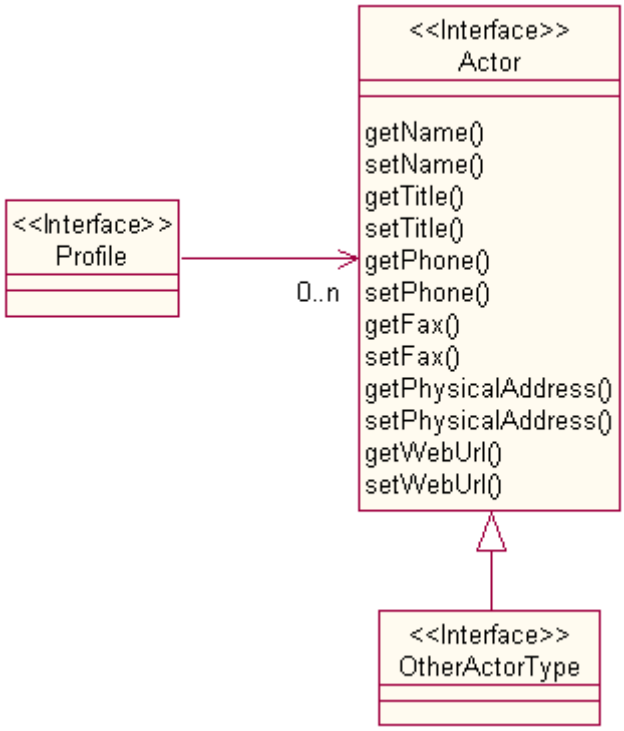

Diagram 3-6: Model for contact information.

Another possibility here would be to define an empty interface ContactInformation and make the interface Actor extend this interface. The methods defined in the Profile interface would then be used to manipulate ContactInformation objects. This would make the Profile extensible to work with other types of contact information other than Actor. Diagram 3-7 shows this new arrangement. The specification defined in this work opted for the first alternative as it reflects more clearly the OWL-S constructs.

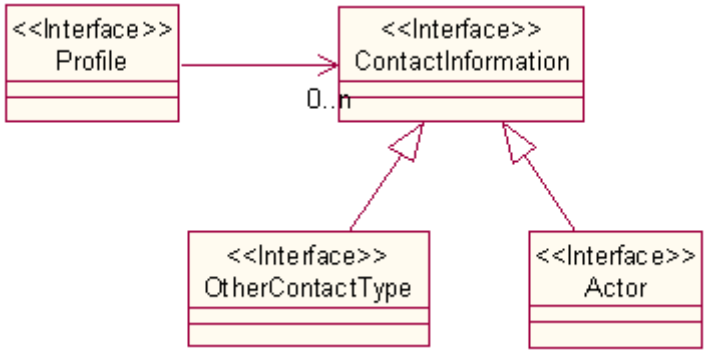

Diagram 3-7: Alternative object model for contact information.

3.3.6 Parameter Descriptions

Another set of constructs that need to be represented in the object model are the input, output, precondition and effect properties. These constructs are defined in the OWL-S specification as sub properties of a parameter, which is itself a property of range ParameterDescription. ParameterDescription, on the other hand, is a class that defines a set of basic properties that parameters should have.

Here a an option is to define a ParameterDescription interface to represent the ParameterDescription class. Methods in the interface would correspond to the properties defined in the class. This interface could then be used to represent inputs, outputs, preconditions and effects properties in the Profile. There is a number of ways this could be specified, though. Some of the possibilities of implementation are discussed below.

A first choice is to define different methods in the Profile interface, one for each type of parameter begin used. These methods would accept ParameterDescription objects as parameters and return these objects, or collections of them, as return values. This organization makes it easier for the developer to separate one type of parameter from the others and keeps the information model simple. Diagram 3-8 shows the object model for Profile and parameter description.

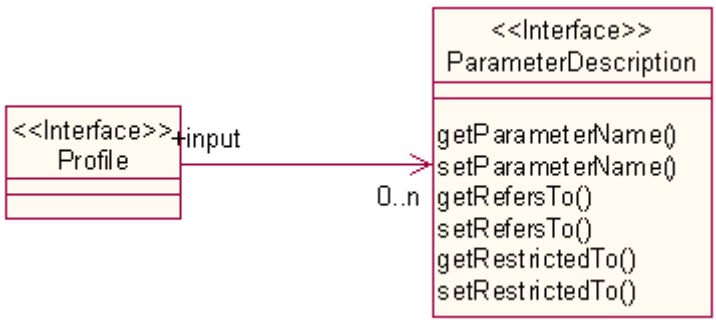

Diagram 3-8: Object model for parameter description.

Another option would be to define in the Profile interface methods to manipulate solely objects of a more general type of parameter. In this case, the specific type would need to be declared as, for example, a parameter in the method call and as a static attribute in the ParameterDescription interface. This second option actually raises the question whether the type of the parameter should be explicitly defined in the model.

Diagram 3-9 shows an alternative object model for Profile and ParameterDescription. In the figure, the Profile is represented by a ProfileWithParameters interface for simplification. The methods presented in the interface substitute the methods regarding input, output, effect and precondition in the first model.

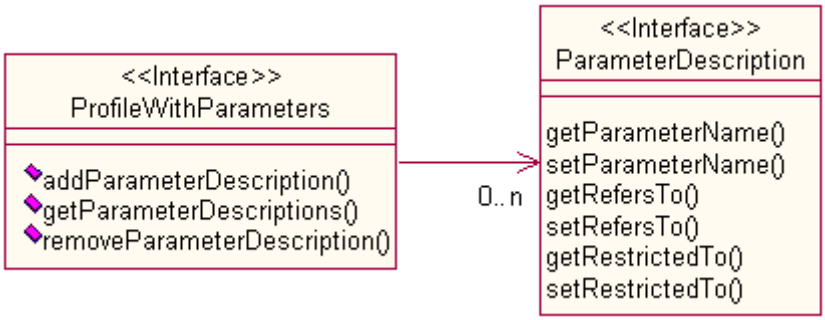

Diagram 3-9: Alternative object model for parameter description.

The first option was chosen between the two as it presents a good set of advantages. Unlike the second possibility, it presents a very simple model, easy for the developer to learn and work. The weak point lies upon the fact that it does not represent the types of parameters explicitly in the model, but this does not seem to be a problem as it defines specific methods to work with each parameter type.

3.3.7 Service Categories and Parameters

One other construct to be discussed is the serviceCategory property, defined as a property of Profile, having the ServiceCategory class as a range. This means that service categories must be either of type ServiceCategory or a subclass of it. The class ServiceCategory itself is defined by its own properties. The specification also defines some subclasses of ServiceCategory. A Profile can have any number of serviceCategory properties.

Here an option to provide an object representation to this construct is to define a ServiceCategory interface with methods representing the properties of the class, and to define methods in the Profile interface to access and manipulate these objects. Subclasses can be represented by interfaces that extend the ServiceCategory interface, and application developers could use the same methods in the Profile interface to handle these service categories. Diagram 3-10 shows the object model for ServiceCategory.

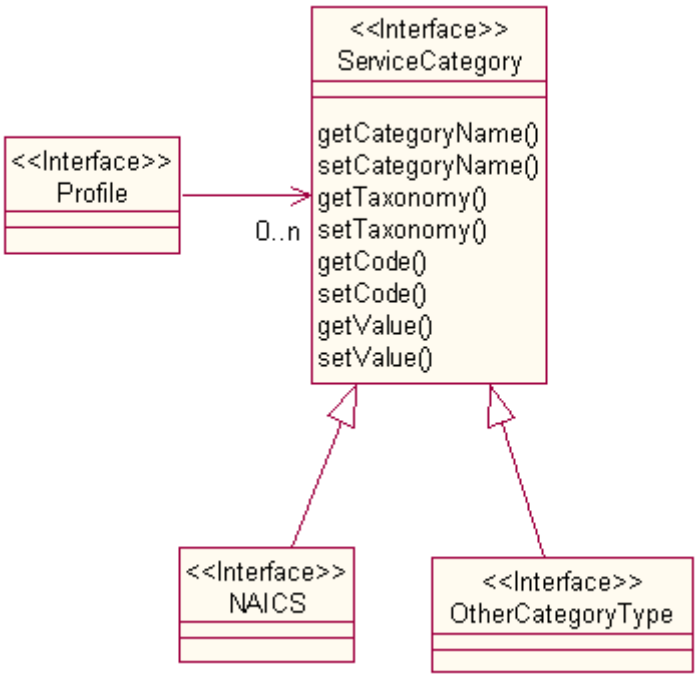

Diagram 3-10: Object model for service category.

Other two constructs are the qualityRating and serviceParameter properties. These properties present similar characteristics as the serviceCategory property. They also have a class as its range, and subclasses can be defined freely to represent new types of quality rating and service parameters. In fact, the current OWL-S specification already defines three other types of service parameter.

As the problems are very similar, a similar solution is adopted here. Interfaces are defined to represent the classes QualityRating and ServiceParameter and methods are specified in the Profile interface to represent these properties. If it is needed, other interfaces that extend from QualityRating and ServiceParameter can be created, and one could use the same methods defined in the Profile interface to access and manipulate these new types. Diagram 3-11 shows the model for these classes.

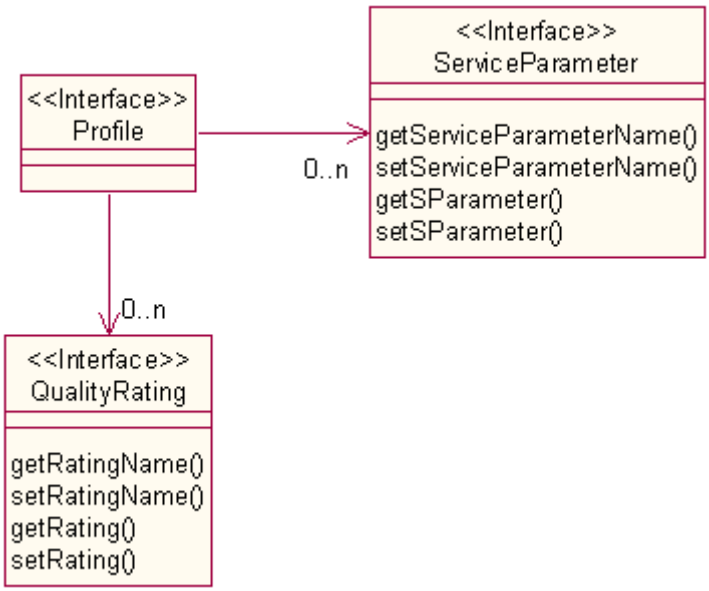

Diagram 3-11: Object model for service parameter and quality rating.

A weak point of this strategy is the constant need to use `instanceof` statements in the code to distinguish between the different types of interfaces. Another solution would be to define new methods in the Profile interface to represent each type of attribute specifically. But this option could make the Profile interface unclear with the number of methods defined on it. Here a choice is made for more clarity in the Profile interface, but this decision could be altered in the future.

This last item concludes the discussion on how to represent OWL-S constructs inside the application. Diagram 3-12 shows the complete object model for the Profile. The next section discusses how the objects defined in this information model can be actually created and defines design constructs for reading and writing these objects from and to files lying in the file system.

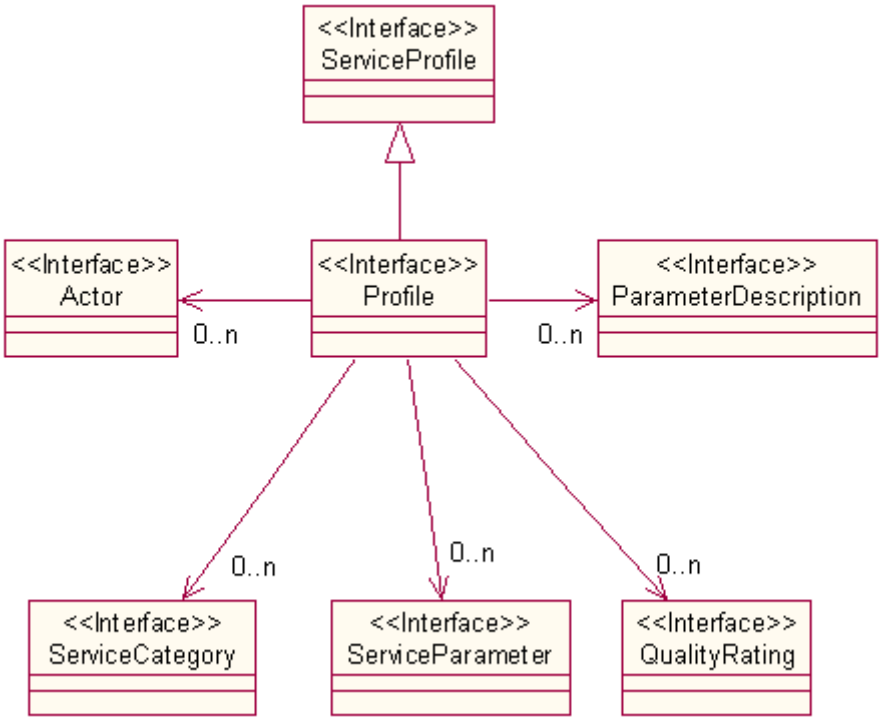

Diagram 3-12: Complete object model for service profile.

3.3.8 Creation of Objects

In order to provide for creation of the objects defined in the sections above, an interface is needed that defines factory methods for these objects. Another requirement for the API is that it should provide functionality for reading and writing these information model objects from and to files located in the file system. This section discusses some design alternatives and presents a solution that provides support to these issues.

A first alternative is to define different interfaces that encapsulate the factory methods and read and write operations, that can be accessed through the RegistryService interface. Diagram 3-13 shows this alternative reflected on the object model for the API. This solution allows for the configuration of the factories and file processors through the properties of the ConnectionFactory.

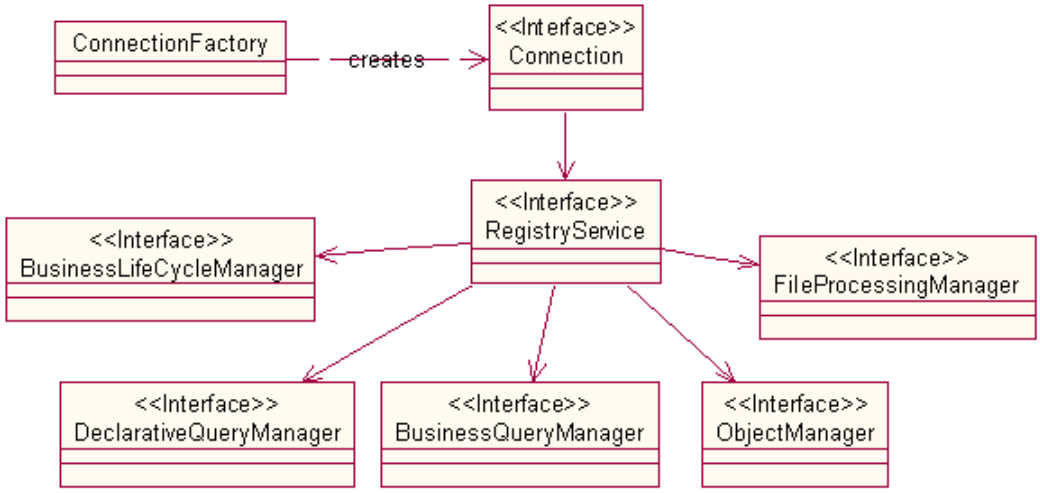

Diagram 3-13: Object model with ObjectManager and FileProcessingManager.

A second alternative is to define distinct and independent abstract classes that define abstract factory and file processing methods. This arrangement is show in Diagram 3-14. The API implementer must extend these abstract classes to provide an implementation of the factory and file processing functionality. Here also properties could be defined that support the configuration of factories and file processors.

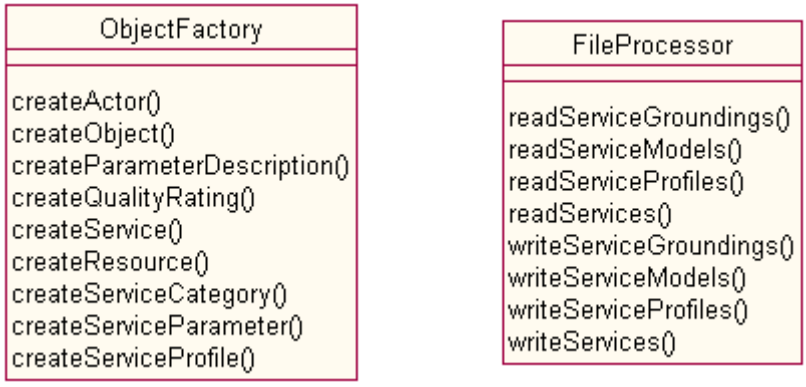

Diagram 3-14: ObjectFactory and FileProcessor interfaces.

The ability to provide for configuration is an important feature as it allows the developer to setup characteristics of the implementation such as the parsers to be used in the case of file processing. This functionality does not seem to be a problem as it is supported by both design alternatives presented above. The point that distinguish these alternatives is that the first one ties the interfaces to a RegistryService object.

This means that the developer will need to create a connection with a remote registry every time it needs to create objects or read and write files on the local file system. This is a clear disadvantage as in the case where the developer needs only to manipulate objects locally. Furthermore, as it was stated in the introduction to this work, the API shall be expanded in the future to provide for functionality other than service discovery.

In this case, coupling the creation of objects and file processing functionality to the design constructs that deal specifically with service discovery and registry operation issues is not a good solution. For this reason, a decision has been taken to follow the second alternative and provide specific abstract classes to handle these functionalities. To be more specific, the classes ObjectFactory and FileProcessor have been defined.

The ObjectFactory defines methods that allow for the creation of objects. The methods are described in Table 3-1. It provides methods for creation of the objects defined in the information model, but also provides a general method that allows for the creation of general Object objects based on a given interface name. This feature provides extensibility to the API as the factory does not need to change to support new types.

Method	**Description**
createActor	Creates an object of type Actor.
createObject	Creates an object of the type given as parameter.
createParameterDescription	Creates an object of type ParameterDescription
createQualityRating	Creates an object of type QualityRating. Here the type of quality rating should be indicated as a parameter.
createResource	Creates an object of type Resource.
createService	Creates an object of type Service.
createServiceCategory	Creates an object of type ServiceCategory.
createServiceParameter	Creates an object of type ServiceParameter.
createServiceProfile	Creates an object of type ServiceProfile.

Table 3-1: ObjectFactory methods.

The FileProcessor class defines abstract methods for reading and writing objects on the files located in the file system. The methods are described in Table 3-2. The class defines methods for reading and writing specifically service, service profile, service model and service grounding objects.

Method	**Description**
readServiceProfiles	Read the service profiles described in a file into ServiceProfile objects.
readServiceGroundings	Read the service groundings described in a file into ServiceGrounding objects.
readServiceModels	Read the service models described in a file into ServiceModel objects.
readServices	Read the services described in a file into Service objects.
writeServiceProfiles	Write ServiceProfile objects into a file.
writeServiceGroundings	Write ServiceGrounding objects into a file.
writeServiceModels	Write ServiceModel objects into a file.
writeServices	Write Service objects into a file.

Table 3-2: FileProcessor methods.

This section discussed the design of an information model and the provision of design constructs for creating these objects as well as OWL-S file processing functionality. The sections to come will focus on constructs regarding registry functionality. To be more specific, the next section defines constructs for dealing with the management of connections with the remote registry.

3.4 Connection Management

If one needs to communicate to a remote partner, it must first establish a connection with this partner. The connection establishment is an important step because it determines how the communication between the partners is going to be carried out. Before the communication takes place both partners must agree on details such as communication protocols and data format. Also during connection some resources may be allocated to improve operation management.

The tasks related to establishing connections and allocating and managing resources are put together under the term connection management. Software applications usually rely on specific software to perform connection management operations while they concentrate on business rules. An API that wishes to communicate and perform operations on a remote registry must provide a way to manage connections.

The API must provide means so that one can configure connection characteristics, create connections with the partner, change connection characteristics during communication if needed, and closing it when the operation is finally done. This facility should abstract developers from connection details and let them focus on business rules. In the next sections a solution to the connection management problem in the context of the API is discussed. The interfaces and classes defined here are organized in the `de.fhg.iao.ws.owl.registry` package.

3.4.1 Connection Architectures

The first task on defining a connection management mechanism is studying the existing APIs the offer this functionality. As a number of frameworks are actually available ,this study focuses only on the ones related to web services remote calls, namely Apache Axis [32] and SAAJ [33], and the more specific registry APIs, UDDI4J [34] and JAXR.

The Apache Axis offers a very simple mechanism, where a class Service is created and used as a factory to create Call objects. The Service provides a method that allows one to configure the connection engine by setting a specific interface that contains configuration information. In addition, the Call objects provide specific methods that allows to configure the call itself, before an invoke method is called and the remote operation is performed. The operation is described in the Code Fragment 3-1.

```
Service  service = new Service();
service.setEngineConfiguration(config);
Call call = (Call) service.createCall();
call.setTargetEndpointAddress( new java.net.URL(endpoint) );
call.setOperationName(new QName("http://www.iao.fhg.de /", "order"));
```

Code Fragment 3-1: Creating a connection using Apache Axis.

The SAAJ provides a similar mechanism. Here a SOAPConnectionFactory abstract class can be used to create SOAPConnection objects. These objects provide methods to make calls to the remote partner and close the connection when the work is done, but no methods to configure the connection are actually provided. This means the configuration is made by the parameters of the call, a SOAPMessage object and a object that represents the endpoint URL. The Code Fragment 3-2 describes the operation.

```
SOAPConnectionFactory factory = SOAPConnectionFactory.newInstance();
SOAPConnection connection = factory.createConnection();
URL endpoint = new URL("http://www.iao.fhg.de/ws/order");
SOAPMessage response = connection.call(request, endpoint);
```

Code Fragment 3-2: Creating a connection usgin SAAJ.

The UDDI4J API defines a UDDIProxy class that provides a method to configure the remote call to registry by the means of general Properties object. This object contains a set of key-value String pairs that can be used to represent different properties. In addition, it also provides methods to perform operations in the registry, which turns out to be the greatest difference comparing it with the other two frameworks presented above.

```
UDDIProxy proxy = new UDDIProxy();
Properties properties = new Properties();
properties.setProperty("org.uddi4j.inquiryURL", "http://www.iao.fhg.de/ws/registry");
proxy.setConfiguration(properties);
BusinessList bl = proxy.find_business("S", null, 0);
```

Code Fragment 3-3: Creating a connection using UDDI4J

In this code fragment, a UDDIProxy object is created that is to be used for performing operations on the remote registry. Then, a Properties object is created and a property indicating the URL of the registry is set to it. This object is then attached to the UDDIProxy by using the set configuration method. Finally, the developer can use the proxy to perform the find business operation on the registry.

The JAXR API also provides a connection management mechanism. It presents a ConnectionFactory object that enables the configuration of the connection using a Properties object just as UDDI4J. In addition, the Connection object that is created provides methods to change the configuration of the connection in runtime. Moreover, the methods used to perform operations on the registry are not defined directly in the Connection interface, but can be reached through a RegistryService interface. The Code Fragment 3-4 shows an example of JAXR.

```
ConnectionFactory factory = ConnectionFactory.newInstance();
Properties properties = new Properties();
properties.setProperty("javax.xml.registry.queryManagerURL",
       "http://www.iao.fhg.de/ws/registry");
factory.setProperties(properties);
Connection connection = factory.createConnection();
RegistryService registryService = connection.getRegistryService();
```

Code Fragment 3-4: Creating a connection using JAXR.

3.4.2 Evaluation of the Architectures

By analyzing the frameworks described above an important pattern is revealed. This pattern is the use of a factory class to create the objects the actually provide the connection management functionality. These factories are used to configure the properties of the connection before it is actually created to perform operations. The exception to this rule is SAAJ that provides no direct means for connection configuration.

Another exception is UDDI4J that provides no connection factory at all. The connection objects are created using a constructor and provide the necessary methods to configure the connection if any configuration is needed. Important here is that, just like JAXR, it makes use of a Properties object that gives the developer flexibility to define the properties it wants to. The properties supported by the connection will depend on the implementation of the API, though.

Another pattern that becomes clear is the use of these connection objects to perform the actual communication with the remote service. This means that these objects provide a single method to make a direct call to a remote service that is going to perform the operation. The exception to this rule is JAXR that provides access to a more abstract RegistryService object. This object is used to access different interfaces that define the methods to perform operations on the registry.

As a result of the analysis, JAXR seems to present better characteristics in relation to the others. On one hand, the separation between connection factory and connection classes provide actually a separation between configuration and runtime management of the connection. On the other hand, the decoupling between connection functionality and registry operation functionality can also be seen as a good property of the API.

Moreover, the JAXR API includes a series of characteristics that classify it as a good model to be followed. In first place, it is an API for accessing registries, similar functionality to being designed in this work. Second, JAXR is organized as a set of interfaces that can be implemented by different vendors using different technologies. Finally, the use of design constructs similar to JAXR can make the shift from JAXR to the service discovery API easier to the developer.

3.4.3 Connection Management Architecture

For the reasons discussed above, this work decided to follow the JAXR connection management model, making changes whenever necessary. The connection management classes and interfaces are defined in the package `de.fhg.iao.ws.owl.registry`. The discussion that follows introduces the classes and interfaces that make up the connection model of the service discovery API and discuss its strong and weak points. Diagram 3-15 presents the connection management architecture for the API.

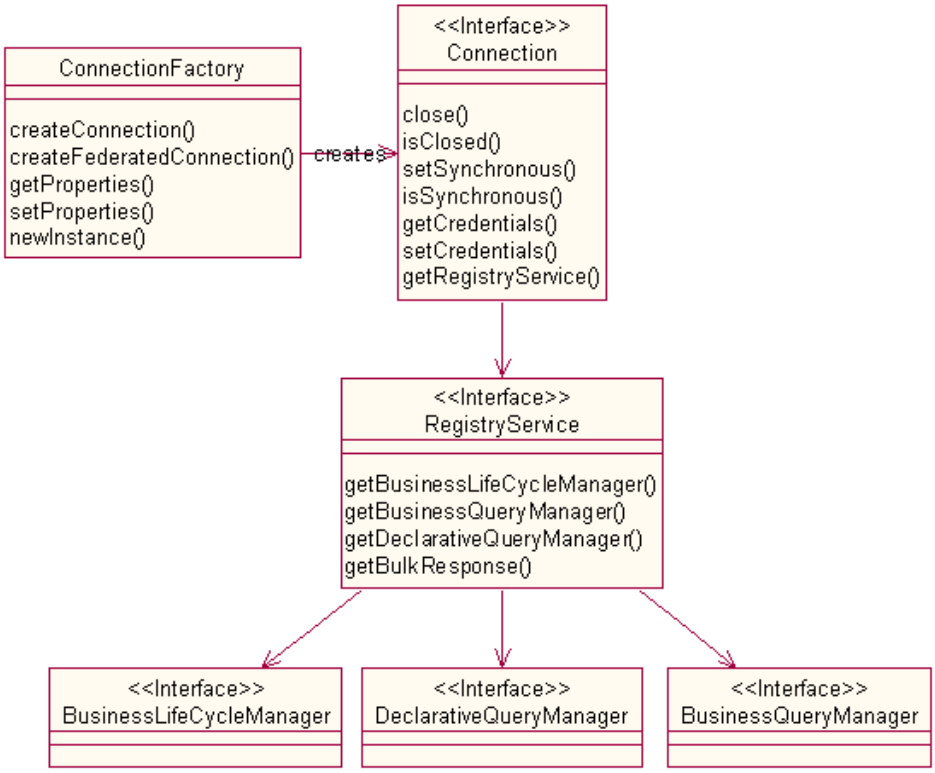

Diagram 3-15: Object model for connection management.

The ConnectionFactory is an abstract class that defines methods for the creation and configuration of connections. Connections can be created using the create methods, and the set and get properties methods are used to configure the connection being created. The properties that are allowed to be supported are defined by the implementation of the API. The Code Fragment 3-5 shows an example of configuring and creating a connection using the service discovery API.

```
ConnectionFactory connectionFactory = ConnectionFactory.newInstance();
Properties properties = new Properties();
properties.setProperty("javax.xml.registry.queryManagerURL",
       "http://localhost:8080/RegistryServer");
connectionFactory.setProperties(properties);
Connection connection = connectionFactory.createConnection();
```

Code Fragment 3-5: Creating a connection using the API.

On the other hand, the interface Connection defines methods to manage the connection. It provides methods to set and get client credentials, information used to authenticate the client application with the registry. In addition, it provides methods to check and change the mode of communication to synchronous or asynchronous on runtime. Finally, it provides methods to check if a connection is open or closed and to close the connection.

The actual realization of these methods is going to depend on the implementation of the API, though. The API implementer might decide not to implement the setSynchronous

method, for example, and define a default value for the communication model. In this case a call to the method should return an exception. The close method, on the other side, must be implemented. The API implementer must check the documentation to know which methods are mandatory.

Another interface defined in the model is the FederatedConnection. It is actually an empty interface and inherits its behavior from the Connection parent interface. This interface is used solely to represent a connection with a group of registries that share the same set of connection properties. It can be used to support distributed queries inside a same connection, but it is not required to be implemented by the API implementer.

3.4.4 Registry Service

The Connection interface defines a last method that returns a RegistryService object. The interface RegistryService encapsulates the functionality related to performing operations on the registry. But instead of defining methods to perform the operations directly, the methods defined on it provide access to more specific objects that are responsible for specifying the actual methods that are going to perform operations on the registry. The Code Fragment 3-6 shows an example of how registry services are used.

```
RegistryService registryService = connection.getRegistryService();
BusinessLifeCycleManager lifeCycleManger =
       registryService.getBusinessLifeCycleManager();
```

Code Fragment 3-6: Creating a manager using the RegistryService.

The interfaces that define the methods to perform operations on the registry are the BusinessLifeCycleManager, the BusinessQueryManager and the DeclarativeQueryManager. The life cycle manager defines methods that deal with save and deletion operations. On the other hand, the business and declarative query managers define methods for executing queries upon the registry.

These interfaces are going to be discussed in more detail in the sections ahead. A more important question to be answered here is whether they are necessary or not. The methods could simply be defined in the RegistryService interface without adding more complexity to the model. In fact, this is exactly the way that UDDI4J defines such methods, specifying all of them in the UDDIProxy class.

The answer to this question is going to depend simply on the number of methods that need to be defined, and if more methods might be defined in the future. In the JAXR case, a whole set of methods were defined and a correct decision was made to break up the functionality into three other interfaces that deal specifically with each type of operation. This is also the case for the API defined in this work.

Finally, the RegistryService defines a last method that is related to operations in a registry but does not represent a registry operation itself. This is the getBulkResponse method that receives an ID as parameter and returns a BulkResponse. This method should be implemented in the case the API implementation supports asynchronous communication. BulkResponses are discussed in detail in section 3.7.

Put together, the interfaces defined in this section should provide a tool for creating and managing connections with different remote registries. The configuration of the connections by the use of properties of the ConnectionFactory is abstract enough to cope with different types of API implementations. The Connection interface also defines a simple set of methods to manage the connection that should suffice.

Moreover, as it was already discussed above, the Connection interface defines also a method that gives access to the objects that are going to perform operations on the registry. These objects were already briefly discussed above and are going to be described in more detail in the following sections. The next section focuses on the life cycle management interfaces, that deals with life cycle operations.

3.5 Life Cycle Management

Life cycle management is the term used in this work to identify the functionality related to performing life cycle operations on remote registries. These are operations that involve the insertion, update and deletion of objects on the registries. One of the main requirements of the API is exactly to provide the functionality regarding life cycle operations. This section will examine some design alternatives for providing life cycle management inside the API. The interfaces defined in this section are organized in the `de.fhg.iao.ws.owl.registry` package.

The first alternative to be considered is to provide this functionality as part of the RegistryService interface. As it was defined in the past section, this interface is the main access point to the functionality provided by the remote registry and so the methods regarding life cycle operations could then be specified on it. A problem with this design is that defining the methods on the RegistryService could make the interface rather unclear.

In fact, according to the requirements, the API should provide methods for inserting, updating and deleting service, service profile, service model and service grounding objects from the registry. In addition to these, new methods could also be defined in the future. So, a design is needed that provides a clear interface to the developer and is at the same time extensible.

3.5.1 Life Cycle Manager

A second alternative is to design a specific interface that defines solely life cycle management functionality. In this case, the RegistryService would provide access to this specific interface instead of defining the life cycle operations directly on its signature. This design solves the problems of the past alternative as it provides a clearer set of interfaces that define specific functionalities and can be more easily extended without affecting the overall design.

In fact, the JAXR API provides a similar solution to this problem by defining interfaces that encapsulate the functionality of life cycle operations. These interfaces are called life cycle managers. To be more specific, two different constructs are defined in the JAXR API. The LifeCycleManager is a super interface that defines factory methods for the creation of objects and general methods for performing life cycle operations. The

BusinessLifeCycleManager extends the latter to provide operations based on specific types of objects.

The API defined in this work follows this model and defines the interfaces LifeCycleManager and BusinessLifeCycleManager to provide the functionality regarding insertion, update, and deletion on remote registries. Diagram 3-16 presents the class model for this design. As it can be seen, the LifeCycleManager specifies solely methods to save and delete general objects from the registry, and in this point it differs from the JAXR interface.

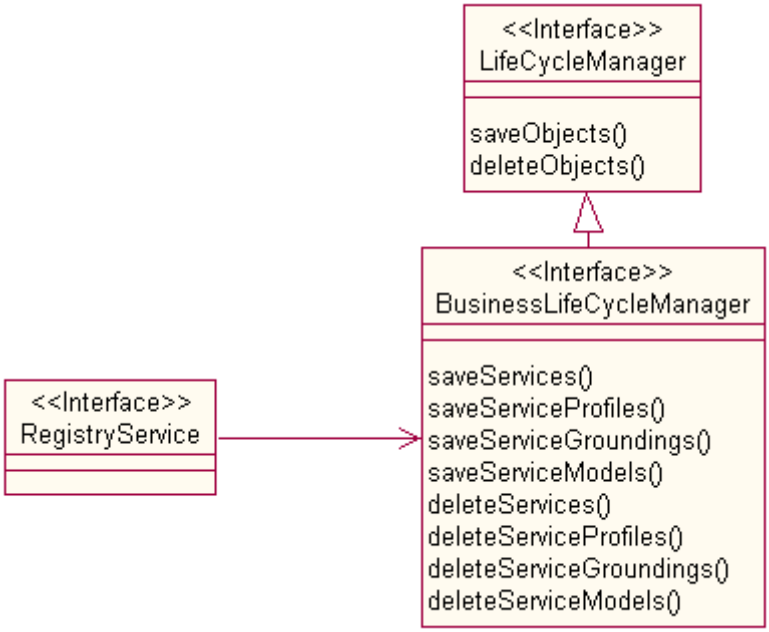

Diagram 3-16: Object model for life cycle management.

In effect, the JAXR LifeCycleManager interface defines not only save and delete operations on its signature, but also factory methods that are responsible for creating the objects defined in the JAXR information model. The reason for not specifying factory methods on the manager defined in this work comes from the fact that the interfaces defined in the information model must be independent of the service discovery API. This means that the developer should not be required to create a connection with a remote registry when he only needs to create and manipulate these objects locally.

3.5.2 Business Life Cycle Manager

The BusinessLifeCycleManager interface defined on Diagram 3-16 defines methods that are used to perform life cycle operations on the remote registries that are based on objects defined in the information model. To be more specific, it defines methods that allow the developer to perform insert, update, and delete operations of service, service profiles, service groundings and service models on the registry. Table 3-3 summarizes the methods defined in this interface.

Method	Description
deleteServiceGroundings	Delete ServiceGrounding objects from the remote registry.
deleteServiceModels	Delete ServiceModel objects from the remote registry.
deleteServiceProfiles	Delete ServiceProfile objects from the remote registry.
deleteServices	Delete Service objects from the remote registry.
saveServiceGroundings	Insert and update ServiceGrounding objects on the remote registry.
saveServiceModels	Insert and update ServiceModel objects on the remote registry.
saveServiceProfiles	Insert and update ServiceProfile objects on the remote registry.
saveServices	Insert and update Service objects on the remote registry.

Table 3-3: BusinessLifeCycleManager methods.

The BusinessLifeCycleManager does not provide any additional methods for performing life cycle operations based on information model objects. If a developer needs to perform an operation on an object of finer granularity such as a ServiceCategory, the he must retrieve a Profile, make the modifications and save the object back on the registry. Another option is to use the methods provided by the LifeCycleManager that receive objects as parameters.

The interfaces and methods discussed in this section define only methods for life cycle management. Methods for performing query operations are defined in a different set of interfaces. These interfaces have been organized together in this API under the term query management. The interfaces and their methods are discussed in the following section.

3.6 Query Management

The term query management is used to identify the functionality related to performing query operations. In fact, providing functions that can be used by the developer to code operations that perform queries on remote registries is the main goal of the API being defined in this work. This section discusses some alternatives of design constructs for query management and presents a set of interfaces that make up the query management model of the API. The interfaces defined in this section are located in the `de.fhg.iao.ws.owl.registry` package.

As it was discussed in the requirements presented on section 3.1.3, the API must provide a set of operations that query the registry for objects defined in the information model. In addition, the API should also provide support for declarative queries. Moreover, the query management model must be extensible in a way that allows for the definition of new types of queries that might to be specified in the future.

A first option is to place the methods on the RegistryService interface. This interface is the main point of access to the functionality provided by the registry, including save, delete, and search operations. But as already discussed on the section 3.5.1, defining all

the methods regarding the registry operations on this single design construct results in a rather unclear interface. Furthermore, adding new methods would make the interface even more unclear.

3.6.1 Query Managers

Another option is to create a new interface that encapsulates the query management functionality. In fact, interfaces that encapsulate life cycle operations were defined in the past section. In this case, the RegistryService provides access to these interfaces, and indirectly to the registry functionality. The advantage of this arrangement is to provide a clearer interface for the RegistryService while separating life cycle and query operations in specific design constructs resulting in a better organization of the API. Diagram 3-17 shows this arrangement.

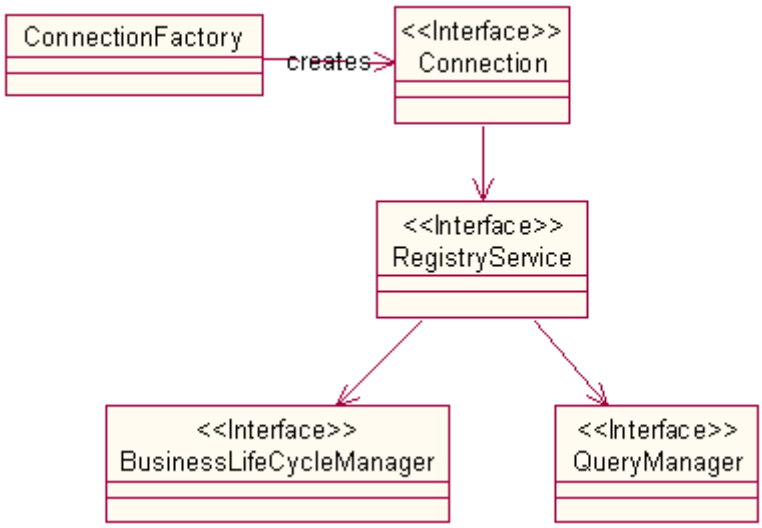

Diagram 3-17: Alternative design for query management.

The JAXR API defines a model of query management that is similar to this second option. In the JAXR model, two additional interfaces are defined that specify methods to perform business and declarative queries, providing for a better separation of the functionality provided by the JAXR API. The QueryManager is defined as a super interface that encapsulates the functionality that is common to both interfaces.

From the design options introduced above, the JAXR API query management model presents the best set of characteristics, as it provides for extensibility and better organization of the operations on the set of interfaces it defines. In addition, the functionality defined on this model is similar to the one that is to be provided by the service discovery API. For this reason, the API defined in this work will use this model as basis for the its query management design. Diagram 3-18 shows the service discovery API query management model.

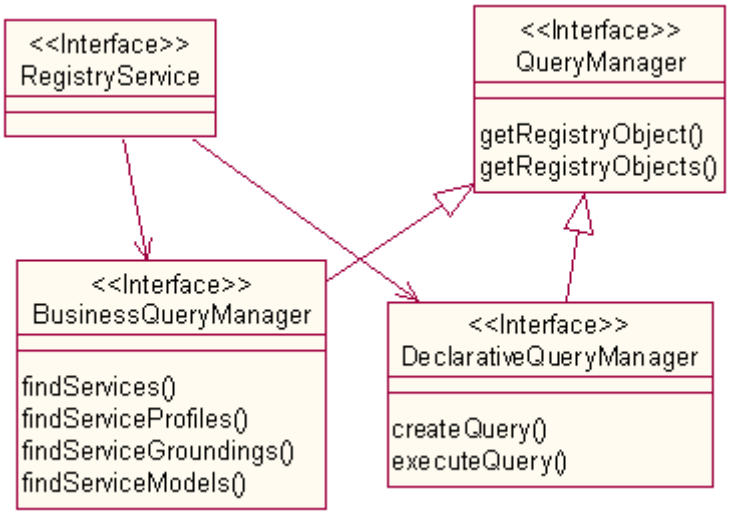

Diagram 3-18: Object model for query management.

In the model defined in this work, the RegistryService interface specifies methods that provide access to the query managers. Query managers are the interfaces that define the query operations supported by the API. The super interface QueryManager defines methods that allow the developer to query the registry for specific objects based on the key of these objects. These methods allow to retrieve objects of any type defined on the information model.

3.6.2 Business Query Managers

The BusinessLifeCycleManager defines a set of methods that allows to search and retrieve objects of a specific type based on a service profile and a set of filters. The methods are summarized on Table 3-4. Filters are used to reduce the scope of query and can improvement the performance of the operation. An example of a filter is the number of objects returned on a query or the number of profiles to be considered as input.

Method	Description
findServiceGroundings	Query for ServiceGrounding objects on the remote registry based on profile information.
findServiceModels	Query for ServiceModel objects on the remote registry based on profile information.
findServiceProfiles	Query for ServiceProfile objects on the remote registry based on profile information.
findServices	Query for Service objects on the remote registry based on profile information.

Table 3-4: BusinessQueryManager methods.

The types of filters that can be used are not defined as part of the API specification, though. The reason for not defining specific filters in the specification comes from the

fact that currently, there is no single standard for the types of filters that could be used. The Matchmaker tool defines a set of filters [35] that can be used in their searches, for example. Some of these filters are described in Table 3-5. But there is no standard specification available at the moment. For this reason, this work leaves up to the API implementer to define the filters his implementation will support.

Filter	Description
Consider Limit	Maximum number of advertisements to be considered (inputted) at each filter.
Accept Limit	Maximum number of advertisements to be accepted (outputted) at each filter.
Return Limit	Maximum number of the results to be returned from a search call.
Similarity Scope	Distance to be searched in ontology tree at Similarity filter.
Subsumption Scope	Distance to be searched in ontology tree at Subsumption filter.

Table 3-5: Matchmaker filters.

3.6.3 Declarative Query Managers

The DeclarativeQueryManager interface provides two methods that are to be used for performing declarative queries on the registry. The methods are listed on Table 3-6. The createQuery method is used to create Query objects. The Query interface defines methods that indicate the type of query being defined and the String that represents the query.

Method	Description
createQuery	Creates a Query object that encapsulates information about the query to be sent to the remote registry.
executeQuery	Executes a query on the registry based on a Query object it receives as a parameter.

Table 3-6: DeclarativeQueryManager methods.

A Query object contains then the String representing the query to be performed. The other method defined in the QueryManager interface, executeQuery, is responsible for receiving this Query object as parameter and performing the query in the registry. The query languages supported are defined as a class variable in the Query interface. If one needs to extend the API to work with new languages it only needs to add a new variable.

3.7 Responses and Exceptions

The remote registry must return as the result of an operation either the data expected by the client, or some indication of error in the case the operation has not been not completed successfully. Moreover, there will be cases where part of the operation has been completed, while another has resulted in errors, as for example, saving only part of the services sent in a collection on a save services operation. Finally, support must be provided to the responses regarding asynchronous calls.

3.7.1 Response Model

This section discusses the design of a response model that deals with the problems described above. The first problem in designing a response model is how to represent responses that deal with partial results and asynchronous calls. Dealing with partial results means that the response must return somehow the results expected by the client as well as information regarding the errors occurred during operation.

On the other hand, to support asynchronous calls is necessary to provide a mechanism that allows the application to identify uniquely a response to the particular request it has made, so that it can use this identifier to retrieve the results later. An alternative to solve this problem is to modify the objects defined in the information model to support partial results and asynchronous calls. A second alternative is to create specific design constructs to represent these responses.

The first alternative has the advantage of providing the developer with responses based on the object model. The problem is that there is no design construct in the model that could be used to represent partial responses and so one would need to be created. In addition, the objects would need to be modified to provide for asynchronous calls. The weak point of this strategy exactly coupling the object model to the representation of responses to registry operations.

The second solution discussed above is to create a design construct independent of the information model that is able to cope with the partial responses and provide support to asynchronous calls. Here an option is made for the latter to provide for the independence of the object model. Based on the response model of the JAXR API, a couple of design constructs were created to represent the responses. They are show in Diagram 3-19.

The first construct to be presented is the RegistryResponse interface. This interface represents responses to calls made to the registry. It defines a specific method for checking the availability of a response. This method is to be used in the case of asynchronous calls. In these cases, the client that made a call some time ago, can use this method to check if the response is already available.

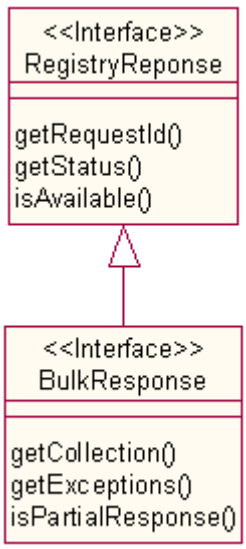

Diagram 3-19: Object model for responses.

The interface also defines a `getStatus` method and class variables for representing the status of a response. This functionality is of use for the cases of partial response and asynchronous calls, as one can check for both, errors in partial responses, or availability on the calls. Finally, it also defines a method to access the ID of the request. This ID is of use to the method getBulkResponse, already mentioned in section 3.4.3.

Another construct, and the most important one, is the interface BulkResponse. This interface is used by the all methods defined in the life cycle and query managers that define operations on the remote registry to encapsulate the response to the calls. It defines a method to access a collection of results in the response. So, once a response is available, a developer can get the results using this method and run through the collection to get the specific objects. Code Fragment 3-7 shows how responses can be retrieved for asynchronous operations using the `getBulkResponse` method.

```
BulkResponse response = lifeCycleManager.saveServiceProfiles(profiles);
String requestId = response.getRequestId();
...
response = registryService.getBulkResponse(requestId);
Collection serviceIds = response.getCollection();
```

Code Fragment 3-7: Retrieving a response for asynchronous operation.

3.7.2 Exception Model

In addition to a response model, the API must provide also an exception model to represent the exceptional cases that might occur when an operation is being performed. The goal is to provide an exception model that is general enough so that it can be extended when new functionality other than service discovery is attached to the API. At the same time, the model should provide constructs that deal with registry operations. The exception model is shown in Diagram 3-20.

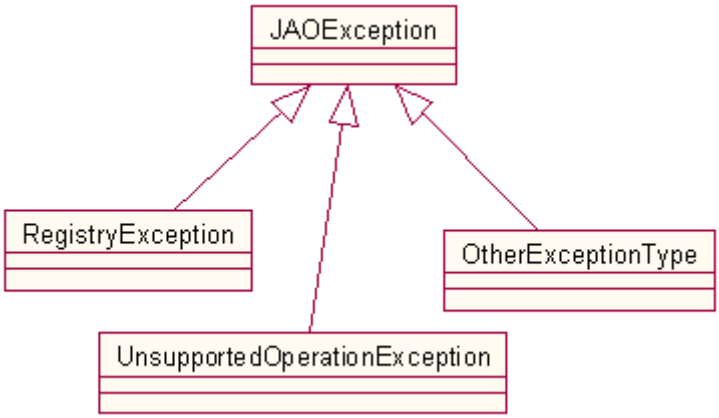

Diagram 3-20: Object model for exceptions.

These classes are defined in the `de.fhg.iao.ws.owl.exceptions` package. The JAOException represents the top class in the model and can be used to represent all types of exceptions occurred while using the API. In addition, a RegistryException class has been created to represent specifically exceptions occurred during registry operations. Finally, a UnsupportedOperationException has been created. This exception should be used in the cases where the API implementation does not implement a determined method.

3.8 Conclusion

This chapter defined the design for a service discovery API for registries based OWL-S ontology. It started with a discussion about the requirements for such a tool and introduced the architecture of the API. Next, an object model for representing the constructs of OWL-S inside a Java program was presented. The chapter went on to define the design constructs of the API regarding connection management, life cycle and query operations, and models for representing responses and exceptions.

As a first step into the design of the API, a set of requirements was presented that included functional and non-functional issues. Here the need for a tool that supported the representation and manipulation of OWL-S design constructs and provided methods to perform operations on a remote registry was discussed along with the necessity of openness and configurability of the framework.

An information model was created that allows for the representation of constructs of the ontology. The model provides a set of interfaces and associated methods that support the representation of the service and profile constructs and can be extended later to incorporate service grounding and model constructs. In addition, new interfaces can be created that extends concepts such as service parameters and service categories.

To handle the requirements regarding to performing operations on the registry, a set of interfaces was designed that deal with different aspects of the process. The interfaces have been divided in groups that deal with connection management, life cycle operations and query methods. This separation allows for a better organization of the

API, by defining interfaces with a small number of methods to handle these different aspects.

In particular, the interfaces that deal with connection management provide the methods that allow for the configurability of the API. In addition, methods that manage connection state and security and authentication issues are defined in these interfaces. On the other hand, the API does not provide constructs to deal with more complex issues related to connection management such as connection pools, leaving up to the developer to implement them.

The life cycle management operations were defined in two different interfaces. The LifeCycleManager deals with save and delete operation that receive general objects as parameters. This arrangement allows for the extensibility of the API as other interfaces created in the future can use these methods. The BusinessLifeCycleManager extends the latter to specify methods concerning operations that deal with specific objects defined in the information model.

Also the query management operations were divided in specific interfaces. On one hand, a declarative manager was defined to handle queries based on declarative languages. On the other side, a business query manager was specified that defined methods for performing query operations on the registry based on the profile objects provided by the user and on specific parameters defined as filters, that can be used to qualify the search.

Also as part of the API specification, design constructs were defined that deal with partial responses and asynchronous calls to the remote registry. This BulkResponse interface is defined as the return type for the methods dealing with operations on the registry and provides support for handling partial failures. The advantage of defining such constructs is that one does not need to modify the model to deal with partial responses and synchronicity.

Finally, the API specification defines an exception model to characterize exceptions raised by the API. In particular all the methods were defined to throw a general JAOException that is the top exception on the model. Other exceptions were defined that deal specifically with operations performed on the remote registry and functions not supported by the API, as the API implementers are not imposed to implement all the methods in the specification.

The model can be extended in the future to provide for new types of exception, as long as they extend from the general JAOException. The decision to provide exceptions to all methods in the API is arguable, as some methods clearly do not qualify for it such as simple get and set methods, and only lead to a burden for the developer that needs to handle these exceptions. On the other side, it provides an alternative for the API implementer in case it is really needed.

It can be concluded that the API specified here defines a tool that is at the same time simple and powerful. It is so because it is able to achieve the requirements of functionality, openness, configurability, and extensibility by defining a set of interfaces and classes that are organized to provide for representation OWL-S constructs,

connection management issues, life cycle and query operations, partial responses and exceptions.

The next chapter presents a reference implementation for the API and discusses the application of API in the context of an example application scenario. More specifically, it presents a critical evaluation of the API by showing how it can be used to develop programs for an example application, discussing the advantages as well as the weak points regarding the tool.

4 Evaluation

This chapter presents an evaluation of the API in the context of an example application scenario. An example scenario is introduced that posts a problem upon which the API can be used as a tool to enhance development. The chapter shows how the API is used to provide solutions to the different problems and discusses the benefits and limitations of the it along the way. The following section introduces a reference implementation of the API that is to be used in the development of the prototype.

4.1 Reference Implementation

This section discusses how a reference implementation for the API has been developed and organized. The discussion includes package organization and the decisions taken on the design and implementation of the interfaces and functions of the API implementation. This reference implementation has been organized in five different packages. The packages and their descriptions are presented on Table 4-1.

Package	Description
de.fhg.iao.ws.owl.impl.infomodel	Includes the classes that implement the interfaces of the information model package.
de.fhg.iao.ws.owl.impl.registry	Includes the classes that implement the interfaces of the registry package.
de.fhg.iao.ws.owl.impl.registry.rpi	Includes a general interface that defines operations to be performed on the registry and that must be implemented for each type of registry the implementation wants to support.
de.fhg.iao.ws.owl.impl.registry.rpi.jaxr	Includes the classes that implements general interface of the package above and other supporting classes. The classes in this package are the ones that actually perform operations on the registry.
de.fhg.iao.ws.owl.impl.xml	Includes the classes that are responsible for XML processing and implement reading operations on the file system.

Table 4-1: Reference implementation package organization.

4.1.1 Information Model

The package `de.fhg.iao.ws.owl.impl.infomodel` described in the table includes the classes that implement the interfaces that define the information model of the API. A main class in this package is the ElementImpl, that implements the Element interface and is extended by the classes defined in the package that represent OWL-S constructs.

The ObjectFactoryImpl is a class that extends the abstract class ObjectFactory defined in the package `de.fhg.iao.ws.owl.infomodel` of the API. Among others, it is responsible for implementing the factory methods that create instances of the objects defined in the information model. And among these factory methods, particularly interesting is the `createObject` method.

This method returns an object according to the interface name provided to it as a parameter. It is specially important to provide support for new interfaces, allowing for the extensibility of the API. As an example, if one needs to create a new service category or parameter, he can create an interface and implementation class for it, and then modify the create object on the implementation to support the new interface.

4.1.2 Registry Operations

The package `de.fhg.iao.ws.owl.impl.registry` in the table includes the classes the implement the interfaces defined on the `de.fhg.iao.ws.owl.registry` package, and so the whole functionality regarding connection management, and life cycle and query operations upon the registry. At this point, is must be mentioned that for this reference implementation, not all the functionality defined in the API interfaces has been actually developed.

One reason for this comes from the fact that the UDDI registry implementation upon which the reference implementation has been developed does not offer some of the functionality defined in the API. To be more specific, this registry implementation does not provide support for declarative queries and asynchronous communication, and for implementing these functions means that the API would have to be programmed to emulate these operations on the top of a registry that does not provide this functionality.

Also functions defined for performing operations regarding service grounding and service models have not been developed, as interfaces that represent these OWL-S constructs is yet to be defined as part of a broader API. For all such functions that an implementation has not been developed, the reference implementation has been programmed to throw an specific exception to indicate that such function is not supported by this API implementation.

In fact, the API has been designed in way that it is open enough for the implementer to decide what functions its implementation should support. The designers of the JAXR API have used a different approach, though, by defining a set of capability levels that indicate which functions the API implementer must provide. This guarantees that an implementation that supports a certain level will provide the functionality specified for that level.

Defining capability levels has advantages, as it gives a guarantee to the developer that if an API implementation supports a determined capability level, then changing to another implementation that supports the same level will not require changes on the program. The problem with this approach is that as registries based on semantic web technologies are still much under research, it is not clear at the moment what are the functions they should support.

In fact, there is no clear standard on developing such registries. Some might be very simple and support basic operations, while other can be more complete. For this reason a decision was taken of not defining these levels as part of the API design. But in this case the API implementers must make it clear in their documentation what type of functions they support. Perhaps later the API design can be modified to include capability level constructs.

The find operations defined in the business query manager are a particular case. These operations specify filters as part of their interface, but the reference implementation does not implement filters on its code, as it is based on an UDDI registry and does not support semantic processing. No exception is thrown for these operations, though. The developer here should simply pass a null as parameter to the method, as the information provided will be ignored.

But with exception of connection management and factory methods functionality, the API reference implementation does not implement all the methods on its manager implementation classes. In effect, the functionality regarding query, save, and delete operations on the registry, and read and write file operations are delegated to another set of objects organized in two other packages that are described in the next sections.

4.1.3 Registry Provider Interfaces

The operations regarding registry operations are delegated to an interface defined in the registry provider interface package `de.fhg.iao.ws.owl.impl.registry.rpi` described on the table above. This interface can be implemented by classes that, in the case of this reference implementation, have been defined in the sub package `de.fhg.iao.ws.owl.impl.registry.rpi.jaxr`. This section briefly presents the registry provider interface package.

The `de.fhg.iao.ws.owl.impl.registry.rpi` package has been designed to contain the interfaces that define operations to be performed upon the remote registry. For this reference implementation a single interface has been defined that specifies all the operations that can be performed on a registry, including all the save, delete and find operations provided in the API. Diagram 4-1 shows this arrangement.

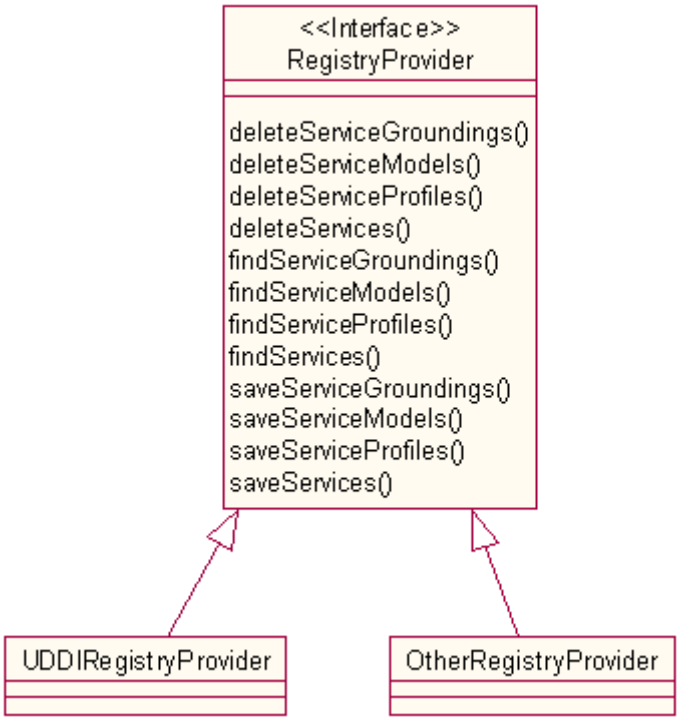

Diagram 4-1: Object model for registry provider interface.

The main idea here is to provide the developer with a way to configure the registry implementation they want to use. In fact, a possibility would be to define this interface as part of the API specification, forcing API implementers to provide implementations not only to the interfaces that are used directly by the developer, but also for this inner interface. This decision would decouple the implementation of API basic functions from registry operations.

Moreover, it would give API implementers a standard interface from which implementations could be developed for the different registries to be supported by the API. This would work more or less like database drivers for which standard interfaces exist that can be implement to provide support to a determined database. The application developer would just need to plug the specific driver to provide support for a new registry.

The downside of this approach is that it restricts the freedom of the API implementer to provide different solutions to the API specification. In this work a decision was taken not to define this interface as part of the specification, at least for this first version. In the future, if a common way of implementing the specification becomes a standard or the need to provide drivers for registries becomes real, this interface can be defined as part of the specification.

4.1.4 Registry Provider Implementation

For the reference implementation defined in this work, the classes that implement the registry operations specified in the registry provider interface are organized into a sub package `de.fhg.iao.ws.owl.impl.registry.rpi.jaxr`. The package has been

named JAXR because the reference implementation has been developed on the top of the JAXR API to perform operations on an UDDI registry. A set of classes is defined in this package that are briefly described below. Diagram 4-2 shows the model for the registry provider implementation.

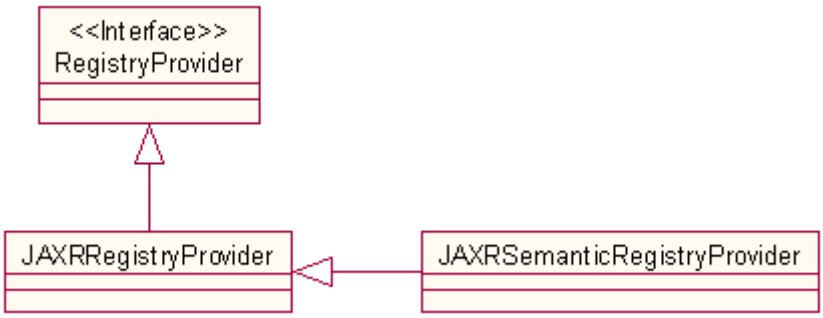

Diagram 4-2: Object model for registry provider implementation.

The most important classes in this package are the JAXRRegistryProvider and the JAXRSemanticRegistryProvider that implement the interface defined in the super package and are responsible for performing operations on the registry. The difference between the two classes is that the JAXRSemanticRegistryProvider implements some semantic functionality that originally should be implemented on the registry, but not possible since a UDDI registry is being used.

The JAXRRegistryProvider implements the methods that perform life cycle operations. In the case of saving a service, for example, the client calls a method defined in the API that is captured by the BusinessLifeCycleManagerImpl class in the API implementation. This class delegates the execution of the method to the JAXRRegistryProvider through the interface RegistryProvider. The JAXRRegistryProvider is then responsible for performing the operation on the registry and handling the results. The Diagram 4-3 shows this arrangement.

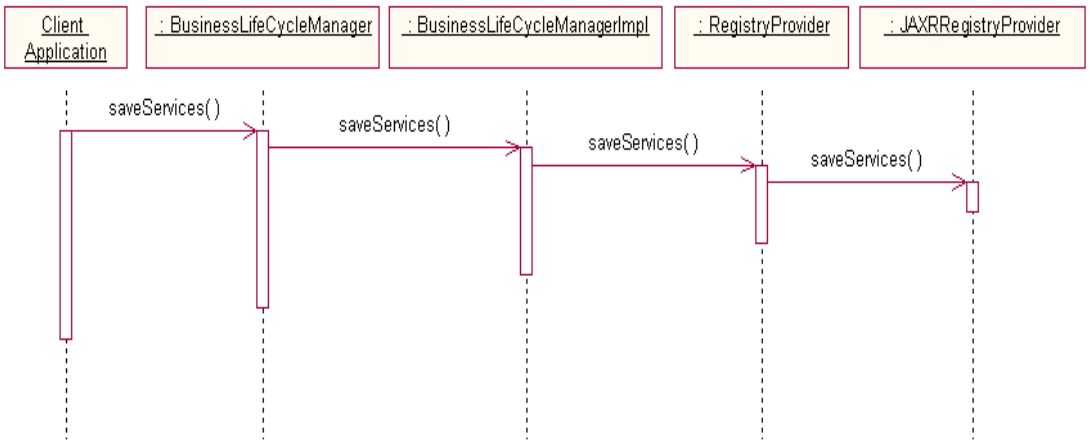

Diagram 4-3: Sequence diagram for save services operation.

The JAXRSemanticRegistryProvider, on the other side, is used for query operations. In fact, it overrides some of the methods regarding find operations defined in the

JAXRRegistryProvider. The Diagram 4-4 shows how operations are executed. In the case of a find services operation, for example, the client calls a method on the API that is captured by the BusinessQueryManagerImpl. This query manager then delegates the execution to the JAXRSemanticRegistryProvider through the RegistryProvider interface, and this class is responsible for performing the operation on the registry.

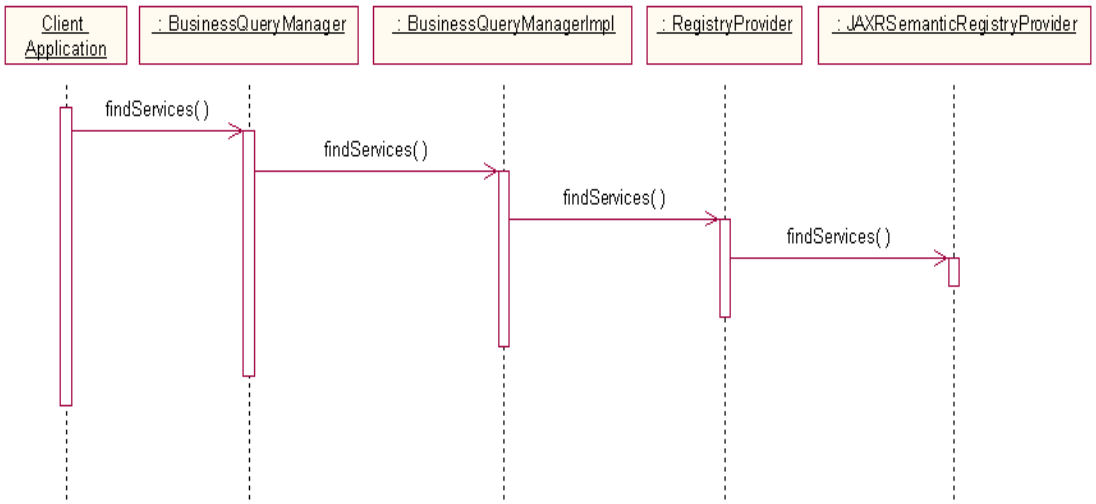

Diagram 4-4: Sequence diagram for find services operation.

The JAXRSemanticRegistryProvider class defines methods that emulate reasoning operations regarding geographic radius descriptions that are specific to the prototype application discussed in the next chapter. But it also serves as example of how different implementations can be plugged into the API. The developer uses the properties of the connection factory to define which implementation class should be used.

Other two important classes are implemented in this package. These are the classes that provide the mapping between OWL-S and JAXR concepts. The OWLSJAXRMapper is used for mapping OWL-S constructs into the JAXR information model for save, delete and find operations. On the other side, the JAXROWLSMapper is used to map JAXR constructs into the OWL-S information model during query operations.

4.1.5 File Processing

The `de.fhg.iao.ws.owl.impl.xml` package contains the classes that are responsible for OWL-S file processing. These classes were put together into this package exactly to isolate XML processing operations from registry specific operations. The FileProcessor implementation class receives the call to the read and write methods from the client program and delegates the execution to classes defined in this package. Diagram 4-5 shows this arrangement.

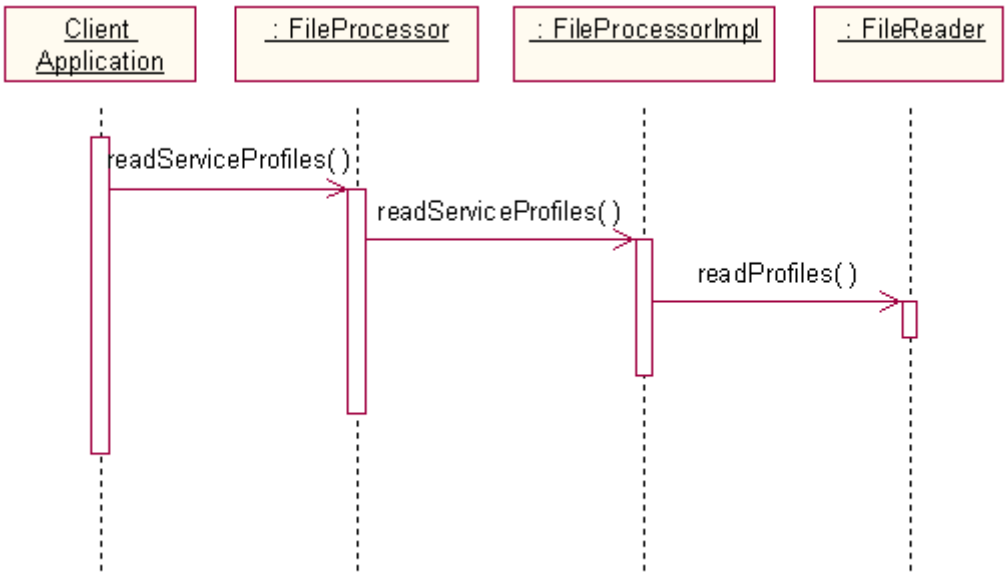

Diagram 4-5: Sequence diagram for read profiles operations.

In the case of read operations, for example, the execution is delegated to a FileReader class, that implements the methods responsible for reading the services and profiles defined in OWL files in the file system. The class uses a SAX parser and the Java API for XML processing, that requires the implementation of handler classes for parsing through the files. For this reason, a handler has been implemented for service and other for profile files.

For this reference implementation an option has been made to use SAX parsing and the default parser provided by the Java API for XML Processing. In this implementation, no option of configuration is possible, although another version can be developed later that allows for the configuration of such aspects. In this case, the developer could use the properties of the FileProcessor to define whether it is going to use a SAX or DOM parser for reading the documents, for example.

The reference implementation still does not provide classes for processing write operations. For these operations, a class or set of classes is to be constructed that is able to process the information contained on the information model objects, more specifically the objects that represent service and profile, and write this data onto files in the file system. These classes could use a DOM API to perform these operations.

4.2 Application Scenario

The transportation services industry has been used in the past chapters as example to introduce the problem definition and discussing possible technological solutions. This same application is going to be used here to present an example application scenario upon which the API can be applied. In the next few paragraphs a more precise definition of the scenario is presented describing in detail the components of the application.

The first components of the application scenario that need to be described are the service providers. Service providers here are companies that offer transportation services to their clients. These companies have information systems that process their data and help reduce costs. But some of these systems that are operated manually by human resources in the company could be operated directly by client applications, reducing even more the operational costs.

The clients are companies or individuals that wish to access the applications provided by the service providers. These clients have applications that will connect to the service providers and place remote calls to the services to perform the operations they need. A good example of a client here would be online stores that receive an order into their software systems and must contact a transportation service to ship the product to their customers.

In this example two different applications are to be exposed. The first one is used to get information about the services being offered by the provider, as shown in Figure 4-1. In this case, the client application must provide information about the origin, destination, time of departure and arrival, and the server must return the service descriptions that match these characteristics, including information regarding such as price of the service, for example.

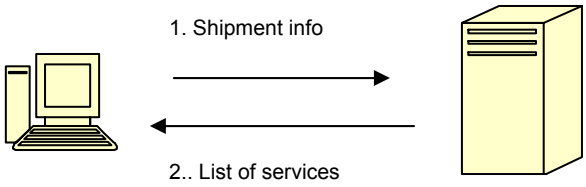

Figure 4-1: Get services information operation.

The second application to be exposed is the transportation ordering service. This application receives a service identification number from the client system and places an order of transportation into the ordering systems of the service provider. Once the ordering application is done, it returns an order number to the client. This arrangement is shown in Figure 4-2. This application is combined with the first one to provide a single service to the client.

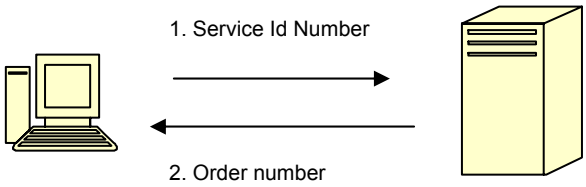

Figure 4-2: Shipment order operation.

As discussed in the past chapters, a possible solution to connect these applications is to use web services combined with semantic web technologies. This prototype uses OWL-S documents to provide a description of the services offered by providers. The

descriptions are going to be stored in a remote registry specific for transportation services that client and service providers can access.

The application developers on the client side are going to use the API to develop applications that are able to perform query operations on the registry using OWL-S descriptions. The developers on the service provider side will use the API to perform insert, update and delete operations regarding their services descriptions. The next section describes how the OWL-S descriptions of a service provider are organized.

4.3 Service Descriptions

As it was already discussed in the past chapters, the OWL-S ontology provides a number of different ways to describe a service. More specifically, service profiles, grounding and model classes can be used to describe the different aspects of a service, such as how to connect to a service and how to execute it. Particularly important for service discovery though, is the Profile ontology class and this prototype focuses on this type of description.

The OWL-S profile description provides a set of elements that can be used to describe a particular web service. This section presents how the services in the example scenario are being described showing which of the OWL-S constructs are being used. In particular, the geographic radius construct requires special attention. An ontology is defined for representing instances of this particular class and is explained more ahead..

4.3.1 Service Description

Before specifying the profile description, the elements in the service description file need to be described. Code Fragment 4-1 shows how the shipment order web service for a company A can be described. The code basically uses the `rdf:ID` attribute in the `service:Service` tag to provide an ID that describes this service uniquely in the registry and encloses the other elements that define the service. The complete code is presented in Appendix A.1.

```
<service:Service rdf:ID="CompanyA_InorderAgent">

   <!-- Reference to the CompanyA Profile -->
   <service:presents rdf:resource="http://www.iao.fhg.de/services/owl-
s/0.9/CompanyAProfile.owl#Profile_CompanyA_InorderAgent"/>

   <!-- Reference to the CompanyA Process Model -->
   <service:describedBy rdf:resource="http://www.iao.fhg.de/services/owl-
s/0.9/CompanyAProcess.owl#CompanyA_InorderAgent_ProcessModel"/>

   <!-- Reference to the CompanyA Grounding -->
   <service:supports rdf:resource="http://www.iao.fhg.de/services/owl-
s/0.9/CompanyAGrounding.owl#Grounding_CompanyA_InorderAgent"/>

</service:Service>
```

Code Fragment 4-1: Service description for company A.

The other element important to notice is the `service:presents` tag, that defines a pointer to the profile associated with this service through its `rdf:resource` attribute. The other elements define the process model and a service grounding for this service, but these are less important for the service discovery process and will not be further described here. It must be noticed that a service might have other profiles, but only one profile tag is defined for this prototype.

4.3.2 Profile Description

Once the service construct has been described, the profile description must be introduced. Code Fragment 4-2 shows a part of the profile document for the service defined above. The complete code is presented in Appendix A.2. Here the profile `profile:Profile` tag defines the unique ID for this profile through the `rdf:ID` and encloses the other elements of the profile that define the characteristics of the service being defined.

```
<profile:Profile rdf:ID="Profile_CompanyA_InorderAgent">

  <!-- reference to the service specification -->
  <service:presentedBy rdf:resource="http://www.daml.org/services/owl-
s/0.9/Service.owl#CompanyA_InorderAgent"/>

  <!-- reference to the process model specification -->
  <profile:has process rdf:resource="http://www.iao.fhg.de/services/owl-
s/0.9/CompanyAProcess.owl#CompanyA_Process"/>

  <profile:serviceName>CompanyA_InorderAgent</profile:serviceName>

  <profile:textDescription>Inorder agent service</profile:textDescription>

  ...

</profile:Profile>
```

Code Fragment 4-2: Profile description for company A service.

The `service:presentedBy` tag is particularly important as it defines a link to the service document described previously in this section and ties the profile description to a particular service. Also the `profile:has_process` tag is important as it links the profile to the process model OWL-S document that describes how the application should be run. The other tags provide simply a service name and text description to the service.

4.3.3 Contact Information

The Code Fragment 4-3 illustrates how contact information is defined in the profile descriptions. It is important to notice how the `profile:contactInformation` tag encloses the `profile:Actor` construct, which is the element that actually defines the contact information. Inside the `profile:Actor` tag, other constructs defined as properties of the Actor class are used to describe name, title, phone and other important contact data.

```
<profile:Profile rdf:ID="Profile_CompanyA_InorderAgent">

...
```

```
  <profile:contactInformation>
    <profile:Actor rdf:ID="CompanyA-inorder">
      <profile:name>CompanyA Inorder department</profile:name>
      <profile:title>Inorder Representative</profile:title>
      <profile:phone>412 268 8780 </profile:phone>
      <profile:fax>412 268 5569 </profile:fax>
      <profile:email>Inorder@CompanyA.com</profile:email>
      <profile:physicalAddress>CompanyAstrasse 20</profile:physicalAddress>
      <profile:webURL>http://www.companyA.com/</profile:webURL>
    </profile:Actor>
  </profile:contactInformation>

...

</profile:Profile>
```

Code Fragment 4-3: Contact information for company A.

4.3.4 Parameter Descriptions

Parameter descriptions are used to define the inputs, outputs, preconditions and effects regarding the execution of the service. Code Fragment 4-4 shows how these constructs can be specified for inputs and outputs. The complete code can be found on Appendix A.2.

```
<profile:Profile rdf:ID="Profile_CompanyA_InorderAgent">
...
  <!-- Descriptions of IOPEs -->
  <profile:input>
    <profile:ParameterDescription rdf:ID="PickupLocation">
      <profile:parameterName>PickupLocation</profile:parameterName>
      <profile:restrictedTo rdf:resource="http://www.iao.fhg.de/services/owl-
s/0.9/Concepts.owl#Location" />
      <profile:refersTo rdf:resource="http://www.iao.fhg.de/services/owl-
s/0.9/CompanyAProcess.owl#pickupLocation_In" />
    </profile:ParameterDescription>
  </profile:input>
...
  <profile:output>
    <profile:ParameterDescription rdf:ID="AvailableServices">
      <profile:parameterName>AvailableServices</profile:parameterName>
      <profile:restrictedTo rdf:resource="http://www.iao.fhg.de/services/owl-
s/0.9/Concepts.owl#AvailableServicesList" />
      <profile:refersTo rdf:resource="http://www.iao.fhg.de/services/owl-
s/0.9/CompanyAProcess.owl#availableServicesList_Out" />
    </profile:ParameterDescription>
  </profile:output>
...
</profile:Profile>
```

Code Fragment 4-4: Parameter descriptions for company A service.

As it can be seen on the Code Fragment 4-4, both inputs and outputs contain a `profile:ParameterDescription` element that describes the input or output parameter being defined. Key elements in this definition are the tags `profile:parameterName` and `profile:restrictedTo` that indicate the name and type of the parameter, respectively. The tag `profile:refersTo` defines a link to the process model this parameter is attached to.

It is important to understand how these parameter descriptions are organized in the Appendix A.2. In this description, the first four input tags and the following output tag define the inputs and outputs to the remote application that provides information about transportation services. In this case, the inputs are the drop off and pick up locations, and the departure and arrival dates, while the output is a list of available services containing more detailed information about them.

The following three input tags, and the output and effect tags are used to describe the parameters of the order application. In this case, the inputs for the service are the login, password and a service identification number, and the output is a order number, while the HaveInorder effect determines the post condition after execution. These applications must be executed in sequence and this fact has to be described using a service model.

Unlike the contact information, the tags regarding inputs, outputs, preconditions, and effects are important because part of the matching process is based on this information. That means that when a client submits a profile to the registry, the registry will look for services registered in its database that match somehow these descriptions and return the results back to the client that will be able to choose the service it wants.

4.3.5 Service Category

Besides inputs and outputs information, the matching process also return results based on service categories and service parameters. In effect, service categories could be specified as a first filter in the matching process, as it defines, based on some classification system, to which category a service belongs. This is particularly useful when a registry contains services of different application domains, such as transportation and hotel industries for example.

In these cases, it can happen that services defined in such different domains have the same set of inputs and outputs, and so they would be returned as part of the result. Using service categories as a first filter will help the registry return better matches to the service the client is looking for. The Code Fragment 4-5 shows the definition of a service category inside a profile document using the North American Industry Classification System, NAICS.

```
<profile:Profile rdf:ID="Profile_CompanyA_InorderAgent">

...

  <!-- Specification of the service category using NAICS -->
  <profile:serviceCategory>
    <profile:NAICS rdf:ID="NAICS-category">
      <profile:value>General Freight Trucking</profile:value>
      <profile:code>4841</profile:code>
    </profile:NAICS>
  </profile:serviceCategory>

...

</profile:Profile>
```

Code Fragment 4-5: Service category for company A service.

Here the tag `profile:serviceCategory` identifies the service category construct, while the `profile:NAICS` determines that the NAICS class is being used to represent the category. Important in a NAICS classification is the code, that identifies a specific category uniquely and is defined here in the `profile:code` tag. A profile description can define different service categories that are based on different classification systems.

4.3.6 Service Parameter

Another important construct are the service parameters. The service parameters can be used to define any sort of additional information that can be used to qualify the service, such as quality rating and geographic radius. They could be used during the matching process as a last filter, so that once service categories and inputs and outputs have been matched, the parameter would be applied to improve the results.

The prototype defined here is going to focus on a geographic radius service parameter defined as part of the profile specification. The Code Fragment 4-6 describes how these constructs are defined in the profile document. The `profile:serviceParameter` tag encloses the parameter being defined, while the `profile:GeographicRadius` defines the parameter information, more specifically the parameter ID, name and value.

```
<profile:Profile rdf:ID="Profile_CompanyA_InorderAgent">

...

  <!-- description of Geographic radius as a service parameter -->
  <profile:serviceParameter>
    <profile:GeographicRadius rdf:ID="CompanyA-geographicRadius">
      <profile:serviceParameterName>CompanyA Geographic
Radius</profile:serviceParameterName>
       <profile:sParameter rdf:resource=" http://www.iao.fhg.de/services/owl-s/0.9/
Country.owl#Germany"/>
      </profile:GeographicRadius>
    </profile:serviceParameter>

...

</profile:Profile>
```

Code Fragment 4-6: Geographic radius service parameter for company A service.

The key element in this definition is the `rdf:resource` in the `profile:sParameter` tag. This resource determines the value of the service parameter, that is defined here as a class belonging to an ontology specified in the file Country.owl. Here a remark must be made that by the time this work is being written, the original Country.owl file defined in the OWL-S profile specification does not specify a complete ontology.

4.3.7 Geographic Radius Ontology

The geographic radius service parameter defined in the OWL-S profile specification only accepts as values instances of a class Country defined in the Country.owl file, or subclasses of it. An alternative then is to create a new service parameter, different from geographic radius, that works with a particular ontology. Another option is to define a new ontology class which the extends from the Country class defined above.

For the prototype defined in this work, a choice was made for the latter. The classes that compose the ontology and their descriptions are described on Table 4-2. For the particular example shown at Code Fragment 4-6, the geographic radius construct being defined specifies that this particular service description refers to transportation services that operate solely inside the country Germany.

Class	Description
France	Describes services operated only inside France.
Germany	Describes services operated only inside Germany.
Italy	Describes services operated only inside Italy.
France_and_Germany	Describes services operated inside France and Germany, and between these countries.
France_and_Italy	Describes services operated inside France and Italy, and between these countries.
Germany_and_Italy	Describes services operated inside Germany and Italy, and between these countries.
Germany_and_France_and_Italy	Describes services operated inside Germany, France and Italy and among these countries.

Table 4-2: Classes of the ontology for geographic radius.

A final remark must be made that as the API reference implementation has been developed based on an UDDI registry and is so enable to perform reasoning operations, an OWL document describing the ontology is actually not needed. That would be necessary only for implementations based on semantic web technologies, in which case such a document would be needed to code and process the reasoning operations.

4.3.8 Other Service Descriptions

The code fragments described in the last sections presented the description of a transportation web service for an example company A. But for the example application scenario to be complete, one must define some other services whose descriptions are going to be stored in the registry and that will compete with the services offered by company A. To make the example simple, a decision was taken to provide descriptions that are similar to the company A description.

So, the service descriptions being defined are going to contain the same constructs specified for company A, namely, service name and text description, contact information, inputs, outputs and effects, NAICS classification and geographic radius. Among these, a special focus is given to the NAICS and geographic radius descriptions, that are going to be used to provide a differentiation among the services.

In fact, services are going to be defined with similar values for the inputs, outputs, and effects, but different values for NAICS classification and geographic radius that enable one to differentiate the services. The Table 4-3 shows a list of the companies with services stored in the registry for the example application, with a special focus on the values for NAICS and geographic radius for these services.

Company	NAICS Classification	Geographic Radius
Company A	General Freight Trucking (4841)	Germany
Company B	General Freight Trucking (4841)	France
Company C	General Freight Trucking (4841)	Italy
Company D	General Freight Trucking (4841)	France and Germany
Company E	General Freight Trucking (4841)	France and Italy
Company F	General Freight Trucking (4841)	Germany and France and Italy
Company G	Specialized Freight Trucking (4842)	Germany
Company H	Specialized Freight Trucking (4842)	France
Company I	Specialized Freight Trucking (4842)	Italy
Company J	Specialized Freight Trucking (4842)	France and Germany
Company K	Specialized Freight Trucking (4842)	France and Italy
Company L	Specialized Freight Trucking (4842)	Germany and France and Italy

Table 4-3: Example companies defined for the prototype.

The table specifies that while the companies A and B offer the same type of transportation service, that is general freight trucking, NAICS code 4841, they actually operate on different regions, which is determined by the geographic radius value, Germany for company A, and France for company B. The company D operating on France and Germany would compete with these companies though, as it offers the same type of service.

On the other side, a company H with geographic radius France would not compete with company B, for example, as it offers a different kind of service, namely, specialized freight trucking. So, the registry on the example application will contain the service descriptions defined above that somehow intersect with each other, and a client application searching for a service must query the registry using a profile definition to find possible services it can use.

This section basically presented the way service descriptions are coded for the example application scenario. The next sections will go on to describe how the API defined in this work can be used to help develop programs that perform operations on the registry and discuss the advantages and limitations. In particular, the following section discusses how the API can be used to develop applications regarding service discovery.

4.4 Query Operations

The past section has defined the service descriptions being stored in a remote registry. This section describes a set of three different client applications that use the registry to search for these transportation services. The applications use the service discovery API to perform query operations on the remote registry that will enable them to find the services they need and the information necessary to execute the service later on.

The first client application to be described is defined here as client A. The client A represents an application that automates the process of subcontracting transportation

services to transport the products from the company to its customers. To be more specific, client A is a company that operates solely inside Germany and needs to deliver its different products using either a general or a specialized trucking service, depending on the product.

It must be mentioned that in a real company the process of contracting would probably be integrated on the enterprise application. For example, the contracting application would be called automatically by some application in the manufactory line every time a certain number of products have been finished. Or it could be called by some sales application that receives and processes sales orders, and is also responsible for shipping the products.

To make the prototype simple, a general user interface has been designed that enables one to interact with the client applications presented in this work. The interface is shown in the Figure 4-3. It is a quite simple interface that allows one to choose transportation service type, origin and destination, and submit the data to an application lying on a server that will process this information and return the results on a web page.

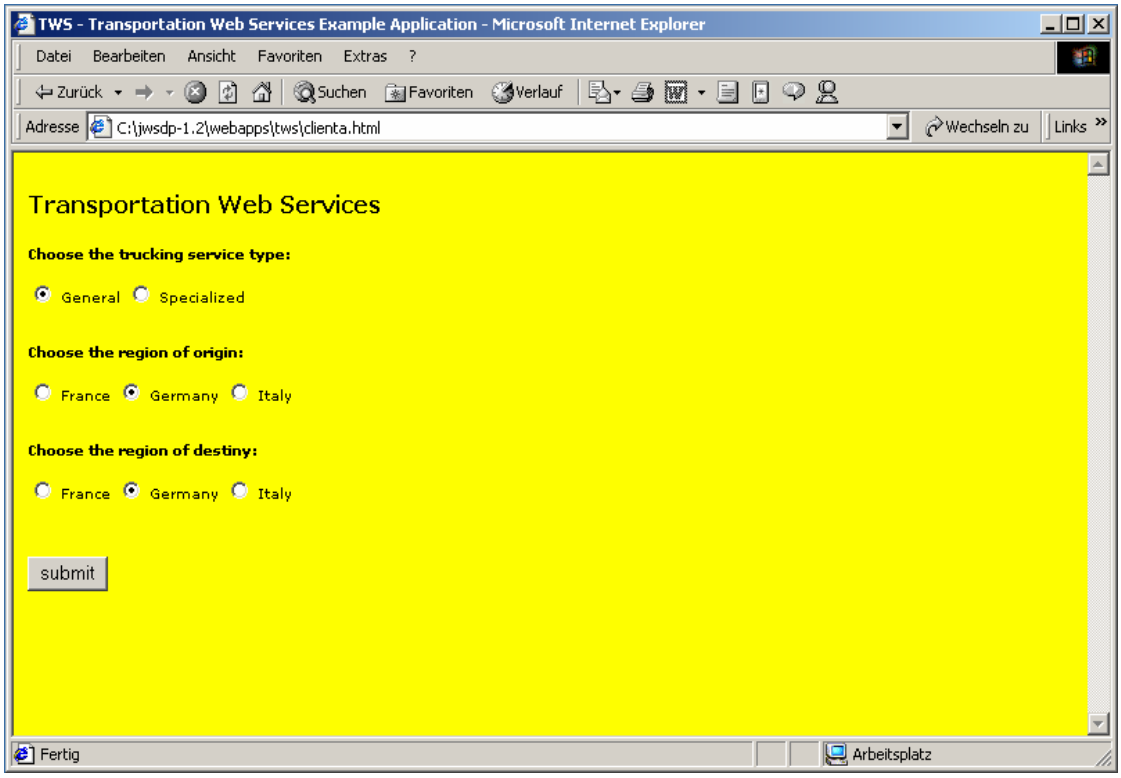

Figure 4-3: Graphical user interface for the prototype.

4.4.1 General Architecture of the Prototype

The next few paragraphs describe then how a client A application can be coded using the API that connects to the registry and perform query operations on it to find out which services exist that can be used by the company. Along the way, the different

possibilities of development are discussed and the advantages and limitations of using the API are analyzed. Figure 4-4 describes the general architecture of the application.

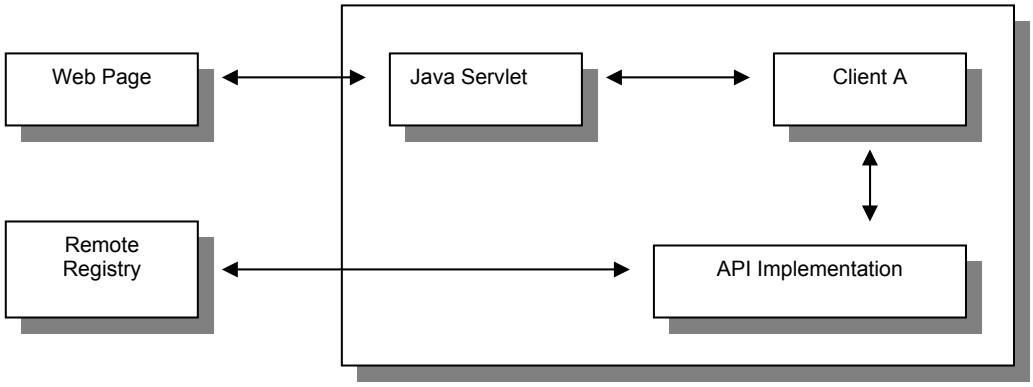

Figure 4-4: Organization of the prototype application.

In this figure one can observe how the prototype is structured. First, a user utilizes a web page to submit the data to the web server. On the server, a Java Servlet application receives the request and calls a method in the client application passing the data gathered in the user interface as parameter. The application will process this data and use the API to perform queries on the registry and return the result back to the Servlet, and so to the user.

4.4.2 Connection Setup

The first step in coding the client application is to develop the code that connects the client to the remote registry. The Code Fragment 4-7 shows how this can be done using the API. In the prototype developed for this work, this code has been placed in a separate method that is executed every time a query operation must be performed. This means that a new connection is created for every query in this case and closed after all.

```
ConnectionFactory connectionFactory = ConnectionFactory.newInstance();
Properties properties = new Properties();
properties.setProperty("javax.xml.registry.queryManagerURL",
        "http://localhost:8080/RegistryServer");
properties.setProperty("de.fhg.iao.ws.owl.RegistryProviderClass",
        "de.fhg.iao.ws.owl.impl.registry.rpi.jaxr.JAXRRegistryProvider");
connectionFactory.setProperties(properties);
Connection connection = connectionFactory.createConnection();
```

Code Fragment 4-7: Connection setup and configuration.

But one could easily implement a more sophisticated mechanism that perhaps creates the connection once and let it open for a determined time, so that the connection does not need to be created every time a query operation is performed. After a time without being used the connection would be closed. Such mechanism and others such as connection pools are not built in the API design constructs, but can be implemented by the developer to handle more specific situations.

The API provides for the configuration of the connection through the properties attribute of the connection factory. In the code fragment above, the method `setProperties` is used to define the address of the remote registry and the registry provider implementation to be used during this connection. This information is hard coded here, but it could be read from a configuration file into Strings that would be used to set the properties.

The program is able to recognize these properties because the reference implementation of the API was programmed to work with them. The API specification does not determine which properties the developer should define. It is up to the API implementer to specify which properties it supports and inform the developer about them. This is actually an advantage of the API, so that in the case of changing implementations one would need to change only the configuration file.

4.4.3 Authentication Information

The next step is to set up authentication information for the connection. Code Fragment 4-8 shows how this is done using the `setCredentials` method of the connection object. This authentication data might not be necessary in the case of query operations, but this reference implementation requires to do so. The method receives a set of `PasswordAuthentication` objects as a parameter which encapsulates a username and the correspondent password.

```
String username = "testuser";
String password = "testuser";
PasswordAuthentication authentication =
       new PasswordAuthentication(username,password.toCharArray());
Set credentials = new HashSet();
credentials.add(authentication);
connection.setCredentials(credentials);
```

Code Fragment 4-8: Authentication credentials setup.

This is so because the API reference implementation is based on JAXR and UDDI registries that require the developer to use this specific type of object. But the API specification does not determine a particular type of object to be used. Here once more, it is up to the implementer to specify the authentication mechanism being used or whether the API implementation supports any mechanism at all.

As long as the API implementations provide support for the specific types of authentication objects, the developer does not need to change the program, and can switch between different implementations whenever it wants. Now that the code for setting up the connection has been written, the next step is to code the query operation itself. In the API, a query is made by calling one of the find methods defined in the query manager passing a profile object as a parameter.

4.4.4 Reading Profile Descriptions

The profile description to be used in the query operation should be similar to the profiles of the transportation companies already stored in the registry, so that a match is possible. In the case of client A, the profile description must provide the same types

constructs described in the last section for the companies, changing only the values for geographic radius and NAICS classification to correspond to the types of services it wants to find.

Here a whole new profile could be coded using the factory methods provided by the API. A better solution though, is to use the `readServiceProfiles` method provided in the API that allows one to read a profile OWL file from the file system returning a collection of profile objects to the program. The Code Fragment 4-9 shows exactly how this is done.

```
FileProcessor fileProcessor = FileProcessor.newInstance();
Collection profiles = fileProcessor.readServiceProfiles(profileFilename);
```

Code Fragment 4-9: Reading profile from the file system.

The first step is to create a new instance of a FileProcessor object that implements the `readServiceProfiles` method. This method returns a collection of profiles defined in a file given file name. The details of how the method is actually implemented are defined by the API implementer. For this reference implementation, the method simply parses through the file, but no validation is actually made. If there is any error in the file, the method will only consider the valid constructs.

The implementation does not provide any parameter to support the configuration of parsers or any other aspect related to XML processing. If the API implementer wants to provide such capability it can use the properties of the FileProcessor to allow the developer to set up these properties and change them whenever is needed.

4.4.5 Modifying Service Categories

The execution of the `readServiceProfiles` method is going to bring to the memory a collection containing a single profile that corresponds to the one on the file systems. This profile needs to be modified, though, as the NAICS category is determined by the value the user submitted through the web page. To do this using the API the developer simply needs modify the NAICS service category associated to the profile object. The Code Fragment 4-10 shows how this is done.

```
Iterator iteratorProfiles = profiles.iterator();
while (iteratorProfiles.hasNext()) {
        profile = (Profile) iteratorProfiles.next();
        Collection serviceCategories = profile.getServiceCategories();
        if ((serviceCategories != null) && (serviceCategories.size() > 0)) {
                Iterator iteratorServiceCategories = serviceCategories.iterator();
                while (iteratorServiceCategories.hasNext()) {
                        ServiceCategory serviceCategory =
                                (ServiceCategory) iteratorServiceCategories.next();
                        if (serviceCategory instanceof NAICS) {
                                if (type.equals("general")) {
                                        serviceCategory.setCode("4841");
                                } else if (type.equals("specialized")) {
                                        serviceCategory.setCode("4842");
                                }
                        }
                }
        }
}
```

Code Fragment 4-10: Modifying service categories information.

The first step here is to read through the collection of profiles to get the single profile stored there. Then the developer uses the `getServiceCategories` method to get the collection of service categories that describe the service, which in this case is a single NAICS category. The code must verify whether the service category is an instance of the NAICS interface, and then set the code according to the type provided by the user as a parameter.

At this point it becomes clear how the semantic web technologies can help integrate applications. In this case, an application that works with the concepts of general and specialized transportation will be able to find a service that executes the operations it wants because these concepts are being mapped into other concepts that are unique and understood by all the applications.

4.4.6 Performing Queries

Once the profile has been changed, the program can perform calls to the remote registry based on this profile. Code Fragment 4-11 shows how this is done. The first step is to acquire a BusinessQueryManager from the registry service object. This manager contains a set of methods that can be used to perform queries on the registry. For this prototype, the `findServiceProfiles` method has been chosen. In this specific case, no filters are used as the API implementation does not support filters and a null value is set the second parameter of the method.

```
BusinessQueryManager queryManager = registryService.getBusinessQueryManager();
BulkResponse response = queryManager.findServiceProfiles(profile, null);
Collection profilesResponse = response.getCollection();
```

Code Fragment 4-11: Find service profiles operation.

This method returns a bulk response that contains a collection of profile objects that can be accessed through the `getCollection` method. To present the results to the user, some of the information is retrieved from these profile objects and returned in a web page. The page returned to the user presents the service description URI, some contact information, the geographic radius and the NAICS code.

Figure 4-5 shows the result of executing the program by submitting a request passing the general transportation type as parameter. The information about origin and destination is ignored by the client application. The result in effect shows another advantage of using semantic web technologies, that is possibility of performing reasoning operations based on rules defined in some ontology.

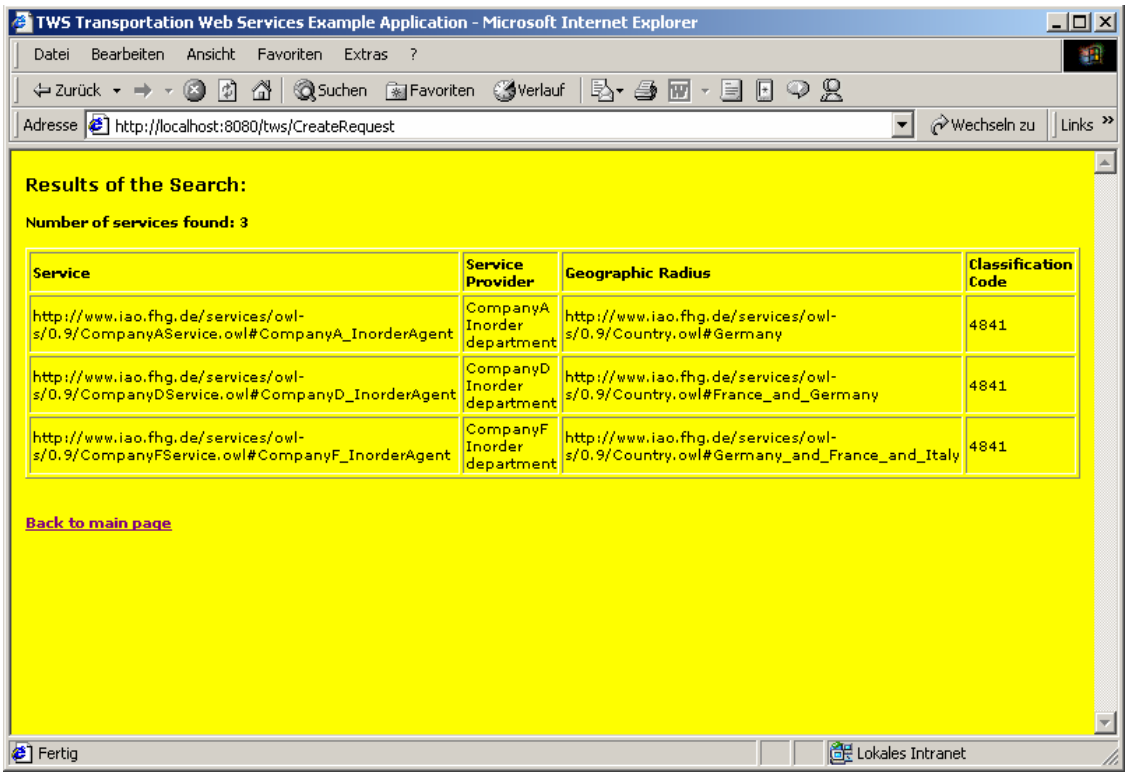

Figure 4-5: Result of search for general transportation in Germany.

In this particular case, the query for a service defined with geographic radius values corresponding to Germany, returns services with geographic radius equals to Germany, Germany and France, and Germany and France and Italy. This is so because the ontology defined for geographic radius specifies, among other rules, that if a company operates its services in the radius Germany and France, or Germany and France and Italy, then it also operates in radius Germany.

The API implementation should not play any role on the execution of reasoning operations, though. It is up to the registry software to implement the rules and perform the reasoning operations. What the API provides is a set of interfaces and associated methods that can be used inside the program to represent these concepts and perform the calls to the registry.

4.4.7 Modifying Profile Descriptions

To discuss update operations, a client B application must be introduced first. The client B is defined here as a company that manufactures its products in France and exports them to Germany and Italy. So the company requires transportation services that operate internationally. Furthermore, the products produced by the company require specialized trucking transportation.

So now one needs to code a client application that receives the country of destination as a parameter, and performs the queries on the registry to find the services it can use. In this particular case, the type of transportation service and the origin country that are

submitted in the web page can be ignored by the application, as the type should be always specialized trucking and the origin must be always France.

But the country of destination must be considered. The idea is the same applied for client A, that is loading a profile from the file system that already specifies an specific type specialized and change only geographic radius descriptions to consider the country of destination. The Code Fragment 4-12 shows how this is coded into the API. Here the `getServiceParameters` method returns a collection of parameters for this profile description, including geographic radius.

```
Collection serviceParameters = profile.getServiceParameters();
if ((serviceParameters != null) && (serviceParameters.size() > 0)) {
        Iterator iteratorServiceParameters = serviceParameters.iterator();
        while (iteratorServiceParameters.hasNext()) {
                ServiceParameter serviceParameter =
                        (ServiceParameter) iteratorServiceParameters.next();
                if (serviceParameter instanceof GeographicRadius) {
                        if (destination.equals("de")) {
                                serviceParameter.setSparameter("http://www.iao.fhg.de/
services/owl-s/0.9/Country.owl#France and Germany");
                        } else if (destination.equals("it")) {
                                serviceParameter.setSparameter("http://www.iao.fhg.de/
services/owl-s/0.9/Country.owl#France_and_Italy");
                        }
                }
        }
}
```

Code Fragment 4-12: Modifying service parameters operation.

The code defined above reads through the collection of parameters until it finds an instance of geographic radius and then modifies its value depending on the value of the variable destination. Here again, destination is a parameter of type String, but it could be an object of any type. What is important is that an ontology Country is being used to tie the concepts of geographic radius defined in the client and in the remote registry.

After defining the new values for the geographic radius, a query operation can be placed exactly how it was done before in the case of client A. Supposing the country of destination is Italy, the query is going to return as a result two different services, one with geographic radius France and Italy, another with radius Germany and France and Italy. The screen containing the results is shown in Figure 4-6.

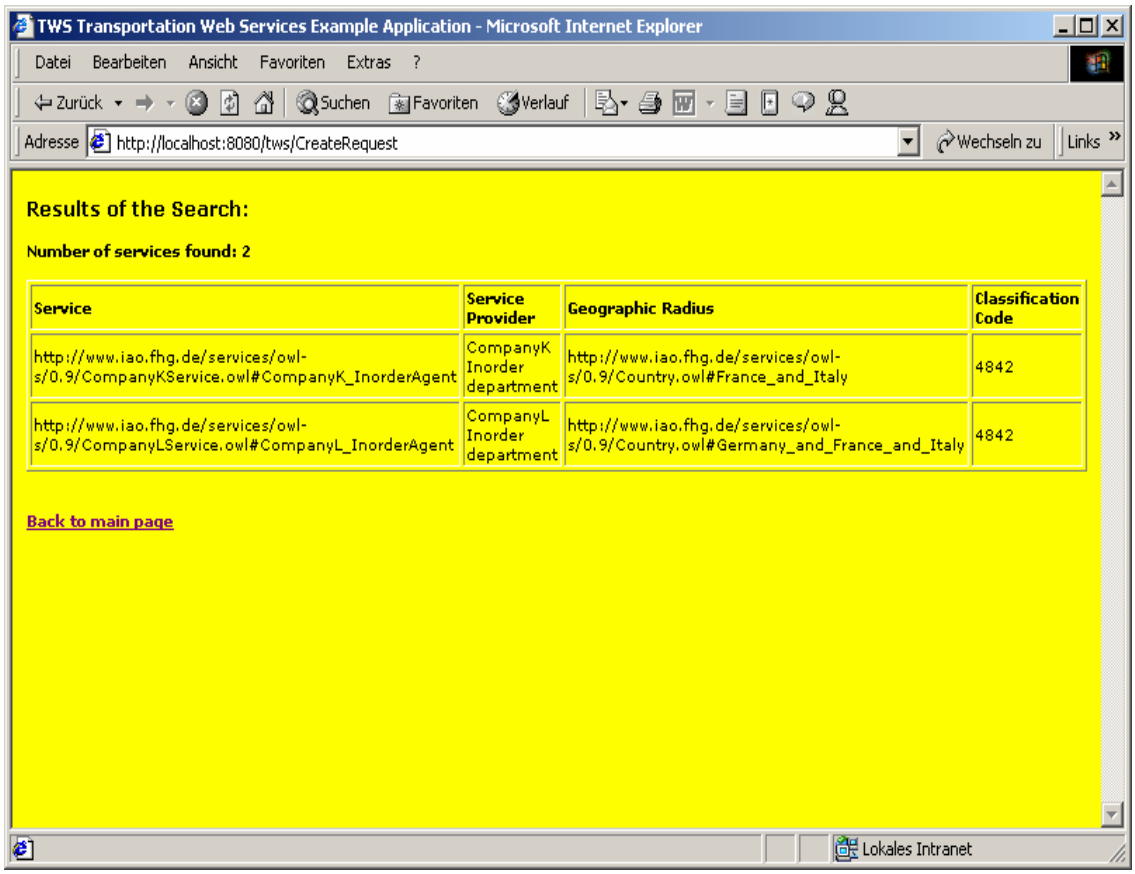

Figure 4-6: Result of search for specialiyed transportation in France and Italy.

4.4.8 Reasoning

The results described above were only possible because a semantic registry is able to perform reasoning operations and deduce that if one needs a service the transports between France and Italy, then a service that transports between Germany and France and Italy will also do the job. An important point here is whether one could define two different geographic radius for a profile and get the same result back.

The answer to this question depends upon the implementation. In fact, there is no rule specifying that a profile should have only one geographic radius. So, it all depends on how these rules are coded in the implementation of the registry. In the particular case of this prototype, only one geographic radius is allowed to be defined in the profile and specifying another one can lead to undesired results.

This would happen because the registry implemented here would try to match profiles with two geographic radius defined by the client, with profiles with a single radius stored in the registry, returning no results. The point that is important to notice is that the API is able to work with any implementation, allowing the developer to define one or two or more geographic radius for a profile, depending on what the registry supports.

This flexibility provided by the API is an important advantage as it allows it to support a broad range of applications that are to be defined in the future. But to continue the discussion about the API regarding query operations, some other advantages and

limitations must be presented. The next section discusses planning and composition issues.

4.4.9 Planning and Composition of Services

To lead the discussion on planning and composition, another client application needs to be introduced at this point. The client C manufactures its products in Germany and needs to transport them to Italy. The type of transportation can be general or specialized, depending on the product. At a first glance, this application looks very similar to the one defined for client A.

The difference one must notice here is that there are no services with geographic radius Germany and Italy defined in the registry, and so it is not possible for the application to simply change the geographic radius value and perform the query operation. A different approach is needed at this point.

In effect, a very simple solution here is to change the value of the geographic radius to Germany and France and Italy and perform the query. A company that covers the three areas will surely be able to transport from Germany to Italy. But the results of the query are somehow misleading because they will present a single company as the only service option, while another options are actually possible.

One such option is to combine the services of two different service providers, one that transports from Germany to France and other that transports from France to Italy. So, it is clear at this point that at least two different solutions are possible. The question is how the API provides support to the developer in this situation. In the next subsections some solutions are discussed and limitations and advantages of the API are pointed out.

4.4.9.1 Planning

A first obvious solution to the particular problem described above is to use the API to perform two different queries on the registry. In this case, a first query would be made for services with geographic radius Germany and France, and another for services with radius France and Italy. But two other problems derive from this alternative. The first problem is to decide what queries should the program perform and whether there is a particular order of execution.

The application should know, for example, that whenever it receives an order to find transportation service from Germany to Italy, it has to perform two different queries, combine the results, and then execute the services in a particular order. For this prototype, a solution would be simple because the application domain is not so complex and the developer could code this intelligence directly in the program.

But for more complex application domains, perhaps with hundreds of different routes, this intelligence would better be coded in a separate planner component. Based on the parameters of the request, this planner could reason about the different routes and pick the choose the queries it must perform to get to the results it needs. Based on the information provided by the component, the client application could then use the API to perform the queries.

So, even though the API does not offer design constructs to deal with these problems, which is certainly a limitation, it can still be used to develop applications and perform queries on the registry, as long as some other component on the top of it take control of the operation. But another problem remains to be solved that derives directly from the problem above, and that is the repetition of services descriptions in the joint result of the queries.

In the case above, for example, both queries performed on the registry would return the service with radius Germany and France and Italy as one of its results. The API in this case is not able to recognize this fact, and the program ends up with two instances of an object that represent and describe the same service. Here once more, in the case of complex applications, this might turn into a problem with a sum of instances of a same service lying in memory.

There is not an easy solution to this problem, and the API does not provide any support for it in its design constructs as well. It is up to the application running on the top of the API to read through the results and pick the ones that actually represent different services and compose them into a single service. In fact, this leads to the question of whether the API provides support for ranking, selection, and composition of the services. Figure 4-7 shows the result of the query for client C, considering a general type of transportation.

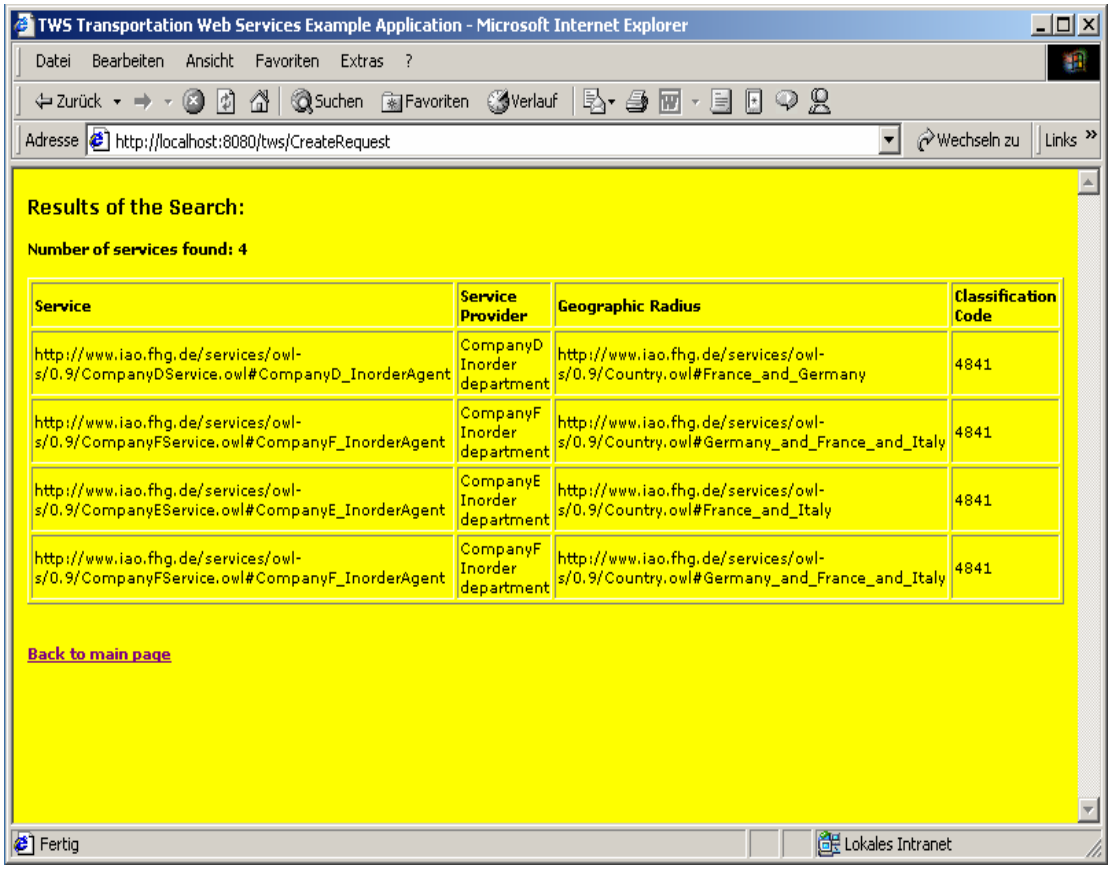

Figure 4-7: Result of search for general transportation from Germany to Italy.

4.4.9.2 Composition and Ranking

As it can be seen, four services have been return as result of the two queries and two of them actually represent the same service. Different possibilities of composition and execution derive from this situation. A possible solution here is to use solely the service of the company with radius Germany and France and Italy. Another possibility already discussed is to combine the services of the companies with radius France and Germany, and France and Italy.

Another not so obvious solution is to combine the services of the company with radius France and Germany with the one with radius France and Germany and Italy. The client application must then be able to read through the results of the queries performed using the API, combine them into single services, rank these combined services into a list and pick the top ranked service for execution.

Here again the API falls short as a tool to solve such problems. Once more, this comes from the fact that this API has been designed basically as tool to represent and manipulate OWL-S constructs inside a Java program, and moreover to query and life cycle operations on remote registries. Support for tasks such as composition, ranking and selection still must be coded on the top of the API.

4.4.9.3 Planning on the Registry

Another perhaps easier solution to this problem is to code these operations directly on the registry. In this case the client application could submit a profile with geographic radius Germany and Italy. The registry would then recognize this particular class, reason about the possible solutions, combine them into single services, perhaps rank them and return the result to the client application that could select the best service based on the ranking.

This is indeed a possible solution and the API in this case does not need to provide any further constructs to support it. In effect, the only requirement here is that the API implementer should provide an implementation of the Collection of objects returned as a result that is ordered, and so support the ranking of results. The downside of this solution is to overload the remote registry with specialized tasks that are not responsibility of a registry.

4.4.9.4 Planning on the API implementation

A final possible solution is to code these functions inside the API implementation itself. In fact, this is not advisable as it overloads the API and restricts its use to more specialized applications. In this case is the implementation of the API that performs the composition and ranking functionality before returning the results to the client. Here a note must be made that for performing ranking operations the OWL-S descriptions must provide enough data to allow for it.

In this prototype the descriptions do not allow for ranking based on the profile information, for example. The registry client or the component running on the top of it cannot make decisions based solely on geographic radius and NAICS information. Here

are good parameter for ranking would be perhaps the price of the service, but the application must first connect to the company web service and retrieve this information.

For the particular case of this prototype, it would be complex to code the composition and ranking functionality on the registry or in the API implementation, because these software components would need to place calls to the web services themselves and handle more complex operations than they were designed for. But in the case the ranking information is coded on the profile, it becomes easier to implement this functionality on these components.

As one can see, the API presents some limitations as it does not provide support for composition, ranking and selection directly on its design constructs. On the other side, it is flexible enough to provide for the development of applications with these functionalities. The decision whether the functionality should be implemented on the client, on the registry, or on the API implementation will depend on the application at hand, and should be taken carefully.

4.5 Life Cycle Operations

The last section discussed the use of the API for performing query operations on remote registries. This section analyzes how the API is used to code life cycle operations upon these registries. As the operations are introduced, the queries presented in the last section are rerun to present the effect of the changes in the prototype. Along the way, the limitations and advantages of using the API are discussed.

4.5.1 Inserting Services and Profiles

The first operation that needs to be presented is how to perform insert operations on the registry. Supposing a new company M wants to provide general transportation services with a geographic radius Germany, then this company can use the API to code the application that is going to connect to the remote registry and perform the insert operation that will save the new service description.

The insert operation is described in Code Fragment 4-13. In fact, the first step regarding the creation of connections is similar to creating connections for query operations and is not shown here. Once a connection is available, the developer can use it to get a registry service, and from the service, a life cycle manager that allows to perform operations on the remote registry.

```
FileProcessor fileProcessor = FileProcessor.newInstance();
Collection services = fileProcessor.readServices(serviceFilename);
RegistryService registryService = connection.getRegistryService();
BusinessLifeCycleManager lifeCycleManager =
       registryService.getBusinessLifeCycleManager();
BulkResponse response = lifeCycleManager.saveServices(services);
```

Code Fragment 4-13: Save services operation.

The developer uses the `readServices` method of the FileProcessor to read the OWL service file into a collection of service objects. Next step is to use the `saveServices` method of the BusinessLifeCycleManager to save the collection of services in the registry. Once the service description has been saved, the profile description for this service must also be inserted. Code Fragment 4-14 shows how the profile is saved.

```
FileProcessor fileProcessor = FileProcessor.newInstance();
Collection profiles = fileProcessor.readServiceProfiles(profileFilename);
RegistryService registryService = connection.getRegistryService();
BusinessLifeCycleManager lifeCycleManager =
       registryService.getBusinessLifeCycleManager();
BulkResponse response = lifeCycleManager.saveServiceProfiles(profiles);
```

Code Fragment 4-14: Save profiles operation.

As it can be seen, the code is not very different from the one used to save services. The difference is that now a `readServiceProfiles` method is being used to provide read profile descriptions into a collection of profile objects, while a `saveServiceProfiles` is used to insert the profiles of the collection in the remote registry. So, coding insert operations using API is a simple task, but there is a limitation regarding the link between services and profiles.

4.5.2 Linking Services and Profiles

The problem related to linking services and profiles comes from the fact that the object model does not provide any association between service and profile objects, leaving it up to the developer or to the API implementer to guarantee that the descriptions are consistent, meaning that the service description points to that profile and that the profile has also a link to the service.

In fact, the API could provide some mechanism for linking the services to profiles in its design constructs. On alternative would be to define an association between the interfaces that represent these concepts in the information model. The downside of this strategy is to make the information model more complex and the objects tightly coupled to each other, making it harder to manipulate them inside the code. So, a decision was taken to decouple these constructs.

Figure 4-8 shows the result of inserting this new service for the client A. As one might notice, the list of results now presents four different services the client company can utilize. The client software can connect to the services to obtain additional information about them and select the one that best fits its needs. The insertion of this new service has no effect for clients B and C, though.

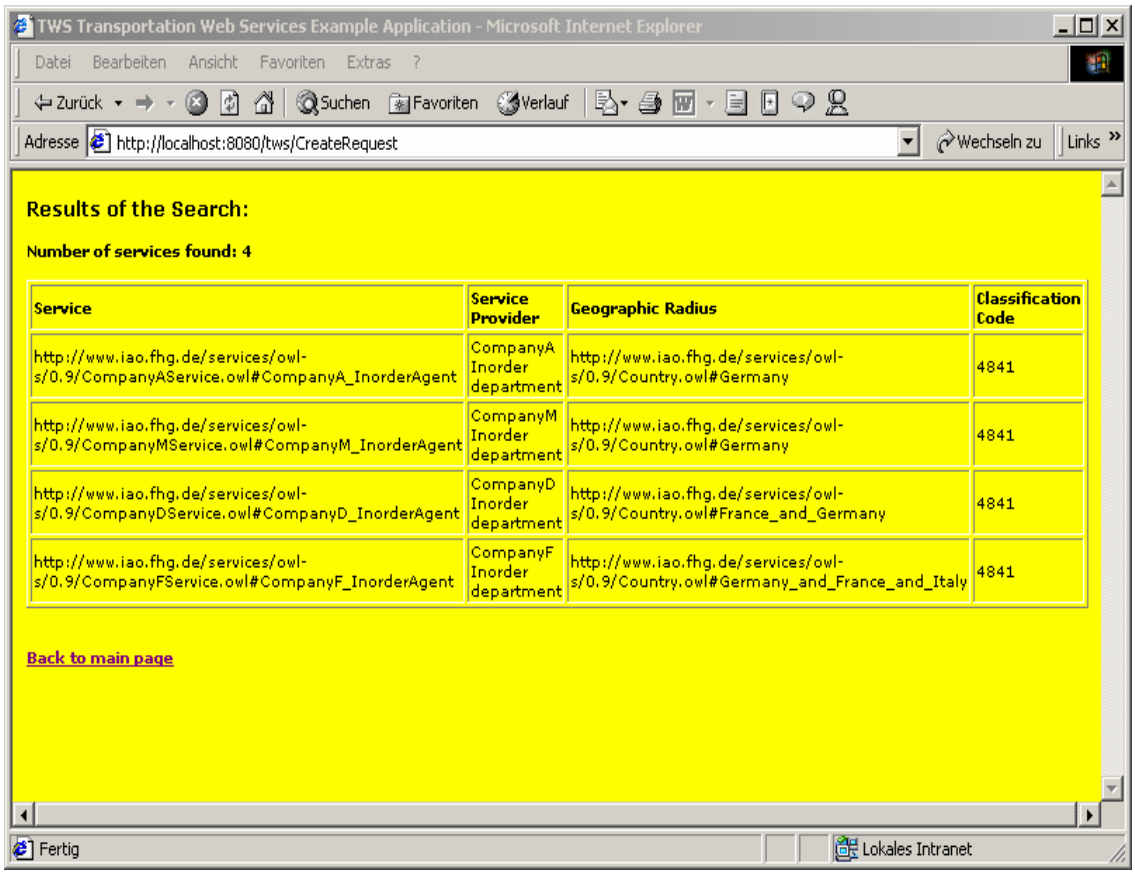

Figure 4-8: Results after inserting new service with radius Germany.

4.5.3 Updating Profiles

The next operation that needs to be shown is an update operation. Supposing company E, that provides general transportation services with a geographic radius France and Italy, now wants to provide also specialized trucking services in this region. The developer here can use the API to code the update operation in different ways. A simple alternative is to read the profile from the file system and use `saveServiceProfiles` to insert it on the registry.

This new profile should contain a unique ID and point to the existing service already saved on the registry. In addition, the NAICS service category defined on it should be specified with the specialized trucking NAICS code. This company would have then a service that is described by two different profiles. And now the query performed by the client B would return three different results instead of two, as shown in Figure 4-9.

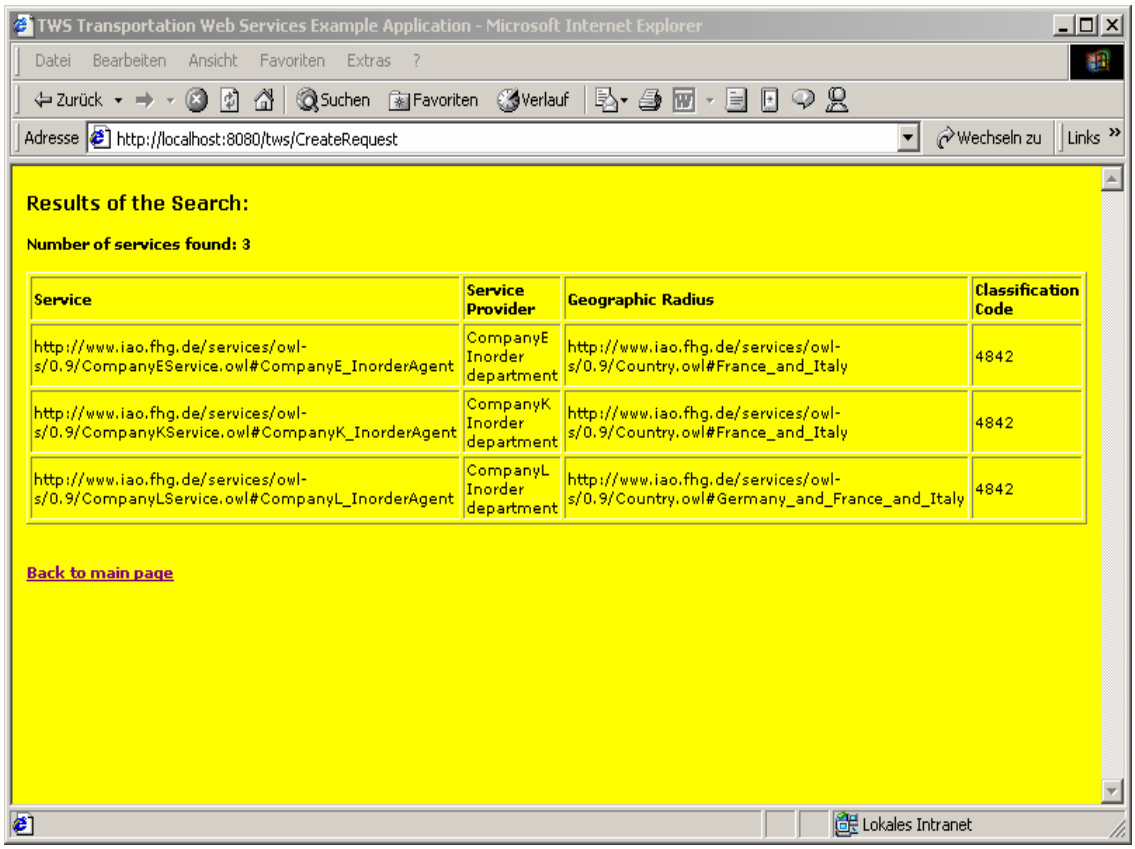

Figure 4-9: Results of updating service with radius France and Italy.

But the operation described above is actually an insert operation. To perform a real update operation, one would need to retrieve a profile from the registry, update some of its aspects and save it back using the same `saveServiceProfiles` method described above. In fact, another way to update the service description for company E is to modify its profile to include an additional NAICS category with value specialized trucking as shown in the Code Fragment 4-15.

```
ObjectFactory objectFactory = ObjectFactory.newInstance();
NAICS naics = (NAICS) objectFactory.createObject("de.fhg.iao.ws.owl.infomodel.NAICS");
naics.setId("NAICS-specialized");
naics.setValue("Specialized Freight Trucking");
naics.setCode("4842");
profile.addServiceCategory(naics);
```

Code Fragment 4-15: Creating a new NAICS object.

The code above creates an object of type NAICS using the factory method provided by the API. Once the object is created, it is added to the profile object that can be saved back in the registry. For the client B application, the result for the query operation will be the same as the one described in Figure 4-9. This is so because for this API implementation, the registry tries to match the single category provided by the user with at least one of the categories defined in the profile.

But the opposite might not be true, though. So, if the user provides two different categories in the profile it is using for the search, then the registry will return only

service descriptions that define both service categories in its profile. As for this particular case, it means the query would return as result only the service modified above, but the services presented before are not going to be shown anymore.

The examples above show at the same time the advantages and limitations of the API. Limitations because the API does not guarantee consistency between objects in the model and allows the developer to design applications as the one described above that might lead to erroneous results. Advantages because it gives the developer the freedom to produce code to support different solutions.

4.5.4 Deleting Services and Profiles

A final point to be discussed is how delete operations can be performed using the API. Suppose that company D that provides general transportation services with radius France and Germany decides to finish its operations and must delete its services from the registry. The Code Fragment 4-16 shows how this can be done using the API. The first step is, one more time, to read from the file system the profile description to be deleted.

```
FileProcessor fileProcessor = FileProcessor.newInstance();
Collection profiles = fileProcessor.readServiceProfiles(profileFilename);
ArrayList profileIds = new ArrayList();
if ((profiles != null) && (profiles.size() > 0)) {
        Iterator iteratorProfiles = profiles.iterator();
        while (iteratorProfiles.hasNext()) {
                Profile profile = (Profile) iteratorProfiles.next();
                profileIds.add(profile.getId());
        }
}
RegistryService registryService = connection.getRegistryService();
BusinessLifeCycleManager lifeCycleManager =
        registryService.getBusinessLifeCycleManager();
BulkResponse response = lifeCycleManager.deleteServiceProfiles(profileIds);
```

Code Fragment 4-16: Delete profiles operation.

The developer must then run through the collection of profiles to extract the ID of the profiles to be deleted and put them into a collection. The next step is to call the `deleteServiceProfiles` method of the API to perform the delete operation on the registry passing the collection of ID elements as a parameter. Figure 4-10 shows the result of this operation for the client A application.

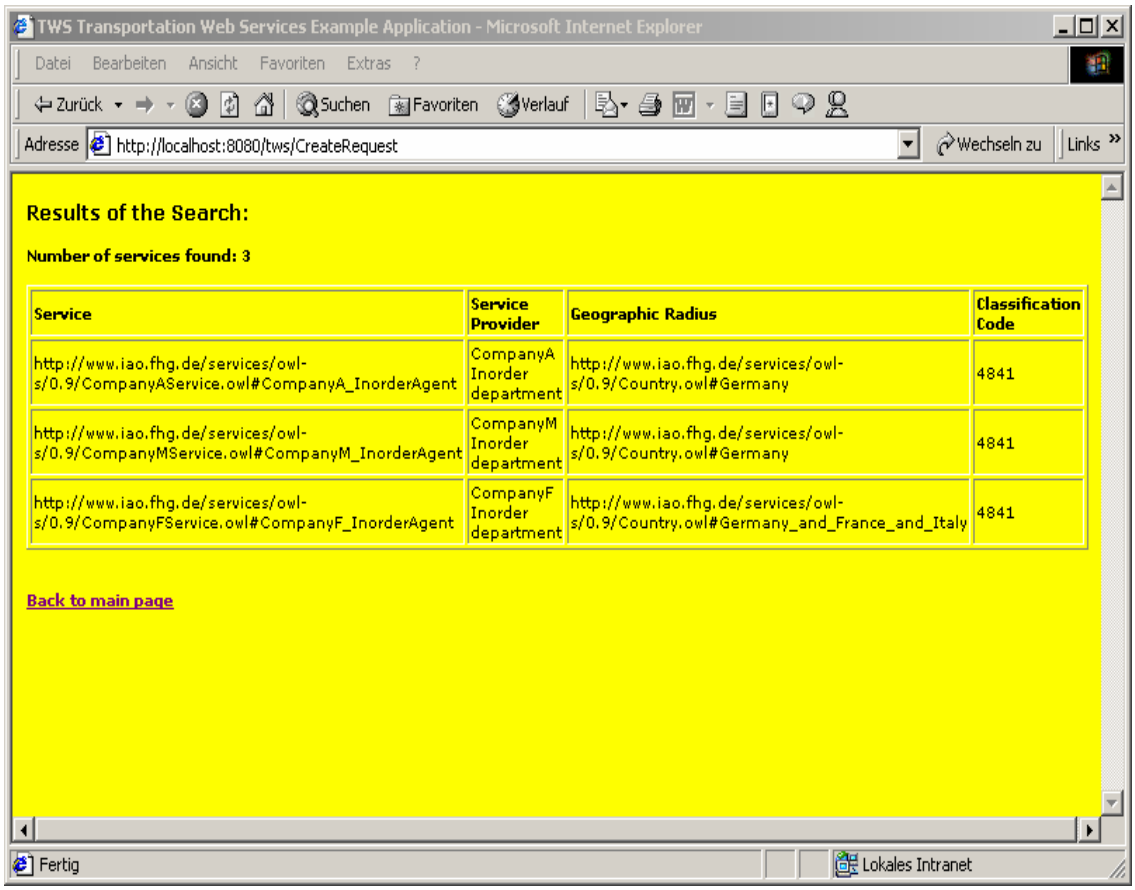

Figure 4-10: Result for search of client A after deleting service from company D.

Here a search for general services with radius Germany is going to return now three results instead of four. To be more specific, companies A, M, and F are included in the result, but company D is not presented anymore. The deletion of company D also has an impact on the query performed by client C application as it can be seen in Figure 4-11.

The client C application that would return four services as a result, after performing two queries on the service, now returns only three. As it can be seen, now the application running on top of the API has only two choices of transportation. It can either use the service provided by company F for the whole route, or it can use this service to transport from Germany to France, and then use company D to transport from France to Italy.

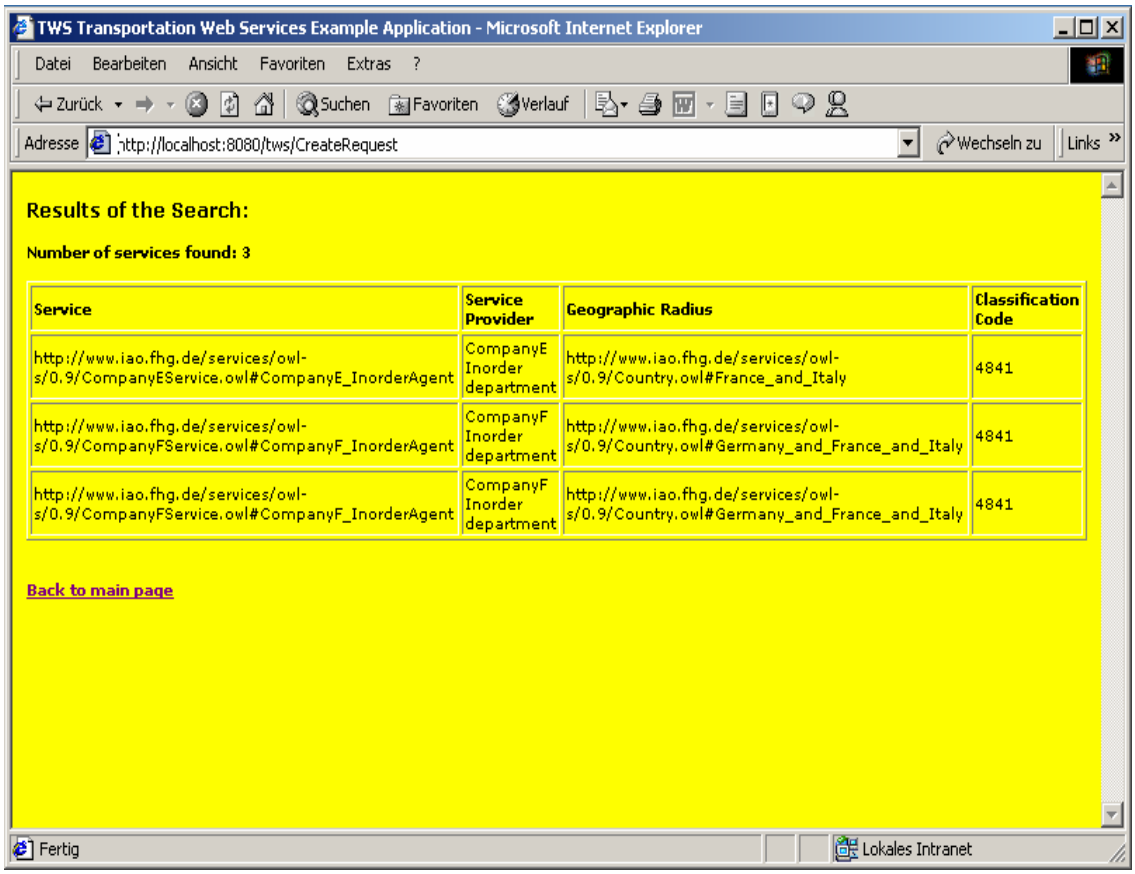

Figure 4-11: Result for search of client C after deleting service from company D.

4.6 Conclusion

This chapter presented a critical evaluation of the API on the context of an example application scenario. It first introduced a reference implementation to be used in the development of the prototype. Next, it described the application scenario for the prototype. Finally, the chapter discussed how the API is used on the development of functions regarding query and life cycle operations in the context of the prototype.

The first part of the chapter presented a proof of concept reference implementation of the API. In this implementation different packages were designed to deal with different functions such as object creation, file processing and registry operations. In particular, the design upon which the implementation of the registry operations was organized is a singular one and the work defends that this arrangement could turn out to be part of the specification in the future.

The registry operations are implemented in classes that delegate the execution to a single interface. This interface is implemented by a couple of classes lying on a package that communicates specifically with UDDI registries to perform the operations. This same arrangement could be used in the future to provide implementations to different kinds of registries. The developer would configure the type of registry it wants through the connection factory properties.

In effect, defining the RegistryProvider interface as part of the API would give the API implementers a specification under which they could develop different solutions to specific registries as database drivers are developed today for different database implementations. But the definition of such an interface also holds back the implementation of different solutions for the API, and so this work decided not to place it as part of the specification for the moment.

The reference implementation shows that implementing the API design constructs is viable and that a number of possibilities is actually possible. In the particular case described in this work, a solution was developed that uses SAX parsers for XML processing and the JAXR API to connect to UDDI registries and perform operations. But each API implementer should organize his implementation in a way that best fits its needs.

Finally, the chapter presented a prototype that used the API to develop service discovery applications in the example application scenario of the transportation industry. The examples showed how the developer can use the design constructs of the API to manipulate OWL-S files and perform operations on the registry in a rather simple way. As the operations were executed the results were presented to foster a discussion upon it.

The discussion called the attention to some issues that are not supported by the API such as planning, composition, ranking and selection of services as well as the importance of ontology design. The API could define design constructs to deal with these issues, but a choice was made for simplicity. In this case, it is up to the developer then to provide the functionality for planning, composition, ranking and selection, running on the top of the API.

Another solution discussed is to let the registry itself to perform these operations. In this case, the burden falls over the developer, that uses the API in a more simple manner. The discussion showed also that the way the ontologies are used describe the service and the way the services themselves are described using OWL-S play a important role on how easy it is going to be to develop service discovery applications using the API.

In effect, the example shows that by simply defining an element on the service description that can be used for ranking, such as the price of the service, could mean that the registry could take the burden of ranking and selection of the services away from the developer. On the other hand, this would surely represent more work for the registry. So, it is clear that a compromise must exist among clients, service providers and registry providers.

The important point to notice is that whatever decision is taken regarding the architecture of the application or the design of the ontologies, the API is flexible enough to be used in any case. The flexibility derives from simplicity of the design constructs of the API already mentioned above. On the other hand, this flexibility requires an extra attention of the developer, especially in dealing with different implementations of the API.

This means that a program that is developed to provide discovery functions for services defined in a registry might not work with some other registries that do not provide the

same type of functionality as, for example, ranking. On the other side, even different implementations of the API that are not compatible can derive in undesired results for the same program coded using the API.

The conclusion is that the same flexibility provided by the API might result in a restriction of its use, that might lead the developer to design different programs to perform the same function for different registry or API implementations. There is in fact no easy solution to this problem, though. It is up to the developer to ensure that the API and registries implementations he works with are compatible.

5 Conclusion

This thesis presented a Java API for the development of service discovery applications based on OWL-S. In the first part of the work, some of the standards and technologies regarding web services and the semantic web were presented. The complementary roles of these technologies and how their combination can lead to the development of more powerful applications was discussed.

As an example, applications could be developed that would be able to discover services on the web, select the ones it needs and place remote calls to them without human intervention. The work discussed the need for new standards and tools to support this vision. In particular, the DAML-S and OWL-S ontologies for the semantic description of web services were presented and applications based on DAML-S were introduced.

Even though these technologies are regarded as an important development, the lack of tool support for these ontologies was pointed out as a shortcoming. To cope in part with this problem, the design and implementation of a Java API was proposed that enables the application developer to represent and manipulate OWL-S constructs inside the program and to perform service discovery operations on remote registries.

The API provides a set of interfaces and associated methods that abstract the developer from implementation details related to connection management and life cycle and query operations, so that he can focus on the business rules. The tool also defines a response model that provides support for partial responses and asynchronous operations, and an exception model that deals with the exceptional situations that might occur during operation.

The thesis also presented a proof of concept reference implementation that is based on UDDI registries and provides some specific reasoning functionality coded on it. The reference implementation shows that implementing the design constructs defined in the API specification is viable, and that a number of possibilities is actually possible. This implementation was then used to develop a prototype in the context of a specific example application scenario.

The development of the prototype showed some of the shortcomings of the API such as lack of support for planning and composition of queries, and ranking and selection of the results. Despite the problems described above, the API proved to be a quite simple and useful framework, abstracting the developer from implementation details and leading to the development of applications that are easier to program and maintain.

5.1 Future Work

The service discovery API described in this work is an initial point to a more broad OWL-S API that is able to perform not only service discovery tasks, but that also provides design constructs and methods that will enable the composition and execution of web services based on OWL-S descriptions. To reach this goal some work needs to be done and some of the tasks are briefly described in this section.

The first task that needs to be mentioned is the improvement of the API design constructs and reference implementation to cover aspects that were not covered in this version. Regarding design constructs, as the API starts to be used the need additional functionality might become clear, and so the need to incorporate these issues in the next versions. The design constructs should also be updated to mirror the release 1.0 of the OWL-S specification.

On the other side, the reference implementation should be further developed to provide support to other registry implementations such as the Matchmaker. In addition, some of functionality that was not implemented in this version such as declarative queries, asynchronous calls and federated connections should be implemented as well. Finally, the registry provider interfaces should be revised to be considered as part of the specification.

A final task toward a complete OWL-S API is the definition of constructs in the information model to represent service grounding and service model descriptions. Moreover, design constructs will need to be defined with associated reference implementation that can be used by the developer to execute the process models and perform calls to the web services based on the service model and grounding information.

A OWL-S Web Services Descriptions

A.1 CompanyAService.owl

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

  <!DOCTYPE uridef (View Source for full doctype...)>

  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
xmlns:owl="http://www.w3.org/2002/07/owl#" xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
xmlns:service="http://www.daml.org/services/owl-s/0.9/Service.owl#"
xmlns="http://www.iao.fhg.de/services/owl-s/0.9/CompanyAService.owl#">

  <owl:Ontology>
    <owl:versionInfo>$Id: CompanyAService.owl,v 1.0 $</owl:versionInfo>
    <rdfs:comment>CompanyA transportation service.</rdfs:comment>
    <owl:imports rdf:resource="http://www.w3.org/2002/07/owl" />
    <owl:imports rdf:resource="http://www.daml.org/services/owl-s/0.9/Service.owl" />
    <owl:imports rdf:resource="http://www.iao.fhg.de/services/owl-
s/0.9/CompanyAProfile.owl" />
    <owl:imports rdf:resource="http://www.iao.fhg.de/services/owl-
s/0.9/CompanyAProcess.owl" />
  </owl:Ontology>

  <service:Service rdf:ID="CompanyA_InorderAgent">

    <!--  Reference to the CompanyA Profile -->
    <service:presents rdf:resource="http://www.iao.fhg.de/services/owl-
s/0.9/CompanyAProfile.owl#Profile_CompanyA_InorderAgent" />

    <!--  Reference to the CompanyA Process Model -->
    <service:describedBy rdf:resource="http://www.iao.fhg.de/services/owl-
s/0.9/CompanyAProcess.owl#CompanyA_InorderAgent_ProcessModel" />

    <!--  Reference to the CompanyA Grounding -->
    <service:supports rdf:resource="http://www.iao.fhg.de/services/owl-
s/0.9/CompanyAGrounding.owl#Grounding_CompanyA_InorderAgent" />
  </service:Service>

</rdf:RDF>
```

A.2 CompanyAProfile.owl

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE uridef (View Source for full doctype...)>

<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
xmlns:owl="http://www.w3.org/2002/07/owl#"
xmlns:service="http://www.daml.org/services/owl-s/0.9/Service.owl#"
xmlns:process="http://www.daml.org/services/owl-s/0.9/Process.owl#"
xmlns:profile="http://www.daml.org/services/owl-s/0.9/Profile.owl#"
xmlns="http://www.iao.fhg.de/services/owl-s/0.9/CompanyAProfile.owl#">

  <owl:Ontology>
    <owl:versionInfo>$Id: CompanyAProfile.owl, v 1.0 $</owl:versionInfo>
    <rdfs:comment>CompanyA transportation service profile.</rdfs:comment>
    <owl:imports rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns" />
    <owl:imports rdf:resource="http://www.w3.org/2000/01/rdf-schema" />
    <owl:imports rdf:resource="http://www.w3.org/2002/07/owl" />
    <owl:imports rdf:resource="http://www.daml.org/services/owl-s/0.9/Service.owl" />
    <owl:imports rdf:resource="http://www.daml.org/services/owl-s/0.9/Profile.owl" />
    <owl:imports rdf:resource="http://www.daml.org/services/owl-s/0.9/Process.owl" />
    <owl:imports rdf:resource="http://www.iao.fhg.de/services/owl-
s/0.9/CompanyAService.owl" />
```

```
    <owl:imports rdf:resource="http://www.iao.fhg.de/services/owl-
s/0.9/CompanyAProcess.owl" />
    <owl:imports rdf:resource="http://www.iao.fhg.de/services/owl-s/0.9/Concepts.owl" />
    <owl:imports rdf:resource="http://www.iao.fhg.de/services/owl-s/0.9/Country.owl" />
  </owl:Ontology>

  <profile:Profile rdf:ID="Profile_CompanyA_InorderAgent">

    <!--  reference to the service specification -->
    <service:presentedBy rdf:resource="http://www.iao.fhg.de/services/owl-
s/0.9/CompanyAService.owl#CompanyA_InorderAgent" />

    <!--  reference to the process model specification -->
    <profile:has process rdf:resource="http://www.iao.fhg.de/services/owl-
s/0.9/CompanyAProcess.owl#CompanyA_Process" />

    <profile:serviceName>CompanyA_InorderAgent</profile:serviceName>
    <profile:textDescription>Inorder agent service</profile:textDescription>

    <profile:contactInformation>
      <profile:Actor rdf:ID="CompanyA-inorder">
        <profile:name>CompanyA Inorder department</profile:name>
        <profile:title>Inorder Representative</profile:title>
        <profile:phone>412 268 8780</profile:phone>
        <profile:fax>412 268 5569</profile:fax>
        <profile:email>Inorder@CompanyA.com</profile:email>
        <profile:physicalAddress>CompanyAStrasse 20</profile:physicalAddress>
        <profile:webURL>http://www.companyA.com/</profile:webURL>
      </profile:Actor>
    </profile:contactInformation>

    <!--  description of Geographic radius as a service parameter -->
    <profile:serviceParameter>
      <profile:GeographicRadius rdf:ID="CompanyA-geographicRadius">
        <profile:serviceParameterName>CompanyA Geographic
Radius</profile:serviceParameterName>
        <profile:sParameter rdf:resource="http://www.iao.fhg.de/services/owl-
s/0.9/Country.owl#Germany" />
      </profile:GeographicRadius>
    </profile:serviceParameter>

    <!--  Specification of the service category using NAICS -->
    <profile:serviceCategory>
      <profile:NAICS rdf:ID="NAICS-category">
        <profile:value>General Freight Trucking</profile:value>
        <profile:code>4841</profile:code>
      </profile:NAICS>
    </profile:serviceCategory>

    <!--  Descriptions of IOPEs -->
    <profile:input>
      <profile:ParameterDescription rdf:ID="DropoffLocation">
        <profile:parameterName>DropoffLocation</profile:parameterName>
        <profile:restrictedTo rdf:resource="http://www.iao.fhg.de/services/owl-
s/0.9/Concepts.owl#Location" />
        <profile:refersTo rdf:resource="http://www.iao.fhg.de/services/owl-
s/0.9/CompanyAProcess.owl#DropoffLocation_In" />
      </profile:ParameterDescription>
    </profile:input>
    <profile:input>
      <profile:ParameterDescription rdf:ID="PickupLocation">
        <profile:parameterName>PickupLocation</profile:parameterName>
        <profile:restrictedTo rdf:resource="http://www.iao.fhg.de/services/owl-
s/0.9/Concepts.owl#Location" />
        <profile:refersTo rdf:resource="http://www.iao.fhg.de/services/owl-
s/0.9/CompanyAProcess.owl#pickupLocation_In" />
      </profile:ParameterDescription>
    </profile:input>
   <profile:input>
     <profile:ParameterDescription rdf:ID="DepartureDate">
       <profile:parameterName>DepartureDate</profile:parameterName>
       <profile:restrictedTo rdf:resource="http://www.iao.fhg.de/services/owl-
s/0.9/Concepts.owl#Date" />
       <profile:refersTo rdf:resource="http://www.iao.fhg.de/services/owl-
s/0.9/CompanyAProcess.owl#departureDate_In" />
     </profile:ParameterDescription>
   </profile:input>
```

```
   <profile:input>
     <profile:ParameterDescription rdf:ID="ArrivalDate">
       <profile:parameterName>ArrivalDate</profile:parameterName>
       <profile:restrictedTo rdf:resource="http://www.iao.fhg.de/services/owl-
s/0.9/Concepts.owl#Date" />
       <profile:refersTo rdf:resource="http://www.iao.fhg.de/services/owl-
s/0.9/CompanyAProcess.owl#arrivalDate_In" />
     </profile:ParameterDescription>
   </profile:input>
   <profile:output>
     <profile:ParameterDescription rdf:ID="AvailableServices">
       <profile:parameterName>AvailableServices</profile:parameterName>
       <profile:restrictedTo rdf:resource="http://www.iao.fhg.de/services/owl-
s/0.9/Concepts.owl#AvailableServicesList" />
       <profile:refersTo rdf:resource="http://www.iao.fhg.de/services/owl-
s/0.9/CompanyAProcess.owl#availableServicesList_Out" />
     </profile:ParameterDescription>
   </profile:output>
   <profile:input>
     <profile:ParameterDescription rdf:ID="LoginName">
       <profile:parameterName>LoginName</profile:parameterName>
       <profile:restrictedTo rdf:resource="http://www.iao.fhg.de/services/owl-
s/0.9/Concepts.owl#AcctName" />
       <profile:refersTo rdf:resource="http://www.iao.fhg.de/services/owl-
s/0.9/CompanyAProcess.owl#loginName In" />
     </profile:ParameterDescription>
   </profile:input>
  <profile:input>
    <profile:ParameterDescription rdf:ID="Password">
      <profile:parameterName>Password</profile:parameterName>
      <profile:restrictedTo rdf:resource="http://www.iao.fhg.de/services/owl-
s/0.9/Concepts.owl#Password" />
      <profile:refersTo rdf:resource="http://www.iao.fhg.de/services/owl-
s/0.9/CompanyAProcess.owl#password_In" />
    </profile:ParameterDescription>
  </profile:input>
  <profile:input>
    <profile:ParameterDescription rdf:ID="Confirmation">
      <profile:parameterName>Confirmation</profile:parameterName>
      <profile:restrictedTo rdf:resource="http://www.iao.fhg.de/services/owl-
s/0.9/Concepts.owl#Confirmation" />
      <profile:refersTo rdf:resource="http://www.iao.fhg.de/services/owl-
s/0.9/CompanyAProcess.owl#confirmation_In" />
    </profile:ParameterDescription>
  </profile:input>
  <profile:output>
    <profile:ParameterDescription rdf:ID="InorderNumber">
      <profile:parameterName>Inorder Number</profile:parameterName>
      <profile:restrictedTo rdf:resource="http://www.iao.fhg.de/services/owl-
s/0.9/Concepts.owl#InorderNumber" />
      <profile:refersTo rdf:resource="http://www.iao.fhg.de/services/owl-
s/0.9/CompanyAProcess.owl#inorderNumber_Out" />
    </profile:ParameterDescription>
  </profile:output>
  <profile:effect>
    <profile:ParameterDescription rdf:ID="HaveInorder">
      <profile:parameterName>HaveInorder</profile:parameterName>
      <profile:restrictedTo rdf:resource="http://www.iao.fhg.de/services/owl-
s/0.9/Concepts.owl#HaveInorder" />
      <profile:refersTo rdf:resource="http://www.iao.fhg.de/services/owl-
s/0.9/CompanyAProcess.owl#haveInorder" />
    </profile:ParameterDescription>
  </profile:effect>
</profile:Profile>
</rdf:RDF>
```

List of Abbreviations

BPEL4WS Business Process Execution Language for Web Services

DAML DARPA Agent Markup Language

DAML-S DAML-based Web Services Ontology

DQL DAML Query Language

JAXR Java API for XML Registries

JMS Java Messaging Services

JTP Java Theorem Prover

NAICS North American Industry Classification System

OIL Ontology Inference Layer

OWL Ontology Web Language

OWL-S OWL-based Web Services Ontology

RDF Resource Description Framework

SAAJ SOAP with Attachments API for Java

SOAP Simple Object Access Protocol

UDDI Universal Discovery, Description and Integration

URI Uniform Resource Identifier

WSCI Web Services Choreography Interface

WSDL Web Services Description Language

XML Extensible Markup Language

References

[1] E. Armstrong, J. Ball, S. Bodoff, D. Carson, I. Evans, M. Fisher, D. Green, K. Hasse, E. Jendrock. The J2EE 1.4 Tutorial, http://java.sun.com/j2ee/1.4/docs/tutorial/doc/index.html, 2003. Last accessed on: 26.01.2004.

[2] Sun Microsystems. Web Services Made Easier. The Java APIs and Architectures for XML. http://java.sun.com/xml/webservices.pdf, 2002. Last accessed on: 26.01.2004.

[3] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. Nielsen, S. Thatte, and D. Winer. Simple Object Access Protocol (SOAP) 1.1. http://www.w3.org/TR/SOAP/, 2000. Last accessed on: 26.01.2004.

[4] UDDI. The UDDI Technical White Paper. http://www.uddi.org/pubs/Iru_UDDI_Technical_White_Paper.pdf, 2000. Last accessed on: 26.01.2004.

[5] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Definition Language (WSDL) 1.1. http://www.w3.org/TR/wsdl, 2001. Last accessed on: 26.01.2004.

[6] P. Brittenham, F. Curbera, D. Ehnebuske, S. Graham. Understanding WSDL in a UDDI registry. How to publish and find WSDL service descriptions. http://www-106.ibm.com/developerworks/library/ws-wsdl/, 2002. Last accessed on: 26.01.2004.

[7] K. Ballinger, D. Ehnebuske, M. Gudgin, M. Nottingham, P. Yendluri. Basic Profile Version 1.0a, http://www.ws-i.org/Profiles/Basic/2003-08/BasicProfile-1.0a.html, 2003. Last accessed on: 26.01.2004.

[8] A. Arkin, S. Askary, S. Fordin, W. Jekeli, K. Kawaguchi, D. Orchad, S. Pogliani, K. Riemer, S. Struble, P. Takacsi-Nagy, I. Trickovic, and S. Zimek. Web Services Choreography Interface (WSCI) 1.0. http://www.w3.org/TR/2002/NOTE-wsci-20020808/, 2002. Last accessed on: 26.01.2004.

[9] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services 1.1, http://www-106.ibm.com/developerworks/library/ws-bpel/, 2003. Last accessed on: 26.01.2004.

[10] F. Cabrera, G. Copeland, B. Cox, T. Freund, J. Klein, T. Storey, S. Thatte. Web Services Transaction (WS-Transaction), http://www-106.ibm.com/developerworks/webservices/library/ws-transpec/, 2002. Last accessed on: 26.01.2004.

[11] B. Atkinson, G. Della-Libera, S. Hada, M. Hondo, P. Halam-Baker, J. Klein, B. LaMacchia, P. Leach, J. Manferdelli, H. Maruyama, A. Nadalin, N. Nagaratnam, H. Prafullchandra, J. Shewchuck, D. Simon. Web Services Security (WS-Security) version

1.0, http://www-106.ibm.com/developerworks/webservices/library/ws-secure/, 2002. Last accessed on: 26.01.2004.

[12] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5):34-43, 2001.

[13] O. Lassila and R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/, 1999. Last accessed on: 26.01.2004.

[14] W3C World Wide Web Consortium. Naming and Addressing. http://www.w3.org/Addressing/, 2002. Last accessed on: 26.01.2004.

[15] DAML DARPA Agent Markup Language Program. http://www.daml.org, 2003.

[16] OIL Ontology Inference Layer. http://www.ontoknowledge.org/oil/, 2003. Last accessed on: 26.01.2004.

[17] I. Horrocks, F. Harmelen, P. Patel-Schneider, T. Berners-Lee, D. Brickley, D. Connolly, M. Dean, S. Decker, D. Fensel, R. Fikes, P. Hayes, J. Heflin, J. Hendler, O. Lassila, D. McGuinness, L. Stein. DAML+OIL. http://www.daml.org/2001/03/daml+oil-index.html, 2001. Last accessed on: 26.01.2004.

[18] DAML Tools. http://www.daml.org/tools, 2003. Last accessed on: 26.01.2004.

[19] N. Noy, D. McGuiness. Ontology Development 101: A Guide to Creating your First Ontology. http://protege.stanford.edu/publications/ontology_development/ontology101.pdf, 2001. Last accessed on: 26.01.2004.

[20] S. Bechhofer, F. Harmelen, J. Hendler, I. Horrocks, D. McGuiness, P. Patel-Schneider, L. Stein. OWL Web Ontology Language Reference. http://www.w3.org/TR/owl-ref/, 2003. Last accessed on: 26.01.2004.

[21] DAML DARPA Agent Markup Language Program. Language Feature Comparison: http://www.daml.org/language/features.html, 2001. Last accessed on: 26.01.2004.

[22] DAML Services. http://www.daml.org/services/, 2003. Last accessed on: 26.01.2004.

[23] A. Ankolenkar, M. Burstein, J. Hobbs, O. Lassila, D. Martin, S. McIlraith, S. Narayanan, M. Paolucci, T. Payne, K. Sycara, and H. Zeng. DAML-S Semantic Markup for Web Services. *Proceedings of the International Semantic Web Working Symposium (SWWS)*, 2001.

[24] A. Ankolenkar, M. Burstein, J. Hobbs, O. Lassila, D. Martin, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, T. Payne, and K. Sycara. DAML-S: Web Service

Description for the Semantic Web, *The First International Semantic Web Conference (ISWC)*, 2002.

[25] D. Martin, M. Burstein , G. Denker , J. Hobbs , L. Kagal , O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, K. Sycara. DAML-S and OWL-S 0.9 draft Release. http://www.daml.org/services/daml-s/0.9/, 2003. Last accessed on: 26.01.2004.

[26] D. Richards, M. Sabou, and S. van Splunter. An Experience Report on using DAML-S. *WWW2003*, 2003.

[27] M. Paolucci, T. Payne, and K. Sycara. Advertising and Matching DAML-S Service Descriptions. *Semantic Web Working Symposium (SWWS)*, 2001.

[28] T. Kawamura, M. Paolucci, T. Payne, and K. Sycara. Semantic Matching of Web Services Capabilities. *International Semantic Web Conference (ISWC)*, 2002.

[29] T. Kawamura, M. Paolucci, T. Payne, and K. Sycara. Importing the Semantic Web in UDDI. *Proceedings Web Services, E-Business and Semantic Web Workshop*, 2002.

[30] D. Mandell and S. McIlraith. Adapting BPEL4WS for the Semantic Web: The Bottom-Up Approach to Web Service Interoperation. *Second International Semantic Web Conference (ISWC2003)*, 2003.

[31] F. Najmi. Java API for XML Registries (JAXR) version 1.0. http://java.sun.com/xml/downloads/jaxr.html, 2002. Last accessed on: 26.01.2004.

[32] Apache AXIS. AXIS Architecture Guide. http://ws.apache.org/axis/java/architecture-guide.html, 2003.

[33] E. Armstrong, J. Ball, S. Bodoff, D. Carson, I. Evans, M. Fisher, S. Fordin, D. Green, K. Hasse, E. Jendrock. The Java Web Services Tutorial. http://java.sun.com/webservices/docs/1.3/tutorial/doc/index.html, 2003. Last accessed on: 26.01.2004.

[34] D. Tidwell. UDD4J: Matchmaking for Web Services. http://www-106.ibm.com/developerworks/library/ws-uddi4j.html, 2001. Last accessed on: 26.01.2004.

[35] Carnegie Mellon University. The Semantic DAML-S Matchmaker: Instructions for Use. http://www-2.cs.cmu.edu/~softagents/daml_Mmaker/daml-s_matchmaker_instructions.htm, 2001. Last accessed on: 26.01.2004.