MATCHING OF WEB SERVICE SPECIFICATIONS USING DAML-S DESCRIPTIONS

Thesis / Diplomarbeit

Stefan Tang
March 18th, 2004

Gutachter:

Prof. Dr. Kurt Geihs
FG Intelligente Netze und Management verteilter Systeme
Fakultät IV - Institut für Telekommunikationssysteme
Technische Universität Berlin

Dr. Gero Mühl
FG Intelligente Netze und Management verteilter Systeme
Fakultät IV - Institut für Telekommunikationssysteme
Technische Universität Berlin

Die selbständige und eigenhändige Anfertigung dieser Arbeit versichere ich an Eides statt.

_____

Stefan Tang

Acknowledgments

I would like to express my special gratitude towards Michael C. Jaeger[1], who advised me on this thesis. As most of this work was done while I was studying at Stanford University, it took more effort than usual to communicate with one another, and I appreciate his time and effort spent in supervising me.

[1] *FG Intelligente Netze und Management verteilter Systeme, Technische Universität Berlin*

Abstract

The emergence of the Semantic Web allows for the unambiguous interpretation of web resources and content through the use of shared web ontologies. Furthermore, ontologies based on DAML-S provide a semantic markup for Web Services. In this work we will investigate how the available semantic information of Web Services can be utilized for achieving a higher degree of automatic interoperability between services. In specific, a matching procedure is developed, and it is designed for allowing one to find Web Services that will satisfy certain requirements. A framework for the useful deployment of this matching procedure in existing environments is given, along with a description of the actual implementation.

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

In recent years Web Services [17] have become the predominant way with which distributed computer programs communicate over the Internet. Web Services greatly enhance the interoperability of programs since they rely solely on standardized internet protocols, such as XML [7], WSDL [6] and SOAP [25]. Today there exists a vast amount of tools that support and facilitate the development, deployment and invocation of Web Services, and there are virtually no limits as to what Web Services can do within their realm.

Web Services offer a great way for automating certain processes. One of these is the possibility of the automatic invocation of a Web Service. An application can autonomously connect and call a remote service, in the simplest case all that is required is the address of the Web Service. Given that address, the application can retrieve a document that describes the Web Service, and from that description it can automatically generate classes or programs (so called stubs or proxies) that call the remote Web Service.

Even though these mechanisms provide means for greater automation and interoperability for distributed applications, there is still one drawback that cannot be completely resolved with existing technologies.

The question that arises is how do we determine what Web Service we want to make use of? Once we know which Web Service we need to invoke, we are set and can let our applications do all the work. But what happens if, for example, we frequently change the requirements we pose towards a Web Service? Or if, in a dynamic environment like the Internet a very likely scenario, the provider of the service we used to invoke goes out of business? In any of these situations, there has to be some kind of manual interaction by a user to solve these problems. With current Web Service mechanisms, users will have to find an appropriate service manually, and determine whether a particular Web Service provides the functionality that is requested or not. This is mostly done by browsing a directory service such as UDDI [11] or by directly obtaining a Web Service description, from a business partner for example. As businesses try to minimize the amount of time and work they have to put into maintaining and running their applications, we would like to develop some kind of mechanism that allows us to overcome this shortcoming.

This is where the Semantic Web [24] comes into play. The Semantic Web is an effort by the W3C consortium [20], and one of its main purposes is to facilitate the discovery of web resources. The Semantic Web should enable the discovery of resources by content, and not just by plain keyword search, as it is mostly done nowadays. This is achieved by adding semantic information to web resources. Through shared conceptualizations (so called ontologies) these web resources are given a pre-defined meaning and will thus be machine understandable.

A standard has emerged that provides such a semantic markup for Web Services. This standard was developed by the DAML consortium and is called DAML-S [8]. By means of DAML-S we are now able to define the "purpose" of a Web Service by giving it a semantic meaning. Web Services that come along with semantic information therefore become meaningful to computer programs. This will enable computer programs to autonomously decide whether a particular Web Service satisfies certain requirements or not. This will result in even greater automation of business processes and work flow management.

The goal of this work is to develop a mechanism that will assist in selecting an appropriate Web Service for a given task. The algorithm that will be developed in this work will decide if a Web Service, based on the available semantic information encoded in DAML-S, fulfills the requirements that another program or user posed towards any requested service. More specifically, it will create a ranking for the compatibility of the Web Service(s) and the requested needs. Such a ranking will eventually become necessary since it is highly unlikely that there is always a Web Service that offers the exact functionality one expects, especially in generic setups, where applications anticipate the eventual existence of some specific service. Based on these rankings, computer programs (or users) can then decide if they still want to make use of a Web Service that does not match exactly the desired functionality. We call this mechanism the *matching algorithm* as it tries to match a Web Service advertisement against a service requirement.

1.1 Preliminary Definitions and Remarks

Before we get started, some basic definitions are to be clarified which will repeatedly occur throughout this work, while assuming that the reader is familiar with more general concepts, such as XML, HTTP [12] and SOAP.

1.1.1 Web Service

Simply put, a Web Service is a self-describing, self-contained, modular unit of application logic that provides some kind of business functionality to other applications through an Internet connection. Other applications can access Web Services through standardized web protocols and data formats, such as HTTP, SOAP and XML. A Web Service can be implemented in any possible programming language, and its implementation is not visible to the user or application that calls the service, since all the interaction happens through standard web protocols. Any program that can process XML and communicate via HTTP is essentially capable of calling a Web Service.

1.1.2 WSDL

WSDL is short for Web Service Description Language. A WSDL document is written in XML, and it basically describes a Web Service. It specifies the location of the service and the operations and methods the service exposes, and how to access them. An application that wants to call a remote Web Service has to examine its WSDL specification first in order to learn how that particular Web Service can be accessed, and how the response of the service can be understood, if necessary.

1.1.3 Semantic Web

The Web as it is known today provides its vast amount of information mostly in forms of human understandable web pages, i.e. plain text. As a consequence, the underlying structure of the data is not evident to computer programs or robots browsing the Web. The Semantic Web (SW) approach is to develop languages and mechanisms for expressing information in machine understandable form. These languages emerge in form of ontologies that will describe web resources easily processed by computer programs. Useful resources about the Semantic Web can be found at [24].

1.1.4 General Setup, Service Provider and Service Requester

The general setup in this work is that some entity needs to make use of an external Web Service in order to fulfill its task. Usually this entity is referred to as the client or agent, we will also call it the service requester. The service that the requester seeks to invoke is referred to as the requested service. The requested service thus is not an actual service but an ideal abstraction of a Web Service. The service requester is not limited to make use of at most one Web Service. Any entity that provides a Web Service is referred to as the service advertiser, or service provider. The Web Service itself which is offered by a service provider will be called the advertised service. If it happens that the advertised service is exactly what the requester was looking for, then the advertised service equals the requested service.

A Web Service that has any kind of semantic description will be referred to as a semantic Web Service.

1.2 Related Work

1.2.1 DAML-S Matchmaker

The DAML-S Matchmaker [26] was developed by the Intelligent Software Agents Group at Carnegie-Mellon University. The matchmaking system "[...] serves as a yellow pages of agent capabilities, matching service providers with service requestors based on agent capability descriptions [...]". Part of the matchmaking system is a database where service providers can register their Web Services via DAML-S descriptions through a Web interface[1]. The system then allows service requesters to upload their service requests (which also happen to be encoded in DAML-S), and the Matchmaker determines the connectivity of the requirements with the registered services in its database. The matchmaking algorithm works in two phases: Given the requested service and an advertised service, it first tries to match every input parameter of the advertised service against all input parameters of the requested service and then every output parameter of the requested service against all output parameter of the advertised service. More precisely, it matches the types associated with each input or output parameter. For each parameter (either input or output) there are several degrees of matching, depending on the semantic relationship between the parameters of the advertisement and the request. Based on these results a global matching result is determined. The matching algorithm developed in this work uses a somewhat similar scheme for creating matching results between services. In this approach however, we take a considerable bigger amount of information into account; besides the input and output types, we also consider their classification, evaluate service classifications and allow for customization through plug-ins.

[1] See Section 4.1 for a more detailed description of the central database approach.

1.2.2 A Software Framework for Matchmaking using Racer

This matchmaking approach was developed by the Information Management Group at the University of Manchester [22]. The described prototype of their matchmaking algorithm is also based on DAML-S descriptions. It uses the Racer Description Logics reasoner to compute semantic matches between services, and utilizes JADE as an agent platform. As in the DAML-S Matchmaker design, service requests in DAML-S are matched against service advertisements. However, this matching algorithm does not split the matching procedure into several parts; it rather tries to find a semantic match directly for the specified service profiles. As we shall see, a service profile is part of any DAML-S description and includes specifications for the input and output parameters. Therefore in this approach the profiles are treated as whole entities, and the components of a profile are not separable. Again, the matching result for the profile is associated with a certain degree of match.

1.3 Structure of this Work

The next chapter deals with semantics in general. An introduction to Description Logics is given, followed by a closer look at DAML+OIL, which is a Description Logics based markup language for describing web content. At this point I would like to note that the detailed level of semantics presented in this chapter is necessary as it provides a solid foundation of the general concepts involving Description Logics. This in turn allows us to understand the benefits that semantic information will bring to the determination of the connectivity between Web Services. However, the construction of clever or complex semantic structures is not part of this work.

After we examined DAML+OIL, we are now able to deal with DAML-S, as DAML-S is encoded in DAML+OIL. This is done in Chapter 3. The structure of DAML-S is examined in some detail which will enable us to define the semantic meaning of a Web Service. We will explore certain properties of the semantic service description that will play important roles in the matching algorithm.

In the following chapter we will take a look at the general framework involving semantic Web Services. Two main scenarios are presented in which the deployment of the matching algorithm can prove to be useful.
In Chapter 5 we finally encounter the matching algorithm. We will identify different matching levels which are based on the semantic foundation and see that the matching process basically consists of four independent parts, which are then grouped together to obtain the final matching result.
Chapter 6 then contains a description of the actual implementation of the matching algorithm and the necessary components. The matching algorithm and all components are programmed in Java. Various open source projects were used to realize the goal of this work and a brief description is given.
What follows in a simple example that demonstrates how the semantic markup of Web Services and the use matching algorithm can assist in achieving the goal of a higher degree of interoperability for services.
Chapter 8 ends this work with conclusions. Also, an outlook is given on possible further development of the matching algorithm.

Software Attached with this work is a CD-ROM, which includes all the developed software and corresponding documentation from this work. The root directory contains a file named *readme.txt*, which serves as a guideline for the use of this software.

Chapter 2

Semantics

Description Logics and DAML+OIL provide the semantic foundation for markup of web resources. First, Description Logics will be explained in some detail, which is a framework for semantic networks. We examine the basic design of a semantic network and more importantly, what conclusions we can draw from existing information. The possibility of automatically deducing new information is the distinguishing feature of matching mechanisms that are based on semantic information, in contrast to yellow-page like approaches, such as UDDI.

Next, the concept of an ontology is introduced, which is frequently used in the Semantic Web context. Web ontologies allow for the creation of arbitrary domain definitions that enable the unambiguous description of web content. A brief introduction to RDF is given, followed by a description of DAML+OIL, which is a XML-based language to define ontologies. DAML+OIL plays an important role in this work, as every DAML-S document is encoded in DAML+OIL.

The mathematical definitions for Description Logics as they are stated in section 2.1.2, 2.1.3 and 2.1.4 were taken directly from [3]. As for DAML+OIL, the complete specification can be found at [18].

2.1 Description Logics

Research in the field of knowledge representation and reasoning led to intelligent systems which have the ability to find implicit consequences of its explicitly represented knowledge. These systems are characterized as knowledge-based systems. One approach of these systems evolved into what we call today Description Logics (DL). Description Logics can be characterized by the following three properties:

- The basic syntactic building blocks are atomic concepts, atomic roles and individuals.
- Expressive power of the language is restricted by using a rather small set of constructors for building complex concepts and roles from existing concepts and roles.
- Implicit knowledge about concepts and individuals can automatically be found with reasoning techniques.

2.1.1 An Introduction to Description Logics

Description Logics developed out of semantic network and frame systems. As the name suggests, these systems are arranged in network structures. In general, the elements of a network are nodes and links. Typically nodes are used to characterize sets of classes of individuals, and links are used to characterize relationships among them. In Description Logics, nodes are mainly referred to as concepts.

Consider the small example network displayed in Figure 2.1, which is taken from [3]. This sample network represents knowledge about persons, parents etc. The link between the concepts `Woman` and `Person` states that a woman is a person. Such a relationship is often termed a "IS-A" relationship. This relationship defines a hierarchy over the concepts, i.e. `Person` is a more general concept than `Woman`. The more general concept is termed the superconcept, whereas the more specific concept is called the subconcept. The "IS-A" relationship also provides the basis for the inheritance of properties; when a concept is more specific than another concept, it inherits the

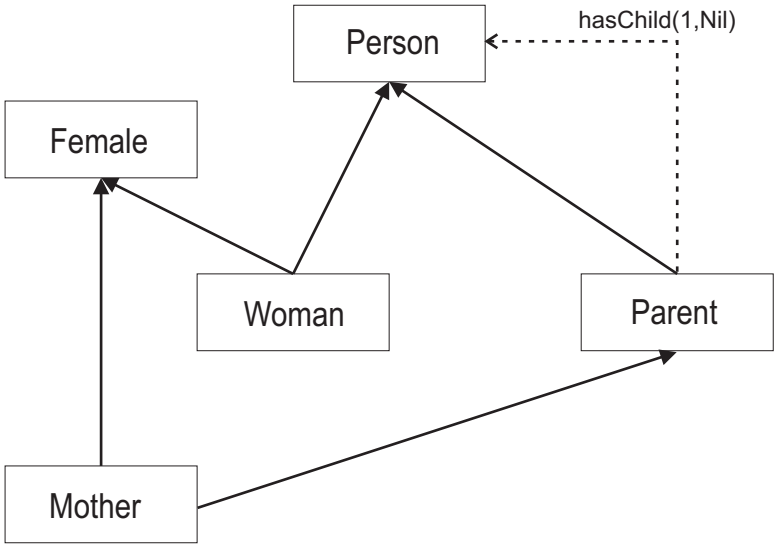

Figure 2.1: A small example network.

properties of the more general one. For example, if the concept `Person` has the property age then the concept `Woman` also has the property age. It is worth mentioning that a concept can have multiple superconcepts, regarding the example network we see that for example the concept `Woman` has two superconcepts, namely `Person` and `Female`.

From atomic concept definitions one can create new concepts by the means of constructors, such as union, intersection etc. The use of the intersection constructor and others is shown in Figure 2.2, where the new concept `Woman` is defined in terms of the atomic concepts `Person` and `Female`. The definition of a concept underlies two restrictions: a concept name can only be used once, and its definition has to be acyclic.

The ability to represent other relationships among concepts besides the "IS-A" relationship is a separating feature of Description Logics to other semantic systems. In our example, the dotted link from `Parent` to `Person` is a so-called *role* relationship. Roles are binary relationships between two concepts. Roles can have a value restriction, limiting the types of objects that can fill that role[1]. In addition, it can have a number restriction, where the first number denotes a lower bound and the

[1]The object in the range of the role is also referred to as the filler of that role.

second number an upper bound. In our example this expresses that a parent has at least one child. Role relationships are also inherited from more general concepts. The concept `Mother` thus inherits the role `hasChild` from the concept `Parent`.

The presented structure of concepts and their relationships is also referred to as the *terminology* of the problem domain, and it is indeed meant to represent the generality and specificity of the concepts involved.

Note that there are implicit relationships between concepts. In our example network, there is a "hidden" relationship between the concepts `Mother` and `Woman`. Since `Mother` is a subconcept of `Female` and `Parent`, and `Parent` itself is a subconcept of `Person`, the concept `Woman` is a superconcept of `Mother`, as the concept `Woman` is a subconcept of `Person` and `Female`. The ability of finding these implicit relationships is the characteristic feature of knowledge-base systems. Such reasoning tasks are also called inference mechanisms and will be discussed later on.

Intensional and Extensional Knowledge Within a knowledge-base there is a clear distinction between intensional knowledge, which is more general knowledge about the problem domain, and extensional knowledge, the more specific knowledge to a particular problem. In Description Logics, these two concepts are typically comprised by two components, the *TBox* and the *ABox*.
The TBox contains the intensional knowledge in the form of the terminology and is built through declarations that define general concepts and its properties. The representation of the concepts and relationships from the example network would comprise the TBox, as they represent general knowledge about the family domain. The ABox contains the extensional knowledge about the domain of interest. In the ABox, one introduces individuals, by giving them names, and one asserts properties of these individuals. An individual defined in the ABox is an instance of a concept from the terminology.

2.1.2 Description Logics Syntax

The basic building blocks are atomic concepts and atomic roles. In the following we denote atomic concepts with the letters A and B, and use R and S for atomic

roles. More complex concepts can be defined from atomic concepts and roles via constructors; we denote them by the letters C and D. Different description languages differ in which constructors they provide. The basic Description Logics family is the language $\mathcal{AL}$ (attributive language). The following are the constructors for the $\mathcal{AL}$ language:

$$
\begin{array}{rcll}
C, D & \longrightarrow & A \mid & \text{(atomic concept)} \\
 & & \top \mid & \text{(universal concept)} \\
 & & \bot \mid & \text{(bottom concept)} \\
 & & \neg A \mid & \text{(atomic negation)} \\
 & & C \sqcap D \mid & \text{(intersection)} \\
 & & \forall R.C \mid & \text{(value restriction)} \\
 & & \exists R.\top & \text{(limited existential quantification)}
\end{array}
$$

We note that in $\mathcal{AL}$ negation can only be applied to atomic concepts, and only the universal concept is allowed in the scope for the limited existential quantification. The description logics language $\mathcal{ALEN}$ is an $\mathcal{AL}$ extension allowing for full existential quantification and number restrictions. The language however that is closest to DAML+OIL in its expressive power includes negation on arbitrary (complex) concepts, cardinality restrictions, inverse roles, transitive roles and data-types. Additional constructors include:

$$
\begin{array}{rcll}
C, D & \longrightarrow & \exists R.C \mid & \text{(full existential quantification)} \\
 & & \neg C \mid & \text{(negation on arbitrary concepts)} \\
 & & \leq nR \mid & \text{(atmost cardinality restrictions)} \\
 & & \geq nR \mid & \text{(atleast cardinality restrictions)} \\
 & & = nR \mid & \text{(exact cardinality restrictions)} \\
 & & \leq nR.C \mid & \text{(qualified atmost cardinality restrictions)} \\
 & & \geq nR.C \mid & \text{(qualified atleast cardinality restrictions)} \\
 & & = nR.C \mid & \text{(qualified exact cardinality restrictions)} \\
 & & \leq_n R \mid & \text{(concrete domain max restrictions)} \\
 & & \geq_n R \mid & \text{(concrete domain min restrictions)} \\
 & & =_n R \mid & \text{(concrete domain exact restrictions)}
\end{array}
$$

2.1.3 Description Logics Semantics

Concepts and Roles We can now formally define the semantics of such languages. An *interpretation* $\mathcal{I}$ consists of a non-empty set $\Delta^{\mathcal{I}}$ (the domain of the interpretation) and an interpretation function which maps every atomic concept A onto a set $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ and every atomic role R onto a binary relation $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. For the $\mathcal{AL}$ language the interpretation function extends to concept descriptions in the following way:

$$\begin{aligned}
\top^{\mathcal{I}} &= \Delta^{\mathcal{I}} \\
\bot^{\mathcal{I}} &= \emptyset \\
(\neg A)^{\mathcal{I}} &= \Delta^{\mathcal{I}} \backslash A^{\mathcal{I}} \\
(C \sqcap D)^{\mathcal{I}} &= C^{\mathcal{I}} \cap D^{\mathcal{I}} \\
(\forall R.C)^{\mathcal{I}} &= \{a \in \Delta^{\mathcal{I}} \mid \forall b.(a,b) \in R^{\mathcal{I}} \rightarrow b \in C^{\mathcal{I}}\} \\
(\exists R.\top)^{\mathcal{I}} &= \{a \in \Delta^{\mathcal{I}} \mid \exists b.(a,b) \in R^{\mathcal{I}}\}
\end{aligned}$$

Interpretation for the additional constructors extends in a similar way, for example the full existential quantification yields

$$(\exists R.C)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} \mid \exists b.(a,b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}$$

and the atmost cardinality restriction yields

$$(\leq nR)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} \mid \#\{b \mid (a,b) \in R^{\mathcal{I}}\} \leq n\}$$

Similar definitions hold for the other constructors.

Terminological Axioms and Definitions *Terminological axioms* make statements about how concepts or roles are related to each other. Terminological axioms have the form

$$\begin{aligned}
C \sqsubseteq D \ (R \sqsubseteq S) &\quad \text{for inclusions and} \\
C \equiv D \ (R \equiv S) &\quad \text{for equalities.}
\end{aligned}$$

The semantics on axioms are defined quite straightforward. An interpretation $\mathcal{I}$ satisfies an inclusion $C \sqsubseteq D$ if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$, and it satisfies an equality $C \equiv D$ if

$$
\begin{array}{rcl}
\text{Woman} & \equiv & \text{Person} \sqcap \text{Female} \\
\text{Man} & \equiv & \text{Person} \sqcap \neg\text{Woman} \\
\text{Mother} & \equiv & \text{Woman} \sqcap \exists\text{hasChild.Person} \\
\text{Father} & \equiv & \text{Man} \sqcap \exists\text{hasChild.Person} \\
\text{Parent} & \equiv & \text{Father} \sqcup \text{Mother} \\
\text{Grandmother} & \equiv & \text{Mother} \sqcap \exists\text{hasChild.Parent} \\
\text{MotherWithManyChildren} & \equiv & \text{Mother} \sqcap \geq 3\ \text{hasChild} \\
\text{MotherWithoutDaughter} & \equiv & \text{Mother} \sqcap \forall\text{hasChild.}\neg\text{Woman} \\
\text{Wife} & \equiv & \text{Woman} \sqcap \exists\text{hasHusband.Man}
\end{array}
$$

Figure 2.2: TBox with concepts about the family domain.

$C^{\mathcal{I}} = D^{\mathcal{I}}$. Let $\mathcal{T}$ denote a set of axioms; then $\mathcal{I}$ satisfies $\mathcal{T}$ if and only if $\mathcal{I}$ satisfies each element of $\mathcal{T}$. If $\mathcal{I}$ satisfies an axiom we say that it is a *model* of this axiom.

A *definition* is an atomic concept of the left-hand side of an equality. Definitions are used to introduce symbolic names for complex descriptions. For example, with the axiom

$$\text{Mother} \equiv \text{Woman} \sqcap \exists\text{hasChild.Person}$$

we associate the name `Mother` to the right-hand side of the equality.

A finite set of definitions $\mathcal{T}$ is called a terminology or TBox. In a terminology no symbolic name is defined more than once. An example TBox with concept and role definitions about family relationships can be seen in Figure 2.2.

Assertions Let a, b and c denote names for individuals. There are two different kind of assertions for individuals:

$$
\begin{array}{ll}
C(a) & \text{Concept assertion} \\
R(a, c) & \text{Role assertion}
\end{array}
$$

The first assertion states that the individual a belongs to the interpretation of C, whereas the latter states that c is a filler of the role R for b. A finite set A of assertions is termed an ABox.

We define semantics for ABoxes by extending the interpretations to include individual names. An interpretation $\mathcal{I}$ now additionally maps each individual name a

MotherWithoutDaughter(ANNA)	Father(BOB)
hasChild(ANNA,BOB)	hasChild(BOB, CLARA)
hasChild(ANNA,PAUL)	

Figure 2.3: ABox with assertions about the family domain.

to an element $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$. The interpretation $\mathcal{I}$ satisfies the concept assertion $C(a)$ if $a^{\mathcal{I}} \in C^{\mathcal{I}}$, and it satisfies the role assertion $R(b, c)$ if $(b^{\mathcal{I}}, c^{\mathcal{I}}) \in R^{\mathcal{I}}$. An interpretation $\mathcal{I}$ satisfies an ABox $\mathcal{A}$ if it satisfies each assertion in $\mathcal{A}$. In this case $\mathcal{I}$ is a model for $\mathcal{A}$. An example ABox with some assertion corresponding to the TBox in Figure 2.2 is given in Figure 2.3.

An interpretation $\mathcal{I}$ satisfies an ABox $\mathcal{A}$ with respect to a TBox $\mathcal{T}$ if in addition to being a model for $\mathcal{A}$, it also is a model for $\mathcal{T}$.

2.1.4 Reasoning Tasks for Concepts and Individuals

Now that the concepts of the TBox and ABox are defined, we can use inference algorithms to deduce new knowledge about the domain of interest. Description Logics support inference patterns that occur in many applications of intelligent information processing systems, and which are also used by humans to structure and understand the world.

Reasoning on Concepts Let $\mathcal{T}$ be a TBox. There are four main reasoning tasks for concepts:

- *Subsumption.* A concept C is subsumed by D with respect to $\mathcal{T}$ if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ for every model $\mathcal{I}$ of $\mathcal{T}$.
- *Satisfiability.* A concept C is satisfiable with respect to $\mathcal{T}$ if there exists model $\mathcal{I}$ such that $C^{\mathcal{I}}$ is nonempty.
- *Equivalence.* Two concepts C and D are equivalent with respect to $\mathcal{T}$ if $C^{\mathcal{I}} = D^{\mathcal{I}}$ for every model $\mathcal{I}$ of $\mathcal{T}$.
- *Disjointness.* Two concepts C and D are disjoint with respect to $\mathcal{T}$ if $C^{\mathcal{I}} \cap D^{\mathcal{I}} = \emptyset$ for every model $\mathcal{I}$ of $\mathcal{T}$.

It can be shown that the latter three can be realized using subsumption.

Subsumption Out of these reasoning tasks, the subsumption pattern is the most important for the proposed matching algorithm. Subsumption can be seen as the determination of subconcept and superconcept relationships between concepts of a given terminology. It means deciding whether one description C is more general than another description D, i.e. if being D logically implies being C. The more general concept C is also called the *subsumer* and the more specific concept D the *subsumee*.

Subsumption allows one to structure the terminology in the form of a subsumption hierarchy. This hierarchy will then provide useful information on the relationships among the concepts of the terminology.

Considering the TBox displayed in Figure 2.2, the concept `Person` subsumes `Woman`, both `Woman` and `Parent` subsume `Mother`, and the concept `Grandmother` is subsumed by `Mother`. Moreover, the concept `Man` and `Woman` are disjoint, and also `Father` and `Mother` are disjoint.

Reasoning on Individuals Classification of individuals, or *instance checking*, determines whether an individual from the ABox is always an instance of a certain concept, i.e. whether the instance relationship is implied by the description of the individual and the concept definition in the TBox. Classification of individuals thus provides useful information on the properties of an asserted individual. Another important inference pattern is *consistency checking*; for example, if the ABox contains the assertions `Father(ANNA)` and `Mother(ANNA)`, the system should be able to find out that these two assertions are inconsistent with respect to the TBox in Figure 2.2. Other reasoning tasks for the ABox include *knowledge base consistency*, which amounts to verifying that every concept in the knowledge base admits at least one individual, *realization*, which determines the most specific concept an individual is an instance of, and *retrieval*, which finds the individuals in the knowledge base that are instances of a given concept. It can be shown that these additional reasoning tasks are equivalent to instance checking.

A description of these reasoning algorithms is beyond the scope of this work[2], and the interested reader is referred to [3].

2.1.5 Open World and Closed World Semantics

It is often the case that Description Logics or semantic networks in general are compared with databases. The TBox can be compared to the schema or metadata of a database and one can draw analogies between the ABox and the actual data that is stored in the database. However, the semantics of the ABoxes differ from the semantic interpretation of the instance of the database data. Absence of information in the database is interpreted as negative information, whereas absence of information in the ABox only indicates a lack of knowledge.

For example, if the only assertion about `ANNA` is `hasChild(ANNA,BOB)`, then in the database language this will be interpreted as the fact that `ANNA` has exactly one child, namely `BOB`. In the ABox however, this assertion only yields the conclusion that `BOB` is a child of `ANNA`, not that he is the only one or that he has any siblings.

Information in a database is thus viewed to be complete, whereas information in the ABox is in general viewed as being incomplete. The semantics of the ABox is therefore characterized as being *open-world*, while the semantics of a database is characterized as *closed-world*.

The different semantics have consequences in the way queries are answered. In the closed-world semantics of a database, queries amount to finite model checking, i.e. checking instance by instance, whereas queries to the knowledge of an ABox are more complex; they require nontrivial reasoning.

2.2 Ontologies

In the context of the Semantic Web, the term ontology occurs frequently. The term is borrowed from philosophy where an ontology denotes a systematic account of Existence. In the context of the Semantic Web, an ontology denotes a description of the

[2]Translation: the author doesn't understand them.

concepts and relationships that exist for a specific domain. Therefore, an ontology is nothing more than a terminology for a given domain of interest. These pre-defined ontologies allow computer agents and programs to unambiguously interpret the meaning of web resources.

One interesting aspect about ontologies is the fact that ontologies can refer to other ontologies. This will eventually lead to the creation of few domain-independent terminologies and many domain-dependent terminologies. The domain-independent ontologies describe very general concepts and relationships, whereas the domain-dependent ontologies build on top of the domain-independent ontologies and describe some concepts and relationships more in detail.

Any ontology that is defined for the use in the web is also referred to as a web ontology.

2.3 RDF

The Resource Description Framework (RDF) [5] was developed by the W3C, providing an XML-based language for modeling semi-structured metadata and enabling knowledge-management applications. RDF is very similar to a basic directed graph to represent its information. An RDF document contains its knowledge in form of triples, a so called subject-verb-object (SVO) form. These triples define the relationships between concepts. For example, the following triple in RDF (expressed in XML)

```
<Class ID="Woman">
    <subClassOf resource="#Person"/>
</Class>
```

states that the concept `Woman` is a subconcept of the concept `Person`. It is its simplicity that makes RDF easy to understand and a sort of assembly language on top of which almost every other information-modeling method can be overlaid. However, it became apparent that the lack of more sophisticated constructs, such as datatypes, enumerations, number restrictions etc, was limiting the use of RDF.

2.4 DAML+OIL

2.4.1 An Introduction to DAML+OIL

In response to RDF the DARPA Agent Markup Language (DAML) was developed by a U.S. government effort around August 2000. DAML builds on top of RDF, and a little later the Ontology Inference Layer (OIL) was introduced, yielding the DAML+OIL language. In essence, DAML+OIL is a Description Logics markup language for defining web ontologies.

It is noteworthy that in DAML+OIL the intensional knowledge and the extensional knowledge reside in the same document, i.e. the terminology and the instance data are not clearly syntactically separated within an ontology. To see how DAML+OIL is structured, we consider a concrete ontology that realizes the definitions displayed in Figure 2.4. This Figure presents a simple classification about computers. The DAML+OIL definition corresponding to Figure 2.4 is displayed in Listing 2.1.

The root element has to be `<rdf:RDF>`. Inside this element we declare all the necessary namespaces. The first element inside the `<rdf:RDF>` element is always `<daml:Ontology>`. Inside `<daml:Ontology>` we can state certain facts, such as the

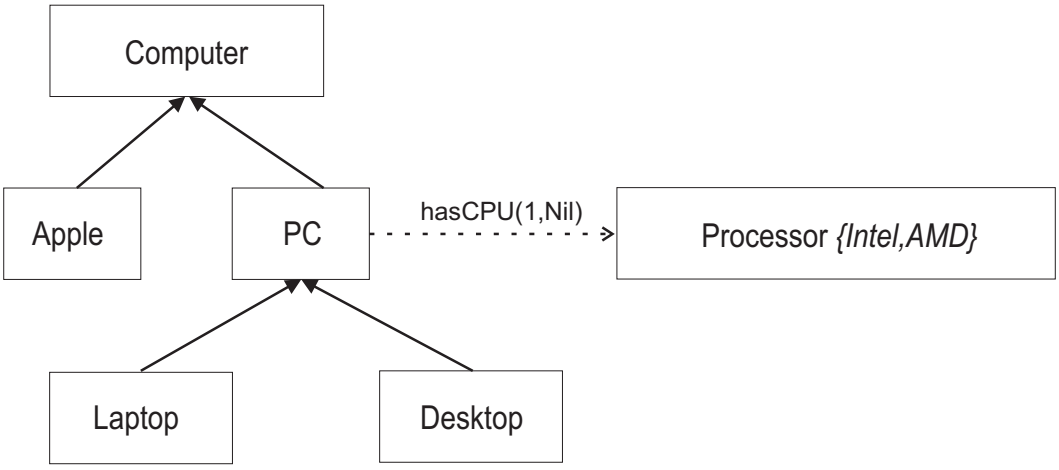

Figure 2.4: Ontology for the computer domain.

version, comments etc. The most important part however is the `<daml:imports>` statement. Any ontology can import other ontologies. The import process is transitive, that is if ontology A imports ontology B and B imports ontology C, then A also imports C.

After the `<daml:Ontology>` element, we can state as many class and property definitions as needed. The `<daml:Class>` element is used to define a concept. The attribute ID uniquely defines this concept. Note that there is no hierarchy within the class definitions, i.e. we do not have class elements inside other class elements. To express that a class is a subclass of another class, we use the `<rdfs:subClassOf>` element and refer to the ID of the parent class. In DAML+OIL it is perfectly fine that a class can have several superclasses. Multiple subclass statements are to be read conjunctively.

A property, or binary relation, connects two concepts/classes with one another. There are two different kinds of property definitions: A datatype property (via the element `<daml:DatatypeProperty>`) relates an object with datatype values (such as Strings, Integers etc.) and an object property (`<daml:ObjectProperty>`) relates two classes. The property has a domain and range specification. The domain specifies to which object that property applies, and the range defines which values the property can take. A property can have multiple ranges. Again, multiple range statements are read conjunctively.

DAML+OIL provides important facilities that give ontology designers more expressiveness in classifying resources. One example of an added feature for defining classes is the support for enumerations, a very common need in RDF design. An enumeration defines a class by giving an explicit list of its members. This is done with the help of the `<daml:oneOf>` and `<daml:collection>` elements. DAML+OIL allows one to state that a class is defined by being one of a given set of instances. In Listing 2.1, the class `Processor` is defined through an enumeration. Each item in the enumeration is defined as an instance of `<daml:Thing>`, which is a special DAML+OIL type and universally includes all instances of all classes.

The mapping of constructors and axioms in DAML+OIL which correspond to the Description Logics constructors and axioms are shown in Table 2.1 and Table 2.2.

Listing 2.1: DAML+OIL definition corresponding to Figure 2.4.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE uridef[
    <!ENTITY default "http://localhost:8080/Ontologies/pc.daml">
]>
<rdf:RDF xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:xsd="http://www.w3.org/2000/10/XMLSchema#">

    <daml:Ontology rdf:about="">
        <daml:versionInfo>v1.0, 22.02.2004</daml:versionInfo>
            <rdfs:comment>Ontology defining computer concepts</rdfs:comment>
    </daml:Ontology>
    <daml:Class rdf:ID="Processor">
        <daml:oneOf parseType="daml:collection">
            <daml:Thing rdf:about="&default;#Athlon"/>
            <daml:Thing rdf:about="&default;#Intel"/>
        </daml:oneOf>
    </daml:Class>
    <daml:Class rdf:ID"Computer"/>
    <daml:Class rdf:ID="PC">
        <rdfs:subClassOf rdf:resource="#Computer" />
    </daml:class>
    <daml:Class rdf:ID="Apple">
        <rdfs:subClassOf rdf:resource="#Computer" />
    </daml:class>
    <daml:Class rdf:ID="Laptop">
        <rdfs:subClassOf rdf:resource="#PC"/>
    </daml:Class>
    <daml:Class rdf:ID="Desktop">
        <rdfs:subClassOf rdf:resource="#PC"/>
    </daml:Class>
    <daml:ObjectProperty rdf:ID="hasCPU">
        <rdfs:domain rdf:resource="#PC"/>
        <rdfs:range rdf:resource="#Processor"/>
    </daml:ObjectProperty>
</rdf:RDF>
```

As mentioned before, extensional information is not separated from the terminology within the DAML+OIL document. Instance data can be defined anywhere within the document. We define instances by using element tags named after the concept which the individual shall be an instance of.

2.4.2 Tools for DAML+OIL

OilEd OilEd [4] is a graphical editor for web ontologies defined in DAML+OIL. It provides a convenient interface, with which the user can load existing or create new ontologies, and edit these. The interface allows the creation of classes and properties and new classes can be easily created by using constructors on previously defined classes and properties. The OilEd editor also comes with a built-in reasoner, so that the user can verify the consistency of his or her ontologies. Some of the ontologies defined for this work were created with the OilEd editor.

DAML+OIL Constructor	Description Logics Syntax
intersectionOf	$C_1 \sqcap \ldots \sqcap C_n$
unionOf	$C_1 \sqcup \ldots \sqcup C_n$
complementOf	$\neg C$
oneOf	$\{a_1, \ldots, a_n\}$
toClass	$\forall R.C$
hasClass	$\exists R.C$
hasValue	$\exists R.\{a\}$
minCardinalityQ	$\geq_n R.C$
maxCardinalityQ	$\leq_n R.C$
cardinalityQ	$=_n R.C$

Table 2.1: DAML+OIL constructors and corresponding DL syntax.

DAML+OIL Axiom	Description Logics Syntax
subClassOf	$C_1 \sqsubseteq C_2$
sameClassAs	$C_1 \equiv C_2$
subPropertyOf	$P_1 \sqsubseteq P_2$
samePropertyAs	$P_1 \equiv P_2$
disjointWith	$C_1 \sqsubseteq \neg C_2$
sameIndividualAs	$\{a_1\} \equiv \{a_2\}$
differentIndividualFrom	$\{a_1\} \sqsubseteq \neg\{a_2\}$
inverseOf	$P_1 \equiv P_2^-$
transitiveProperty	$P^+ \sqsubseteq P$
uniqueProperty	$\top \sqsubseteq \leq 1P$
unambiguousProperty	$\top \sqsubseteq \leq 1P^-$

Table 2.2: DAML+OIL axioms and corresponding DL syntax.

Chapter 3

DAML-S

The DARPA Agent Markup Language for Services (DAML-S) provides a semantic markup for Web Services. It was first developed by the DAML coalition [2] around May 2001. The latest release is DAML-S version 0.9, and a complete specification can be found at [8].

While existing languages such as WSDL provide low-level communication layers for Web Services, DAML-S was developed in order to provide a higher-level application layer which sits on top of these. As these low-level communication languages and protocols define *how* to access a Web Service, DAML-S aims at providing an answer as to *why* one uses a certain Web Service, and what this Web Service actually does.

In response to this motivation the DAML coalition identified four basic tasks, and hence DAML-S was developed to assist in realizing these. These tasks include:

1. Automatic Web Service discovery. This task involves the automatic location of a Web Service that adheres to some constraints. This means that the Web Service has to satisfy certain criteria. For example, one wishes to find Web Services that sell desktop computers and accept credit cards as payment.

2. Automatic Web Service invocation. Automatic Web Service invocation means, given that an appropriate service has been found, the autonomous execution of this service by a computer program or an agent without the need of any human interaction.

3. Automatic Web Service composition and interoperation. This task defines the automatic composition of several Web Services into a new Web Service. For example, an online store wants to provide a complete travel package to its customers. This Service might be composed of several individual Web Services, such as Hotel Booking, Flight Booking etc.

4. Automatic Web Service execution monitoring. Execution monitoring provides a way for a computer program or an agent to supervise the execution of individual services, or more importantly, the composition of several Web Services.

This work is concerned with the matching of Web Services. In essence this amounts to providing means with which a service requester can select the appropriate Web Service from a set of advertised services. The algorithms that will be developed thus fall into the first category, i.e. they will make use of the mechanisms DAML-S provides to enable and facilitate the automatic discovery and selection of Web Services.

3.1 The Service Ontology

DAML-S itself is an ontology describing the organization of a Web Service. There are four basic classes (or concepts), namely `Service`, `ServiceProfile`, `ServiceModel` and `ServiceGrounding`.

The class `Service` provides an organizational entry point for any Web Service description. This class has three properties: *presents*, *describedBy* and *supports*. The ranges of these properties are the classes `ServiceProfile`, `ServiceModel` and `ServiceGrounding`, respectively. Each instance of the class `Service` (that is any actual Web Service description) *presents* a descendant class of `ServiceProfile`, is *describedBy* a descendant class of `ServiceModel` and *supports* a descendant class of `ServiceGrounding`. The fact that the service presents a descendant class of `Profile`[1] (and not necessarily a direct instance of `Profile`) allows for profile classifications, as

[1] A service profile is actually presented through the class `Profile`, which is a direct subclass of the class `ServiceProfile`.

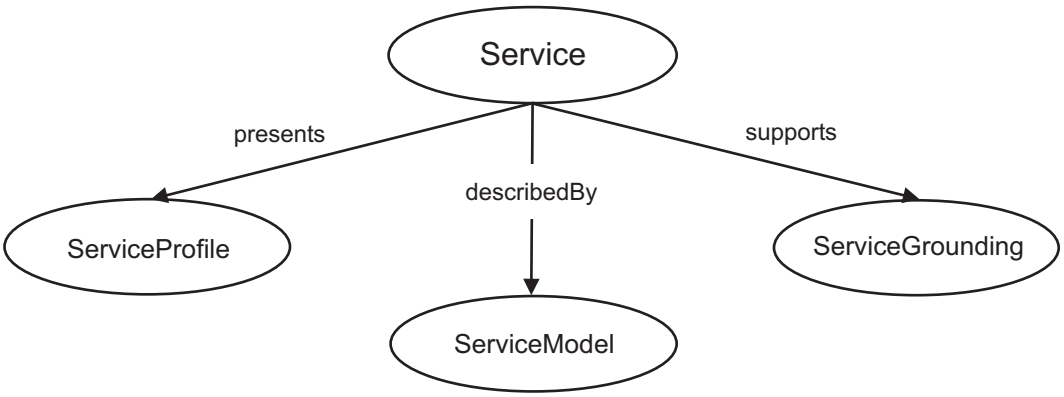

Figure 3.1: The service ontology.

explained below. As for the cardinality constraints of the properties, a service can have arbitrarily many profiles (including none), either one or none models and must have at least one grounding.

The `ServiceProfile` provides a precise description of the offered Web Service; it gives a detailed description of all the information necessary in order for a service-seeking agent to decide if that particular service meets its needs. Each method that a service is providing is described in a descendant class of `Profile`. If say, for example, a service wants to offer methods for searching and buying computers, the service has two profile descriptions, one for each method.

The `ServiceModel` describes how the service works, i.e. it explains what happens once the service is executed. It offers a detailed explanation of how the service is carried out. The exact functionality of the service is described in terms of a process model, or more specifically, by a *Process Ontology* and a *Process Control Ontology*.

The `ServiceGrounding` defines how the service can be accessed by other computer programs or agents. Typically the grounding will specify which protocols are used, which messages are exchanged etc.

As we will focus on the automatic Web Service discovery and selection as mentioned above, our main interest lies in the profile, which provides all the necessary information for the matching algorithm.

Note that exactly one instance of `Service` will exist for each distinct published Web Service. The top level of the service ontology can be seen in Figure 3.1.

3.2 The Profile

The profile provides three basic types of information. First of all, the class `Profile` presents contact information, which refers to the entity that provides that particular service, and which is mainly intended for human users. This information is provided via the following properties and is usually ignored when the service profile is automatically processed, but it may prove to be useful if the automatic discovery fails:

- `serviceName` is the name of the service offered. A service can have at most one service name.
- `textDescription` provides a short description of the service. Any additional information that the provider of the service wants to share fits in here. A service can have at most one text description.
- `contactInformation` specifies a person or entity that serves as a contact reference to the user of the service. A service can have multiple contact information, and each instance is of type `Actor`.

The next information provided is a functional description of the service. This functional description is expressed in terms of the input parameters required by the service and the output parameters generated by the service. Besides the input and output parameters, the functional description is described by two sets of conditions, namely preconditions, which have to hold before the service can be executed properly, and effects, i.e. conditions that hold after the successful execution of the service. For instance, if we were to call a Web Service in order to buy a computer via credit card, a precondition would be that the credit card is valid, and an effect of the execution of the service would be that the credit card is charged a specific amount (namely the price of the computer). These four functional descriptions are also referred to as IOPE (input output precondition effect).

The IOPEs are realized through an object property called `parameter`. This property assigns a class of type `ParameterDescription` to the profile. The following four object properties are subproperties of parameter, and describe the functional behavior of the Web Service:

- `input`. Specifies one input parameter of the profile. A profile can have as many inputs as required (including none).
- `output`. Specifies one output parameter of the profile. A profile can have as many outputs as required (including none).
- `precondition`. Specifies one precondition of the profile. There can be as many preconditions as necessary.
- `effect`. Specifies one effect of the profile. A profile can have as many effects as required.

Finally, the profile comes with a bunch of additional properties that are used to describe features of the service. These include:

- `serviceParameter`. A list of properties that may accompany the service profile. Each service parameter is an instance of the class `ServiceParameter`.
- `serviceCategory` is used to classify the service with respect to some ontology or taxonomy of services. The value of the property is an instance of the class `ServiceCategory`.
- `qualityRating` specifies the rating of the service according to some rating system. The quality rating is an instance of the class `QualityRating`.

3.2.1 Additional Classes in the Profile

Other classes that were referred to include the classes `Actor`, `ServiceParameter`, `QualityRating` and `ServiceCategory`. A detailed description is found at [8]. The following class however is worth mentioning, as it concerns the IOPEs directly.

Class ParameterDescription The class `ParameterDescription` provides values to inputs and outputs, and as well to preconditions and effects. It possesses the following properties:

- `parameterName` defines the name of the parameter that is described.

- `restrictedTo`. This attribute provides a restriction on the values that this parameter can take on.

- `refersTo`. A reference to the corresponding parameter in the process model.

3.3 Classification

3.3.1 Profile Classification...

As mentioned earlier, the entity that describes a specific method or operation of a Web Service is an instance of the class `Profile`. However, it does not need to be a direct instance of the class `Profile`. Moreover, it can be an instance of any class that is a subclass of `Profile`. The instance can be a direct subclass of `Profile`, or it might be an implicit instance, i.e. a subclass that has to be found via subsumption reasoning. The fact that indirect instances of the class `Profile` are possible allows for what we will refer to as profile classification. It is now possible to create ontologies in DAML+OIL which define a service hierarchy. For example, if we are to provide a service that sells computers, we would want to declare the corresponding profile as an eBusiness service, more specifically as a selling eBusiness service which is focused on computers. Any profile that advertises a specific functionality of a Web Service can thus be classified into a certain category. Profile classification might be one of the most interesting aspects when describing a Web Service, since important facts about the functionality of a Web Service can be gathered from its profile, more specifically from the information into which service category the profile falls.

Note that the profile classification is different to the `serviceCategory` attribute of the profile. The `serviceCategory` attribute points to classified taxomonies outside DAML-S, and thus the semantic reasoning capabilities are lost. However, it allows to make use of taxanomies which existed before the introduction of DAML+OIL or DAML-S.

3.3.2 ...and Parameter Classification

Not only does the profile itself allow for classification through subclassing, but one can also classify the IOPEs. This is achieved by defining any IOPE parameter as a subproperty of the corresponding IOPE. For example, we could define the property `hasOrderQty` as a subproperty of `input`. Web Services that sell certain items can now define the `hasOrderQty` as an input parameter, and agents examining this Web Service will know that this input parameter stands for the number of items one will order (as both the Web Service and the agent would make use of domain-independent ontologies, which include the definition of `hasOrderQty`). Although not mandatory, classification of IOPEs is strongly advised as it exposes a better view of the functionality of the Web Service.

3.4 The Profile for Requesting Services

Not only does the class `Profile` serve as a tool for describing advertised Web Services, it can also be used to describe requested services. That is, one can describe an ideal service by a profile. An ideal service is a Web Service which represents exactly the desired functionality that is needed by the service requester. Therefore, a service requester can create a profile of his ideal service, and try to match this profile against the profiles of the advertised services.

3.5 The Service Model and Grounding

The service model gives a detailed view of the Web Service; in fact it presents the service as a process. It provides complementary information to the profile, and actually describes how the service works once it is carried out. Whereas the service profile and service model are both abstract descriptions of the service, the service grounding in contrary is a concrete specification of how the service can be accessed. It provides details such as which protocols and message formats are used in order to communicate with the service. In particular, the DAML-S class `ServiceGrounding` builds on

top of WSDL. Simply said, the grounding defines how the abstract definitions of the inputs, outputs etc are to be realized as concrete messages in WSDL and used to call the Web Service.

Both the service model and service grounding thus provide the necessary information for a software program or agent to make use of the service once it is discovered and declared to be useful. All the information for the discovery of a Web Service however is provided by the Profile, and therefore we do only consider the class `Profile` in the matching algorithm.

Note that there are no constraints between the profile and the model, i.e. the model is decoupled from the profile. This means that the profile and model can be inconsistent. In this unlikely scenario, the matching algorithm would prove to be useless.

Chapter 4

Scenarios

This chapter will present different scenarios in which the application of semantic information to Web Services might prove to be a useful addition to existing infrastructure components. There are two main scenarios for the use of the matching algorithm. These two scenarios distinguish themselves mainly by the fact as to where the execution of the matching algorithm takes place. In the first scenario, the matching algorithm is controlled and executed by some central authority, and any service seeking user makes use of the matching service offered by this authority. In the second scenario, the matching algorithm will be executed locally. In addition to these two scenarios, a mechanism has been developed that allows for the connection between the latter scenario and existing infrastructure components such as UDDI, and thus broadening the range of applications in which the matching algorithm can be deployed successfully.

4.1 Scenario 1 - Server-sided Execution

In this scenario a central authority offers a server which provides an interface to the matching algorithm and maintains a semantic database or repository where service providers can register their Web Services. This is similar to the UDDI setup in which the UDDI Business Registry serves as a central database providing Web Service information. In contrast to UDDI, however, with a semantic database or repository

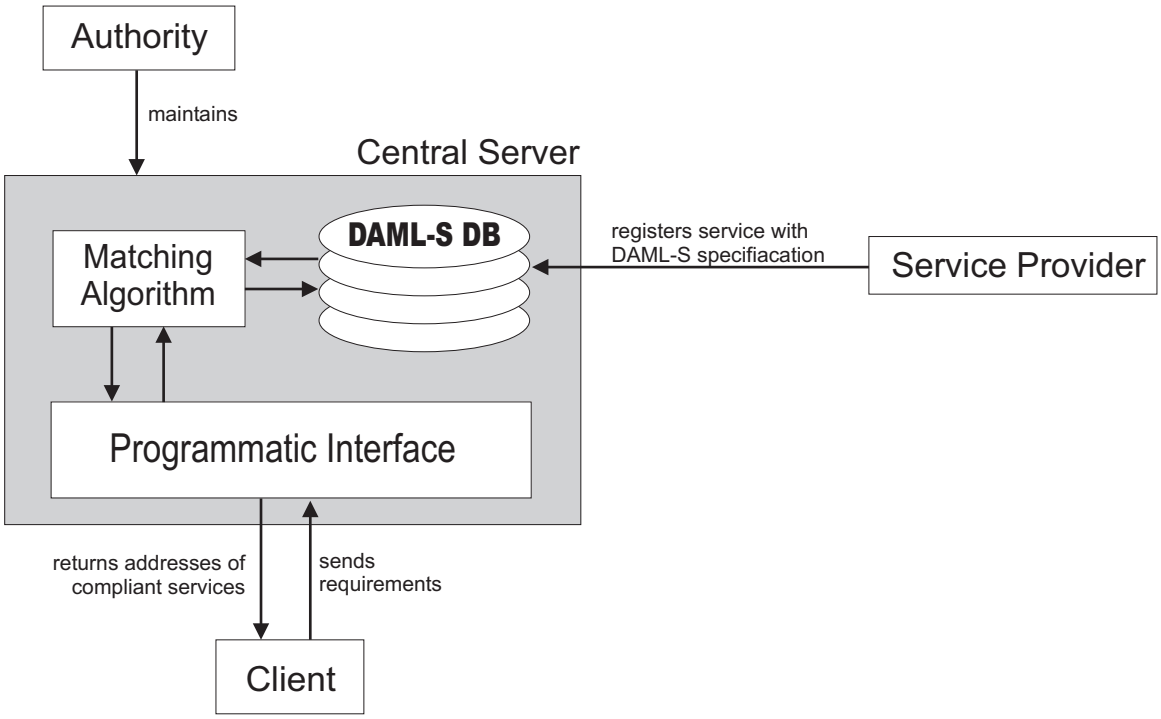

Figure 4.1: Scenario involving a central server.

service providers do register their services with corresponding DAML-S descriptions (and possibly additional "regular" information which they would provide if registering with the UDDI registry). A client seeking a Web Service now connects to the server through some interface and sends its requirements; the server then matches the obtained information against all the available Web Services in its database or repository. When a match is found, or several for that matter, the server returns the addresses of the appropriate Web Services.

This scenario is illustrated in Figure 4.1.

4.2 Scenario 2 - Local Execution

In this scenario the execution of the matching algorithm takes place locally. This means that the client seeking service is responsible for implementing and maintaining

the matching algorithm. The client naturally knows the semantic description of his ideal service, but on the other hand he has to somehow obtain semantic descriptions (i.e. DAML-S documents) of other Web Services, in order to match these against his very own requirements. There are basically two possible ways how the client can obtain these semantic descriptions:

- The client manually obtained the description, for example from a business partner.
- The client queried a database or repository which provides semantic information on Web Services. Unlike in scenario 1, this database does not perform any matching but simply provides semantic descriptions for Web Services.

In general, both described ways of obtaining the service descriptions may not be too practicable. The downfall of the first method is that it requires a certain level of manual interaction. The second method in contrast allows for automatic processing (i.e. querying the database) but in the worst case the client has to query the complete database and run the matching algorithm over every single Web Service description to finally find a match.

4.3 Extending Scenario 2

To overcome the shortcoming of scenario 2, a mechanism has been developed in this work that will enable the connection of the latter scenario with existing business registries such as UDDI. Although this mechanism is not limited to UDDI, a scenario using UDDI is described as a sample setup. In fact, any business registry for Web Services that allows computer programs to automatically connect and browse their databases is well-suited. UDDI however is already a highly accepted and well-known technology so that we will consider UDDI in the following.

4.3.1 Automatic Location of DAML-S Descriptions

This mechanism serves as a purpose for automatically finding the semantic description of a Web Service, i.e. its DAML-S specification, if available. It works in the same

fashion as when retrieving WSDL documents. If the address of a Web Service is known, then by adding *?wsdl* to the address will return its WSDL specification. Say, for example, a Web Service is located at *http://myServer.com/myService*, then the WSDL document can be retrieved at *http://myServer.com/myService?wsdl*.

During this work a simple server plug-in was developed, which processes requests for DAML-S documents. Now, with the suffix *?damls*, users or agents can locate DAML-S specifications. Thus, considering the example above, the semantic description of the service could be retrieved at *http://myServer.com/myService?damls*. This server plug-in works with the Apache Axis Implementation [27], and is described in more detail in Chapter 6.

Note that although the mechanism of discovering WSDL and DAML-S documents works in a similar way, they are fundamentally different in the sense that for any Web Service its WSDL document can be automatically generated by the server, whereas the DAML-S document is manually generated and deployed by the service provider. This difference arises because a WSDL specification is a "simple" definition of the Web Service's parameters and how to access it, whereas a DAML-S specification defines a meaning to a Web Service, which the server does not know, obviously. Thus, the two mechanisms look similar from the user/client point of view, but the provider has to put an additional effort into creating and deploying a semantic description of a Web Service. Figure 4.2 illustrates the two mechanisms.

4.3.2 Limitations of UDDI

The Universal Description, Discovery and Integration (UDDI) scheme is a business registry which enables businesses to quickly, easy and dynamically locate each other and transact with one another. Businesses register their services with UDDI, and the UDDI Business Registry uses standard industry taxonomies, or classification schemes, to categorize businesses, services, and service types. Each business entity or service can be associated with a variety of identifiers and categories, enabling users to search the registry by industry, product category, and geographical location. The UDDI Business Registry offers both a web-based user interface and a programmatic interface.

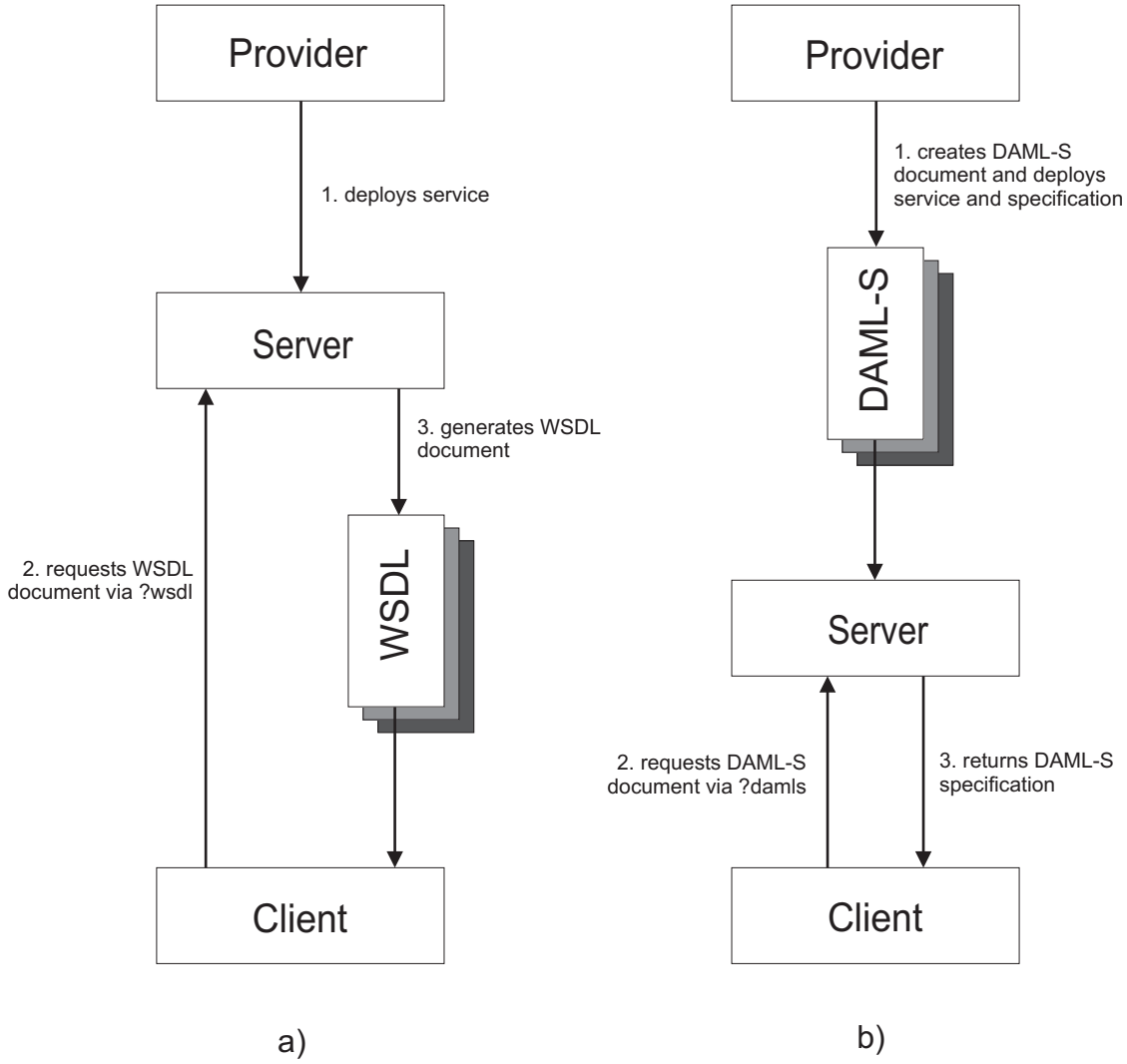

Figure 4.2: Document retrievals: a) *?wsdl* and b) *?daml-s*.

Typically, if a business registers a Web Service, besides the usual information it additionally stores a WSDL description of the service (or a reference to the WSDL document). This WSDL specification then enables the user to easily connect to the Web Service. More importantly, this specification allows for the automatic connection and execution of a Web Service.

As of now, the UDDI Business Registry does not offer any support for semantic information of Web Services. Although computer programs can autonomously browse

the registry through the programmatic interface, they have only limited possibilities in determining whether a Web Service fits their needs or not since they do not have access to possibly available semantic information. In the following section we take a look at the mechanism that allows users or agents to circumvent this limitation.

4.3.3 Connecting UDDI and Local Matching

With the help of automatic discovery of DAML-S documents, we now have a scenario where an agent can make use of existing technologies such as UDDI and still benefit from semantic information provided by DAML-S. This setup is illustrated in Figure 4.3. A service provider registers his Web Service as usual with the UDDI Business Registry (step 1). An agent then queries the UDDI database for Web Services which fall into a certain category (2). The UDDI server then returns all entries of Web Services which fall into this category (3). Besides other information, these entries include the addresses of the Web Services. They do not, however, include any semantic information, but now the agent can obtain the DAML-S documents from the server addresses by querying the servers via *?damls* (4). These DAML-S documents then will

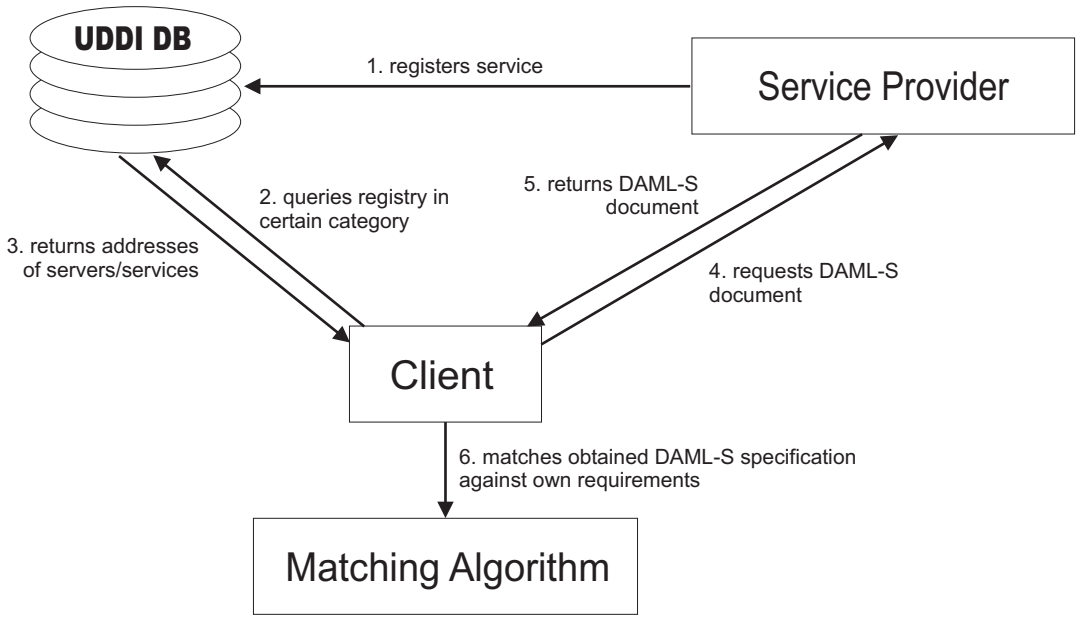

Figure 4.3: Scenario involving UDDI.

be matched against the agents requirements with the proposed matching algorithm (6), and the agent can decide which Web Service fits best (if any fits at all).

Consider for example a client who wants to make use of a specific information service (e.g. stock quotes, weather updates etc...). The client queries the UDDI Business Registry for services in the (fictitious) category "Information Services" and receives all the addresses of registered services in that category. Popular services such as getting stock quotes might have their own category, but many services will fall into a more general category. The client however can still determine the right service by matching the semantic information that he now obtains directly from the addresses the UDDI Business Registry returned.

4.4 Comparing the Two Approaches

The fundamental difference between these two approaches is the execution of the matching algorithm. In the first scenario, the matching algorithm is executed on the server side. The client or agent sends his requirements to the central server/database, and the server matches these requirements against the services that are registered in its database. The server then returns the addresses or information about the compliant Web Services. In the latter scenario, the client or agent obtains the DAML-S specifications, then locally runs the matching algorithm against his requirements and eventually decides which Web Service to call.

The advantage of the first approach lies in the fact that the client or agent has less effort to spend, as the server performs all the lookup of services and the actual matching. The central server scenario offers an easy and convenient way of finding complaint Web Services; a lot of work and time can be saved by using existing infrastructure components and mechanisms without having to worry about own implementation and maintaining issues.

A disadvantage however is that the client loses any control on how the matching is performed. As we will see in the next chapter, the proposed matching algorithm is to some degree extendable by self-defined plug-ins. In addition to the loss of flexibility, the user has to trust the authority that his central server actually offers the desired

functionality. Different servers might implement different matching algorithms, and there might be cases in which the internal functionality of this particular matching service is not publicly available. This can lead to unsatisfying results. A client might not get a perfect match on a Web Service from a central server although there might exist a Web Service which would perfectly fulfill all the requirements posed by the client program, simply because the executed matching algorithm does not work in a desired fashion. This could partially be resolved if the server allows the client to choose from multiple matching algorithms (if available) or to control the behavior of the matching algorithm with parameters, if possible.

This will not happen in the second approach. The advantage of locally executing any matching algorithm lies in the fact that the client now can choose which matching algorithm to use and how this mechanism is implemented. In the cases where the behavior of the matching service can be controlled by parameters (or plug-ins as is the case with this matching algorithm), the client can customize its functionality to some extend. This may give more accurate and desired results than in the first case.

However, the downside of the latter approach is obviously that the implementing entity has to do more work. It might require a lot of work to implement a specific matching algorithm into its very own applications, and then connecting this application to existing registries such as UDDI takes another effort. At last, maintaining costs and time is not to be forgotten in cases when existing infrastructure components or implemented software change with time. For example, existing business registries could go out of business, change their connectivity behavior, or if the client uses a matching algorithm from a third-party developer, the development might be discontinued; the functionality could change etc.

Chapter 5

The Matching Algorithm

In this chapter we examine the matching algorithm, which determines the connectivity between a service request and a service advertisement. More specifically, the matching algorithm matches exactly one profile of an advertised service against a profile of a service request. As it is not always the case that a perfect match to a service request exists, the result of the matching algorithm will contain different degrees of matching. These degrees of matching will define how good the functionality of any advertised service fits to that of a requested service. Clients that use the matching algorithm (usually autonomous software agents or human users) can then specify the minimal expected degree of matching, and thus check if an advertised service meets the matching requirements for their service request.

Although a service can have multiple profiles, we now use the words service and profile interchangeably, denoting one specific profile definition. If we were to match a whole advertised service, i.e. all of its profiles, against a service request, we would have to call the matching algorithm for each advertised service profile (along with the profile of the requested service).

The matching algorithm itself is not concerned with how the two service profiles are obtained, we now assume that we are given two service profiles and a set of respective

ontologies[1]. In the following the advertised service denotes a service offered by a provider, and the requested service denotes an ideal abstraction of the service that the requester is looking for.

5.1 Four Stages of Service Matching

The algorithm has four stages of matching. Each stage is independent of the other three, and the final matching result will be based on the individual results of these four stages. The four stages include input parameter matching, output parameter matching, profile matching and user-defined matching.

5.1.1 Reasons for a Splitted Algorithm

One may wonder why this algorithm splits the matching procedure into several parts. As we learned from the DAML-S specification, a Web Service is semantically described by an instance of the class `Profile`. This instance possesses all the inputs and outputs associated with that service as object properties; it is therefore possible to find a relation between two services by directly finding the semantic relationship between the instances of the respective `Profile` classes, which would include all the existing relationships between the input and output parameters, as these are properties of the profile.

An advantage of a non-splitted version is that all the information that is part of the class Profile would be automatically evaluated if specified; for example, information about the quality of service is evaluated if specified by both profiles via the class `QualityRating`.
DAML-S allows for a very detailed description of a Web Service. It is this amount of information however that counters the usability of a non-splitted matching approach. It might easily occur that two profiles will be declared as non-compatible (i.e. no semantic relation could be determined) because one (probably less important) property

[1] In practice only the two profiles need to be provided, as all necessary ontologies are imported by the DAML-S service specifications.

in a profile stands in contrast to a property in the other profile. One might argue that a failure of the matching algorithm is the desired result in this case, but there are many possible scenarios where such a mismatch is tolerable. This will become clear when we examine the different parts of the matching structure. It is also likely that with a non-splitted algorithm many matching attempts will result in failures as the DAML-S specifications can be arbitrarily complex. By splitting up the profile and determining the matches individually for the subparts, we expect a higher connectivity rate between service requests and advertisements.

The disadvantage that parts of the profile are ignored can be compensated with the proper usage of user-defined plug-ins. For example, a plug-in that evaluates the quality of service information could easily be added to the matching algorithm.

5.2 Concept and Property Matching

Most parts of the matching algorithm will build on the semantics of the services. The semantics itself are described by ontologies, as stated earlier on. As we have seen these ontologies are written in DAML+OIL. Our first task is thus classifying the different relations between concepts and properties when comparing two concepts or two properties.

5.2.1 Concept Matching

Let A, B denote two concepts which originate from a given set of ontologies. We then have the three following relationships between A and B:

1. *Equivalent(A,B).* A and B are equivalent, meaning that both denote exactly the same concept.

2. *Subsumes(A,B).* The concept B is subsumed by the concept A, meaning that A denotes a more general concept than B.

3. *Fail(A,B).* The concepts A and B are in no relation with each other with respect to all referenced ontologies.

The matching algorithm will provide a function called `conceptMatch`, which determines the above mentioned degrees for two given concepts. Note that in addition it also returns a result in the case that the concept B subsumes the concept A, we will refer to this degree as an *inverted subsumption*. Later on we will make use of both subsumption relationships for concepts. The function `conceptMatch` queries the underlying knowledge base to determine the relationship between the concepts. We will see in the next chapter how querying the knowledge base works. We note that for checking the equivalence of two concepts, it does not suffice to check if `conceptA==conceptB`. Two concepts can be equal even if they have different names and are defined in two different ontologies, as it is possible to equalize two concepts with usage of the `<daml:sameClassAs>` element.

5.2.2 Property Matching

Let R and S denote two properties which originate from a given set of ontologies. We then have the following relationships between R and S:

1. *Equivalent(R,S).* R and S are equivalent which means that both denote exactly the same concept.

2. *Subproperty(R,S).* The property R is a subproperty of the property S.

3. *Fail(R,S).* The two properties R and S are in no relation to one another with respect to all referenced ontologies.

The matching algorithm will provide a function called `propertyMatch`, which determines the above mentioned degrees for two given properties. The function also queries the knowledge base. As with concepts, two different properties can be equalized with the use of the tag element `<daml:samePropertyAs>`, and therefore we need to query the knowledge base to find out if two properties are equivalent.

5.3 Input Parameter Matching

With input parameter matching we try to determine how good the inputs of the advertised service correspond to the inputs of the requested service. During the input parameter matching, for each input of the advertised service the algorithm tries to find a match with an input of the requested service. Every input of the advertised service has to be matched if we are to make proper use of that service, as we can not expect to call a remote function properly if we can not provide a meaningful value for at least one of the input arguments. Since any input itself is a property, we have two possibilities of finding a match, property matches and type matches.

First, if the inputs are classified (see Section 3.3.2), we can determine the relation between the input of the requested and advertised service: Either both of the inputs denote the same property, one input is a subproperty of the other or both inputs are unrelated. Note that when both inputs are unclassified, they denote the same property (i.e. `input`). We define four different degrees of a property-match:

- *Equivalent.* Both inputs denote the same property.
- *Subproperty.* The input of the requested service is a subproperty of the input of the advertised service.
- *Unclassified.* Either the input of the requested service or of the advertised service is not classified. We mention this case explicitly as a false match might occur if one of the inputs is not classified. In general, it is not always possible to determine the meaning of an input parameter by just looking at the specified type in the range of the input parameter. For example, consider the case when a service method that sells computers provides an unclassified input parameter that expects an instance of the type price (defined in some general ontology). But it is unclear if this input now denotes a retailer price or an end consumer price.
- *Fail.* If none of the above conditions hold.

Next, we are able to match the types that are specified in the ranges of each input.

Rank	property-match result	type-match result
0	Fail	Any
	Any	Fail
1	Unclassified	Invert Subsumes
2		Subsumes
3		Equivalent
4	Subproperty	Invert Subsumes
5		Subsumes
6		Equivalent
7	Equivalent	Invert Subsumes
8		Subsumes
9		Equivalent

Table 5.1: Rankings for the matching of two parameters.

As these types can be concepts, we can apply subsumption reasoning. We define four different degrees of a type-match:

- *Equivalent.* Both types denote the same concept.
- *Subsumes.* The type of the input parameter of the advertised service is subsumed by the type of the input parameter of the requested service. This means that the advertised service would obtain an instance of a more general type for an input than expected. Chapter 7 will provide more insight as to why this degree makes sense.
- *Invert Subsumes.* The type of the input parameter of the advertised service subsumes the type of the input parameter of the requested service.
- *Fail.* If none of the above conditions hold.

Given these two types of matching, we can define a ranking for two inputs. These rankings are shown in Table 5.1. The higher the rank, the better the two input parameters match. We note that the result of the property-match is given a higher priority than that of the type-match, as a classification of an input gives more insight to the purpose of the input parameter than its type definition.

Rank	Degree of match	Explanation
0	FAIL	There is at least one input of the advertised service for which there exists no matching input of the requested service, i.e. for one input the rank is zero.
1	UNCLASSIFIED	Every input of the advertised service has been matched with an input of the requested service. However, for at least one input pair the property-match degree is *Unclassified*, i.e. the rank for one matched input pair is one, two or three.
2	SUBPROPERTY	Every input of the advertised service could be matched with one input of the requested service. For every input pair the property-match degree is either *Equivalent* or *Subproperty*, but we have at least one input pair matched with degree *Subproperty*, and the type-match degree is *Subsumes*, *Invert Subsumes* or *Equivalent*, i.e. the rank for one input pair is four, five or six.
3	TYPE_INVERT	Every input of the advertised service could be matched with one input of the requested service. For every input pair the property-match degree is *Equivalent* but at least for one input pair the type-match degree is *Invert Subsumes*, i.e. the rank for one input pair is seven.
4	TYPE_SUBSUMES	Every input of the advertised service could be matched with one input of the requested service. For every input pair the property-match degree is *Equivalent* but at least for one input pair the type-match degree is *Subsumes*, i.e. the rank for one input pair is eight.
5	MATCH	Every input of the advertised service could be matched exactly with one input parameter of the requested service. For every input pair the property-match degree and type-match degree is *Equivalent*, i.e. all input pairs have degree nine.

Table 5.2: Matching degrees for input parameter matching.

The matching algorithm will provide a function called `rankForParameters` which determines the rank of two parameters according to Table 5.1 and which calls the functions `propertyMatch` and `conceptMatch`.

Listing 5.1: Input parameter matching algorithm.

```
public int match(Vector reqInputsList, Vector advInputsList,
                 Reasoner reasoner) {
  if (advInputsList==null) {
    return MATCH;
  }
  int minOverallRank = 9;
  for (int i=0; i<advInputsList.size(); i++) {
    Input advInput = (Input)advInputsList.elementAt(i);
    Input bestMatch = null;
    int maxRank = 0;
    if (reqInputsList!=null) {
      for (int j = 0; j < reqInputsList.size(); j++) {
        Input tempReqInput = (Input) reqInputsList.elementAt(j);
        int rank = reasoner.rankForParameters(tempReqInput, advInput);
        if (rank > maxRank) {
          maxRank = rank;
          bestMatch = tempReqInput;
        }
      }
    }
    if (bestMatch==null) {
      return FAIL;
    }
    if (maxRank<minOverallRank) {
      minOverallRank = maxRank;
    }
  }
  if (minOverallRank==1 || minOverallRank==2 || minOverallRank==3) {
    return UNCLASSIFIED;
  } else if (minOverallRank==4 || minOverallRank==5 || minOverallRank==6) {
    return SUBPROPERTY;
  } else if (minOverallRank==7) {
    return TYPE_INVERT;
  } else if (minOverallRank==8) {
    return TYPE_SUBSUME;
  } else {
    return MATCH;
  }
}
```

Table 5.2 lists the results of the input parameter matching based on the results for matched input pairs. The matching algorithm always assumes the worst case scenario: if all inputs for the advertised service have been matched, the input pair with the lowest rank defines a specific matching result. For example, if one input pair has rank 1, 2 or 3, then the input parameter matching result can not exceed *UNCLASSIFIED*, even if all other input pairs have a higher rank.

The input parameter matching then operates as follows: for each input of the advertised service, it tries to find the input parameter of the requested service that has the highest rank if combined with the input of the advertised service. A match is found if the highest rank is greater than zero. The input of the advertised service is then said to form a (matched) input pair with the input of the requested service that has the highest rank greater than zero.

Listing 5.1 shows the input parameter matching algorithm in Java. The two vectors `reqInputsList` and `advInputsList` denote lists of the input parameters of the requested and advertised service, respectively. The input parameter matching makes use of the function `rankForParameters` through the class `Reasoner`, which will be explained in the next chapter.

The input parameter matching returns *MATCH* in case the list `advInputsList` is empty, as no useful input matching can be performed when the advertised service does not take any input arguments.

5.4 Output Parameter Matching

With output parameter matching we try to determine how good the outputs of the advertised service correspond to the outputs of the requested service. During the output parameter matching, the algorithm tries to find a match for each output of the requested service. For the outputs we again use the property-match and type-match degrees to find a relationship between two output parameters. However, now the roles are reversed, meaning that a *Subproperty* degree is recognized when the output of the advertised service is a subproperty of the output of the requested service and a *Subsumes* degree is found whenever the type of the output parameter of the requested

Rank	Degree of match	Explanation
0	FAIL	Every output parameter of the requested service could not be matched.
1	PARTIAL_FAIL	At least one output of the requested service remains unmatched, but there is at least one output pair that has rank greater than zero.
2	UNCLASSIFIED	Every output of the requested service has been matched with an input of the requested service. However, for at least one input pair the property-match degree is *Unclassified*, i.e. the rank for one matched output pair is one, two or three.
3	SUBPROPERTY	Every output of the requested service could be matched with one output of the advertised service. For every output pair the property-match degree is either *Equivalent* or *Subproperty*, but we have at least one output pair matched with degree *Subproperty*, and the type-match degree is either *Subsumes*, *Invert Subsumes* or *Equivalent*, i.e. the rank for one output pair is four, five or six.
4	TYPE_INVERT	Every output of the requested service could be matched with one output of the advertised service. For every output pair the property-match degree is *Equivalent* but at least for one output pair the type-match degree is *Invert Subsumes*, i.e. the rank for one output pair is seven.
5	TYPE_SUBSUMES	Every output of the requested service could be matched with one output of the advertised service. For every output pair the property-match degree is *Equivalent* but at least for one output pair the type-match degree is *Subsumes*, i.e. the rank for one output pair is eight.
6	MATCH	Every output of the requested service could be matched exactly with one output parameter of the advertised service. For every output pair the property-match degree and type-match degree is *Equivalent*, i.e. all output pairs have rank nine.

Table 5.3: Matching degrees for output parameter matching.

service is subsumed by the type of the output parameter of the advertised service. This becomes clear if one now imagines the advertiser as a requester trying to match the inputs of the requester whom (from his point of view) he sees as an advertiser.

As it was the case with input parameter matching, we now define a property-match as follows:

- *Equivalent.* Both outputs denote the same property.
- *Subproperty.* The output of the advertised service is a subproperty of the output of the requested service.
- *Unclassified.* Either the output of the requested service or of the advertised service is not classified
- *Fail.* If none of the above conditions hold.

The type-match degrees for the outputs are given by:

- *Equivalent.* Both types denote the same concept.
- *Subsumes.* The type of the output parameter of the advertised service subsumes the type of the output parameter of the requested service.
- *Invert Subsumes.* The type of the output parameter of the advertised service is subsumed by the type of the output parameter of the requested service.
- *Fail.* If none of the above conditions hold.

Switching the order of arguments for the output parameter matching when using the `rankForParameters` function, we can re-use the rankings as defined in Table 5.1 and obtain the output parameter matching results as shown in Table 5.3. For every output parameter of the requested service the algorithm tries to find the matching output parameter of the advertised service which would result in the highest rank. As expected, these two output parameters then form a (matched) output pair.

Listing 5.2: Output parameter matching algorithm.

```
public int match(Vector reqOutputsList, Vector advOutputsList, Reasoner reasoner) {
  if (reqOutputsList==null) {
    return MATCH;
  } else {
    if (advOutputsList==null) {
      return PARTIAL_FAIL;
    }
  }
  boolean atLeastOneFail = false;
  int minOverallRank  = 9;
  for (int i=0; i<reqOutputsList.size();i++) {
    Output reqOutput = (Output)reqOutputsList.elementAt(i);
    Output bestMatch = null; int maxRank = 0;
    for (int j=0; j<advOutputsList.size(); j++) {
      Output tempAdvOutput = (Output)advOutputsList.elementAt(j);
      int rank = reasoner.rankForParameters(tempAdvOutput, reqOutput);
      if (rank>maxRank) {
        maxRank = rank;
        bestMatch = tempAdvOutput;
      }
    }
    if (bestMatch==null) {
      atLeastOneFail = true;
    }
    if (maxRank<minOverallRank) {
      minOverallRank = maxRank;
    }
  }
  if (atLeastOneFail) {
    return PARTIAL_FAIL;
  }
  if (minOverallRank==1 || minOverallRank==2 || minOverallRank==3) {
    return UNCLASSIFIED;
  } else if (minOverallRank==4 || minOverallRank==5 || minOverallRank==6) {
    return SUBPROPERTY;
  } else if (minOverallRank==7) {
    return TYPE_INVERT;
  } else if (minOverallRank==8) {
    return TYPE_SUBSUME;
  } else if (minOverallRank==9) {
    return MATCH;
  } else {
    return FAIL;
  }
}
```

In contrast to the input parameter matching, we have a *PARTIAL_FAIL* degree as it is possible that one output parameter of the requested service remains unmatched. In this scenario the advertised service can match some of the requested output parameters, but leaves at least one output parameter unmatched. A partial fail could still prove to be useful in certain situations. Consider the example where a requester seeks a function which provides search results for books in general and presents this information to its users via the web. The requester expects the Web Service to return information on the price, the author, the title, the ISBN number etc. An advertised service however, that returns only part of this information might still be of use for the requester, as users could still find their desired book based on partial information.

Listing 5.2 shows the output parameter matching algorithm in Java. The vectors `reqOutputsList` and `advOutputsList` represent lists which hold all the output parameters of the requested and advertised service. We note that in the case where the list `reqOutputsList` is empty, the output parameter matching function returns *MATCH*, as the requester does not expect any return value. In the case that the `reqOutputsList` is not empty but `advOutputsList` is, the function returns *PARTIAL_FAIL* as there is at least one output parameter of the requested service which can not be matched (as the advertised service has no outputs).

5.5 Profile Matching

As stated earlier, a service described by DAML-S can be classified into certain categories by letting its profile be an instance of a subclass of the class `Profile`. These service categories itself are nothing more than DAML+OIL ontologies. With profile matching we try to determine how good the service category of the advertised service fits into the service category that the requested service demands.

Let `reqServiceCat` denote the service classification of the requested service, and let `advServiceCat` denote the service classification of the advertised service. Both `reqServiceCat` and `advServiceCat` are thus concepts defined in some service classifying ontology. The identified matching results when matching the profile of the

Rank	Degree of match	Explanation
0	FAIL	The two concepts `advServiceCat` and `reqServiceCat` are in no relation with each other. i.e. for one input the rank is zero.
1	UNCLASSIFIED	Either `advServiceCat` or `reqServiceCat` is not classified, i.e. one of them is a direct instance of the class Profile.
2	SUBSUMES	`advServiceCat` is subsumed by `reqServiceCat`. Here `advServiceCat` denotes a more specific concept than `reqServiceCat`. This might still be a good case since the classification of the advertised service is a subclass of the classification of the requested service, meaning that the advertised service offers a more specific functionality than demanded by the service requester.
3	MATCH	`reqServiceCat` and `advServiceCat` are equivalent. This is obviously the best case scenario since the advertised service is exactly the type of service that the requested service was looking for.

Table 5.4: Matching degrees for profile matching. `reqServiceCat` and `advServiceCat` denote the service categories for the requested and advertised service.

service request and the advertised profile are shown in Table 5.4.

To see why the *SUBSUMES* degree makes sense, we consider a web example again. Assume a computer program which acts as a broker wants to let its user buy computers from different web stores. Therefore the program needs to make use of functions by web merchants who offer Web Services which allow to buy computers. The broker program thus looks for Web Services which are classified into the service category `BuyComputers`, for example. Any advertised Web Service which falls into the categories `BuyDesktopComputers` or `BuyLaptopComputers`, which both are subcategories of the category `BuyComputers`, would still be useful to the computer agent,

Listing 5.3: Profile matching algorithm.

```
public int match(String reqServiceCat, String advServiceCat, Reasoner reasoner) {
  if (reqServiceCat==null || advServiceCat==null) {
    return UNCLASSIFIED;
  }
  if (reqServiceCat.equals(DamlsParser.profileID) ||
      advServiceCat.equals(DamlsParser.profileID)) {
    return UNCLASSIFIED;
  } else {
    int match = reasoner.conceptMatch(reqServiceCat, advServiceCat);
    if (match == Reasoner.EQUIVALENT) {
      return MATCH;
    }
    if (match == Reasoner.SUBSUMES) {
      return SUBSUMES;
    }
    else {
      return FAIL;
    }
  }
}
```

allowing its users in turn to buy more specific kind of computers.

Listing 5.3 shows the profile matching algorithm in Java.

5.6 User-Defined Matching

The matching algorithm also allows for user-defined matching. In user-defined matching, the entity who manages the matching algorithm can define additional matching functions by the means of a plug-in. Any self-defined matching function returns either true or false, depending on whether the matching was successful or not. Every plug-in function receives a certain set of information about the requested and advertised service, such as the respective service descriptions in DAML-S, and can then decide if the requested and advertised service match.

Effective use of this matching type can be done when evaluating additional information of the service profile provided by DAML-S. For example, an important aspect that comes with the user-defined matching is the possibility of exploiting quality of service aspects for web Services. As described earlier, a profile in DAML-S allows for

the specification of quality of service behavior for a given Web Service via its class `QualityRating`. Thus, given well-defined and well-known quality of service definitions, it is possible to create plug-ins that perform a quality of service matching based on the available additional information in a service profile. These plug-ins might then be found useful by a broad range of users. In general, with the self-definable service parameter elements in the DAML-S profile it is basically possible to specify any additional user constraints for the requested service to ensure that the advertised profile will support these.

There can be as many matching plug-ins as desired, and each plug-in can be individually enabled or disabled. All matching results from self-defined plug-ins are interpreted conjunctively. If one user-defined matching function fails, the whole user-defined matching fails. Therefore the following results have been identified with user-defined matching:

MATCH Every user-defined plug-in that is activated returns true. Deactivated plug-ins are not considered.

FAIL At least one of the activated user-defined plug-in matching function fails, i.e. at least one function returns false.

Listing 5.4 shows the plug-in matching in Java. The vector `userDefinedPlugIns` contains a list of all plug-ins, whereas the variables `reqService` and `advService` contain the respective services.

More about the plug-in mechanism and how these plug-ins are managed will be described in more detail in Chapter 6.

5.7 The Final Matching Result

The final matching result for the two considered services is composed of the matching results of the four previously described matching parts and of so called minimal

Listing 5.4: Plug-in matching algorithm.

```
public boolean match(Vector userDefinedPlugIns, Configuration config,
      Service reqService, Service advService, Reasoner reasoner) {
  if (userDefinedPlugIns==null) {
    return true;
  }
  for (int i=0; i<userDefinedPlugIns.size(); i++){
    UserPlugIn plugIn = (UserPlugIn)userDefinedPlugIns.elementAt(i);
    String fullPlugInName = plugIn.getClass().getName();
    if (config.isActivePlugIn(fullPlugInName)) {
      if (!plugIn.match(reqService, advService, reasoner)) {
        return false;
      }
    }
  }
  return true;
}
```

matching degrees the user expects. In addition to specifying the requested service and the advertised service, the user of the matching algorithm specifies lower bounds on the matching degrees for the input parameter, output parameter and profile matching. Each partial matching result has to satisfy its minimal requirement (i.e. return a matching degree ranked higher or equal than the lower bound) for the matching algorithm to succeed. In addition, if the user-defined matching returns FAIL, the final matching result will also return FAIL, regardless of the results of the other three partial matching results. If it returns MATCH, the matching result is only based on the other three partial matching results as described.

We therefore have the following final matching result:

MATCH Each partial matching result (input, output and profile matching) satisfies its minimal expected degree, and the user-defined matching returns *Match*.

FAIL At least one of the partial matching results does not satisfy its minimal expected degree, or the user-defined matching returns *FAIL*.

This matching procedure is illustrated in Figure 5.1.

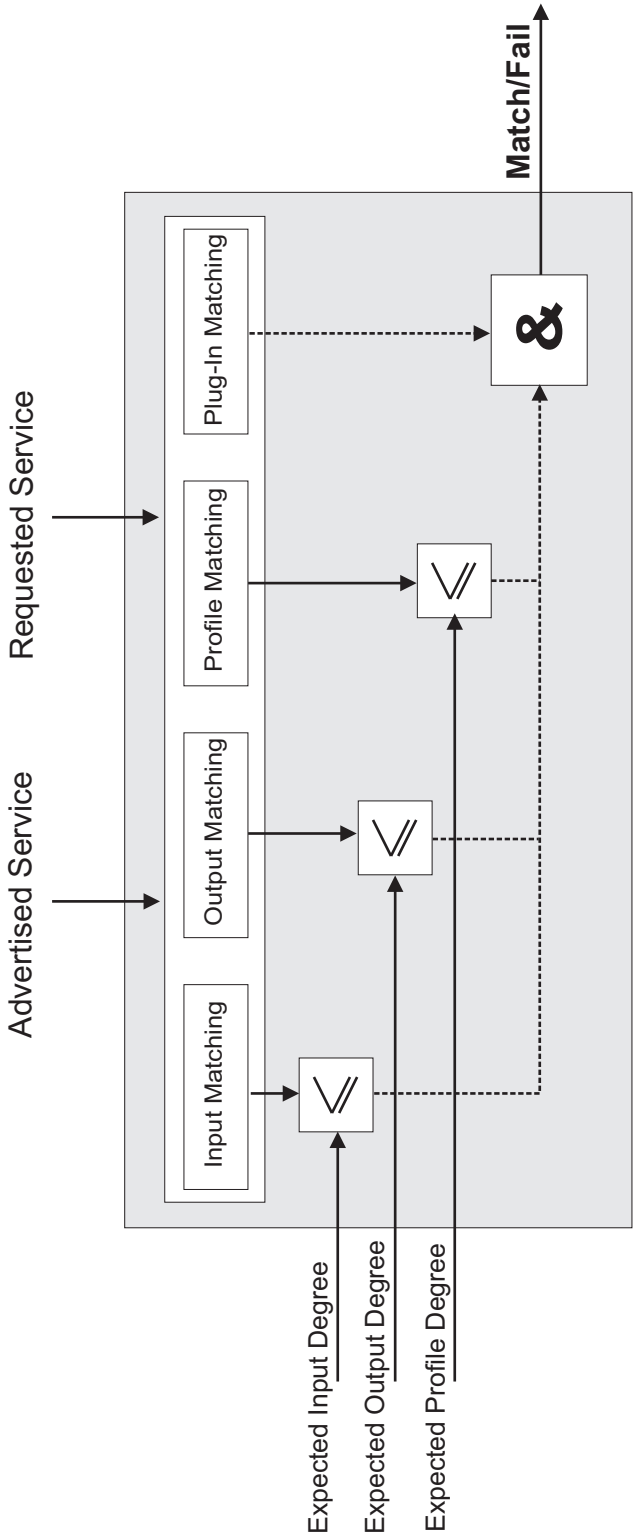

Figure 5.1: The Matching Algorithm.

Chapter 6

Implementation

This chapter presents the implementation of the concepts that were developed in this work and described before. All the programming tasks were done in the object-oriented language Java [16]. Besides a detailed description of the basic principles of the implementation, a brief summary of existing technologies is given, which assisted in realizing this work.

We start by looking at the implementation of the Apache Axis plug-in which realizes the automatic discovery of DAML-S documents, as described in section 4.3.1. What follows is a look at the matching algorithm itself. We conclude this chapter with a short explanation of the graphical user interface (GUI) that was developed. This GUI displays the semantic descriptions of Web Services; allows the user to call the matching algorithm and finally displays the matching results.

All the developed software comes along with this work and can be found on the CD-ROM. This distribution also includes Java API documentation as well as some DAML-S sample files, which are described in Chapter 7.

6.1 The DAML-S Apache Axis Plug-In

The DAML-S plug-in was written for use with the Apache Axis [27] module, which can be used in conjunction with the Apache Tomcat Web Server. Both the Apache

Tomcat Web Server and the Apache Axis are open source projects that were developed by the Apache Foundation and are publicly available at [1].

Apache Tomcat Web Server and Apache Axis The Tomcat Web Server is an open source Web Server which supports the Java J2EE specification [9]. The Web Server supports a so called servlet container, which allows the deployment of Java Servlets, Java Server Pages etc.

The Apache Axis is an implementation of the SOAP protocol for Java (it is also available in C++). It can be deployed in most servlet containers, such as the Tomcat Web Server. The Axis module provides means for the use of Web Services which are written in the Java language. Once the Axis is deployed in a servlet container (the Axis itself is nothing more than a servlet), it handles all the work regarding Web Services, i.e. the decoding of incoming SOAP messages, calling the appropriate service etc.

6.1.1 DAML-S Plug-In

The DAML-S plug-in is a so called query string plug-in. The Apache Axis implementation admits query string plug-ins in order to define queries to a Web Service other than the three standard queries: *?wsdl*, *?version* and *?method*. Every query string plug-in implements the interface `org.apache.axis.transport.http.QSHandler`. This interface defines one method: `invoke`. Every query string plug-in thus has to implement this method; anytime the service is queried with the query string (such as *?damls* in our case) this method is called. This method receives one parameter of type `MessageContext` when invoked. As the name suggests, the class `MessageContext` provides information about the context in which the plug-in is called. For example, we can obtain the information which Web Service is queried etc.

The rather short invoke method for the DAML-S query string plug-in can be seen in Listing 6.1.

The idea for our plug-in is rather simple. Every time a new Web Service is deployed, the user who deploys the service has to provide a parameter via the deployment descriptor. This parameter is called *DAMLS_PATH*, and its value is the absolute path to the DAML-S description of this particular Web Service. When the plug-in is invoked, the program first retrieves relevant context information, such as the `HttpServletResponse` object and the `PrintWriter` object which allows the plug-in to send an answer to the calling entity. It then reads out the parameter, opens the DAML-S file and then copies this file to the response stream.

Listing 6.1: Invoke method of the DAML-S plug-in.

```
public void invoke (MessageContext msgContext) throws AxisFault {

   // The writer to write the response to the user
   PrintWriter writer = (PrintWriter)
          msgContext.getProperty(HTTPConstants.PLUGIN_WRITER);
   // The response object
   HttpServletResponse response = (HttpServletResponse)
          msgContext.getProperty (HTTPConstants.MC_HTTP_SERVLETRESPONSE);
   // The service object
   SOAPService service = msgContext.getService();
   // Path to the DAML-S service file
   String path = (String)service.getOption("DAMLS_PATH");
   try {
        File damlsFile = new File(path);
        FileInputStream in = new FileInputStream(damlsFile);
        int c;
        while ((c=in.read())!=-1) {
          writer.write(c);
        }
        writer.flush();
   } catch (FileNotFoundException fnfe) {
        // The DAMLS file was not found, generate HTTP 404 Error
        response.setStatus(HttpURLConnection.HTTP_NOT_FOUND);
        response.setContentType("text/html");
   } catch (IOException ioe) {
        // The DAMLS file could not be opened, generate HTTP Internal Error
        response.setStatus(HttpURLConnection.HTTP_INTERNAL_ERROR);
        response.setContentType("text/html");
   }
}
```

Listing 6.2: Sample Deployment Descriptor which activates the Axis plug-in.

```
<deployment name="test" xmlns="http://xml.apache.org/axis/wsdd/">
   <service name="MyService">
      <parameter name="DAMLS_PATH" value="/home/Service.damls" />
      <transport name="http">
         <parameter name="qs.damls" value="de.tuberlin.ivs.damls.QSDamlSHandler"/>
      </transport>
   </service>
</deployment>
```

In case that the DAML-S file could not be found by the plug-in, a standard HTTP 404 Not Found error is returned. This could happen if the value of the parameter *DAMLS_PATH* is not correctly specified. In the case that the file could not be read, a standard HTTP 500 Internal Error is returned. The reason why standard HTTP errors are returned is obvious; we designed this plug-in as to assist the automatic discovery and execution of Web Services. It is therefore highly likely that an autonomous computer program or agent will query the Web Service via the *?damls* mechanism, and thus the program needs to be able to automatically handle the cases where an error occurs. This is possible when using standard HTTP errors.

6.1.2 Deploying the Plug-In

The only question that is left is how to activate the plug-in. When a Web Service is deployed through its deployment descriptor, one can define the activation of any query string plug-in for this Web Service. This is done by adding a simple parameter to the transport section of the deployment descriptor. The parameter name defines the query string to which the service should react, and the value points to the actual class that implements the query string handler. For example, if the name of the Web Service to be deployed is *MyService*, then the sample deployment descriptor shown in Listing 6.2 will deploy the service and enable the *?damls* query.

If this service is deployed on the server *http://myServer*, then the corresponding DAML-S specification could be obtained under *http://myServer/MyService?damls*, assuming the DAML-S document is available at the expected location */home/Service.damls*.

6.2 Assisting Technologies

Here a brief overview of software is given that is used in the implementation.

JDOM JDOM [19] is an XML parser for Java. It uses the Simple API for XML (SAX) and the Document Object Model (DOM) and provides a convenient and easy accessible API for programmers. The JDOM parser parses an XML file into the Document Object Model, which is a tree-like representation of the parsed XML file. From this DOM the XML file can be evaluated by browsing its tree structure.

Jena Jena [23] is a Java framework for enabling Semantic Web applications. Jena is open source software and was developed at the HP Semantic Labs. It provides a programmatic environment for RDF, RDFS and the Web Ontology Language (OWL). In essence, Jena converts RDF files into an RDF model represent by subject-verb-object (SVO) triples and supports information retrieval through the RDQL query language. It also includes a rule-based inference engine for basic reasoning support.

Jess The Java expert system shell (Jess) [14] is a rule engine and scripting environment written in Java and developed by the Sandia National Laboratories. Jess supports the development of rule-based expert systems. With Jess one can construct knowledge bases in Java and reason over them. Jess uses the powerful Rete algorithm [13] to process rules and deduce new information. Jess uses the KIF axiomatization [15] for representing its rules and facts.

DAMLJessKB DAMLJessKB [21] is a Description Logics Reasoner for ontologies written in DAML+OIL. DAMLJessKB provides a Java API that supports the reading and information retrieval of DAML files. DAMLJessKB utilizes both Jena and Jess. DAMLJessKB currently supports the basic reasoning tasks such as subsumption and classification and therefore provides the functionality demanded by the matching algorithm. The basic workflow of DamlJessKB is as follows:

- Read in Jess rules and facts that represent the DAML language.

- Read in the DAML file using Jena and create the SVO triples.
- Assert these triples into the Jess Rete network.
- Apply Jess rules to the data (which eventually deduces new information).
- Make the data available through queries.

As DAMLJessKB uses Jess to store its knowledge base, queries to the DAML data is thus performed via the KIF language.

6.3 The Matching Algorithm

The matching algorithm class provides one central method: `match`. This method matches two services and returns the matching result. The basic flow is as follows:

- A configuration file written in XML is evaluated and all the specified user-defined plug-ins are loaded.
- Both services are parsed into an internal Java representation.
- A DAMLJessKB object is created, and all the semantic information regarding these two services is loaded into the knowledge base. Also, a reasoner class is created, which wraps around the DAMLJessKB object and provides convenient access to the underlying knowledge base.
- The matching is executed with the help of the reasoner class.

We will describe all these steps in further detail now.

6.3.1 The Configuration File

The behavior of the matching algorithm can be controlled through a configuration file. This configuration is encoded in an XML file named *config.xml*. As of now, the configuration file serves mainly as a means for specifying the user-defined plug-ins. The following elements are currently supported:

Listing 6.3: Sample configuration file.

```
<matching-config>
        <plug-in>
                <name>myPlugIn</name>
                <class>de.tuberlin.ivs.daml.plugin.myPlugIn</class>
                <active>true</active>
        </plug-in>
</matching-config>
```

- `<matching-config>` The root element of the configuration file.
  - `<plug-in>` Belongs at the top-level inside the `<matching-config>` element. Defines a plug-in, and there may be arbitrarily many `<plug-in>` elements inside the root element.
    - `<name>` Specifies the name of the plug-in.
    - `<class>` Specifies the name of the implementing class of the plug-in, including the package path.
    - `<active>` Specifies whether the plug-in is active or not. Permissible values are *true* and *false*.

The class `de.tuberlin.ivs.daml.matching.config.ConfigurationReader` reads the XML configuration file with the use of the JDOM XML parser. Every time the matching algorithm is instantiated, the configuration file will be read.

A sample configuration file that loads one plug-in can be seen in Listing 6.3.

6.3.2 The Service Parser

Both the advertised and requested service are parsed into an internal representation of a DAML-S description with the use of the JDOM XML parser. These representation classes hold all the information of the respective services and provide methods that allow convenient access to these properties. For example, as a service can have multiple profiles, the representation class of a service allows one to easily obtain a specific profile by name, and from a profile one can obtain all input parameters etc by just calling the respective functions instead of having to deal with the XML file

(or the parsed XML file for that matter) every time one wishes to extract information from the DAML-S descriptions. These classes thus provide more convenience to programmers, especially with regard to the user-defined plug-ins, as both parsed services are passed on to any implementing plug-in. These representation classes are found in the package `de.tuberlin.ivs.daml.service`, and their exact functionality can be read from the Java API documentation. The parser itself is implemented in class `de.tuberlin.ivs.daml.service.DamlsParser`.

6.3.3 The Reasoner Class

The reasoner is implemented in class `de.tuberlin.ivs.daml.matching.Reasoner`. The reasoner wraps around a DAMLJessKB object and holds the current knowledge base. It provides four central methods:

- `loadDamlFile(URL path)`. Loads a DAML file and adds its information to the current knowledge base.
- `rankForParameters(Parameter param1, Parameter param2)`. Determines the parameter ranking according to Table 5.1.
- `propertyMatch(PropertyA, PropertyB)`. Determines the relationship between `PropertyA` and `PropertyB` as defined in Section 5.2.2 by querying the current knowledge base.
- `conceptMatch(ConceptA, ConceptB)`. Determines the relationship between the concepts `ConceptA` and `ConceptB` as defined in Section 5.2.1 by querying the current knowledge base.

The latter two methods use the KIF axiomatization [15] for querying the knowledge base.

We give a small excerpt of how querying the knowledge base works. Consider we want to query for the subproperty relationship. First we create a query on the knowledge base `kb` (an instance of the class DAMLJessKB) via

```
kb.executeCommand("(defquery query-subproperty " +
     "(declare (variables ?y)) (PropertyValue " +
     "http://www.w3.org/2000/01/rdf-schema\#subPropertyOf ?x ?y) )");
```

Next we run this query on the current knowledge base and ask for matching subjects (remember, the information in the knowledge base is stored as subject-verb-object triples). Let the variables `propertyA` and `propertyB` denote the names of the involved properties. The following code line returns a vector of all subjects that satisfy the query (i.e. all properties that are subproperties of `propertyA`):

```
Vector values = kb.simpleSubjectQuery("query-subproperty",
     new String[] {propertyA},1);
```

We can then iterate through this list and find out if `propertyB` is in that list, i.e. if `propertyB` is a subproperty of `propertyA`:

```
for (int i=0; i<values.size(); i++) {
     String value = (String)values.elementAt(i);
     value = value.substring(value.lastIndexOf("\#")+1);
     if (value.equals(propertyB)) {
             return true;
     }
}
return false;
```

We thus obtain *true* if `propertyB` is a subproperty of `propertyA` and *false* otherwise. Querying the knowledge base for other relationships works in a similar fashion by first defining the corresponding queries and then executing these on the knowledge base. Of course it is also possible to ask for objects instead of subjects and so forth.

6.3.4 User-Defined Plug-Ins

User-defined plug-ins are loaded at runtime with the dynamic class loader mechanism. Each plug-in that is defined in the configuration file will be instantiated right after the

configuration file is evaluated. For this to work properly, every plug-in has to implement the interface `de.tuberlin.ivs.daml.matching.UserPlugIn`. This interface specifies one method:

```
public boolean match(Service reqService, Service advService,
    Reasoner reasoner);
```

The arguments of this method are the representation classes of the advertised and requested service, and an instance of the Reasoner class that provides access to the knowledge base. This method thus has to be implemented by each plug-in (as it implements this interface), allowing the matching algorithm to dynamically load and correctly invoke any plug-in. All the functionality that a plug-ins wants to provide has to be determined through the `match` method (either by directly implementing the appropriate code or by calling other methods). When a plug-in is declared as active, any time the matching algorithm is executed the corresponding `match` method of the plug-in is invoked and the result is taken into account for the final matching result.

6.3.5 The Matching Algorithm

When an instance of the matching algorithm is created, the configuration file is read and the user-defined plug-ins are loaded through its constructor. The constructor has the following signature:

```
public MatchingAlgorithm(PrintStream out) throws
    ConfigurationException, PlugInLoadException
```

The exceptions are explained in the next section. The variable `out` specifies the stream to which output information is written. The matching algorithm exposes one central method:

```
public boolean match(URL reqServiceURL, String reqProfileID, URL
    advServiceURL, String advProfileID, int minProfileDegree, int
    minInputDegree, int minOutputDegree) throws DamlsParseException
```

This method takes the URLs to the DAML-S descriptions of the requested service and the advertised service. It then parses both files into the internal representation

and loads its information into the knowledge base through the reasoner class. Since a service can have multiple profiles, the matching algorithm also takes the IDs of the profiles from the two services that should be matched. If a profile ID for a service is not specified, the algorithm matches the first profile definition it encounters. It then performs the matching. The matching result is based on the requirements the user poses towards the requested and advertised service in terms of the minimal matching degrees. The three integer arguments `minProfileDegree`, `minInputDegree` and `minOutputDegree` that the user has to provide to the matching algorithm specify the minimal expected degree of matching for the respective partial matching results. These integer values are simply the ranks as they were defined in Chapter 5 for the partial matching stages. This function then returns *true* if the matching results of the requested service with the advertised service satisfies the minimal matching degrees.

As the IDs of the profiles are probably not known beforehand, or if one wishes to match every profile within a particular service, the class also provides the method

```
public Vector getProfileIDsForService(URL serviceURL)
    throws DamlsParseException
```

This method returns a vector of all the IDs of the profiles that a service exposes. If one wishes to match all profiles for any given service, one would first call this helper method to obtain all the profile IDs and then call the matching algorithm for every profile.

As the algorithm is splitted, it makes use of three other classes which implement the input parameter, output parameter and profile matching. Each of these classes exposes its very own `match` method, which performs the respective matching and also returns an integer value that specifies the matching result. Each class also provides the necessary integer flags that define its possible matching results so that the matching algorithm can easily evaluate the partial results. After these three classes are called, every active user-defined plug-in will be evaluated by calling the plug-ins `match` method.

6.3.6 Exceptions

There are three exceptions that can arise and tell the user that something has gone wrong. These exceptions wrap around other exceptions encountered during the matching and its information can be retrieved through the `getMessage()` method.

ConfigurationException Arises when an error occurred while the algorithm tried to read the configuration file. For example, when the *config.xml* file could not be found, or when the XML file is malformed this exception is thrown.

PlugInLoadException Arises when an error occurred while the algorithm tries to dynamically load all specified plug-ins. For example, when the implementing class of a plug-in is not found, or when the instantiation of the plug-in failed, this exception is thrown.

DamlsParseException Arises when an error occurred during the parsing of the DAML-S specifications. For example, when the path to the requested or advertised service DAML-S files is not correct, this exception is thrown.

6.4 The Graphical User Interface

This interface allows the testing and verification of the matching algorithm. It enables the user to load a requested and an advertised service, and then displays both services along with its basic properties, such as all the inputs and outputs. The user can specify the minimal expected matching degrees; have the matching algorithm applied to the selected profiles and finally view the results of the matching algorithm.

A snapshot of the GUI is shown in Figure 6.1. For more information about the GUI the user is referred to the Java API documentation that comes along with the CD-ROM.

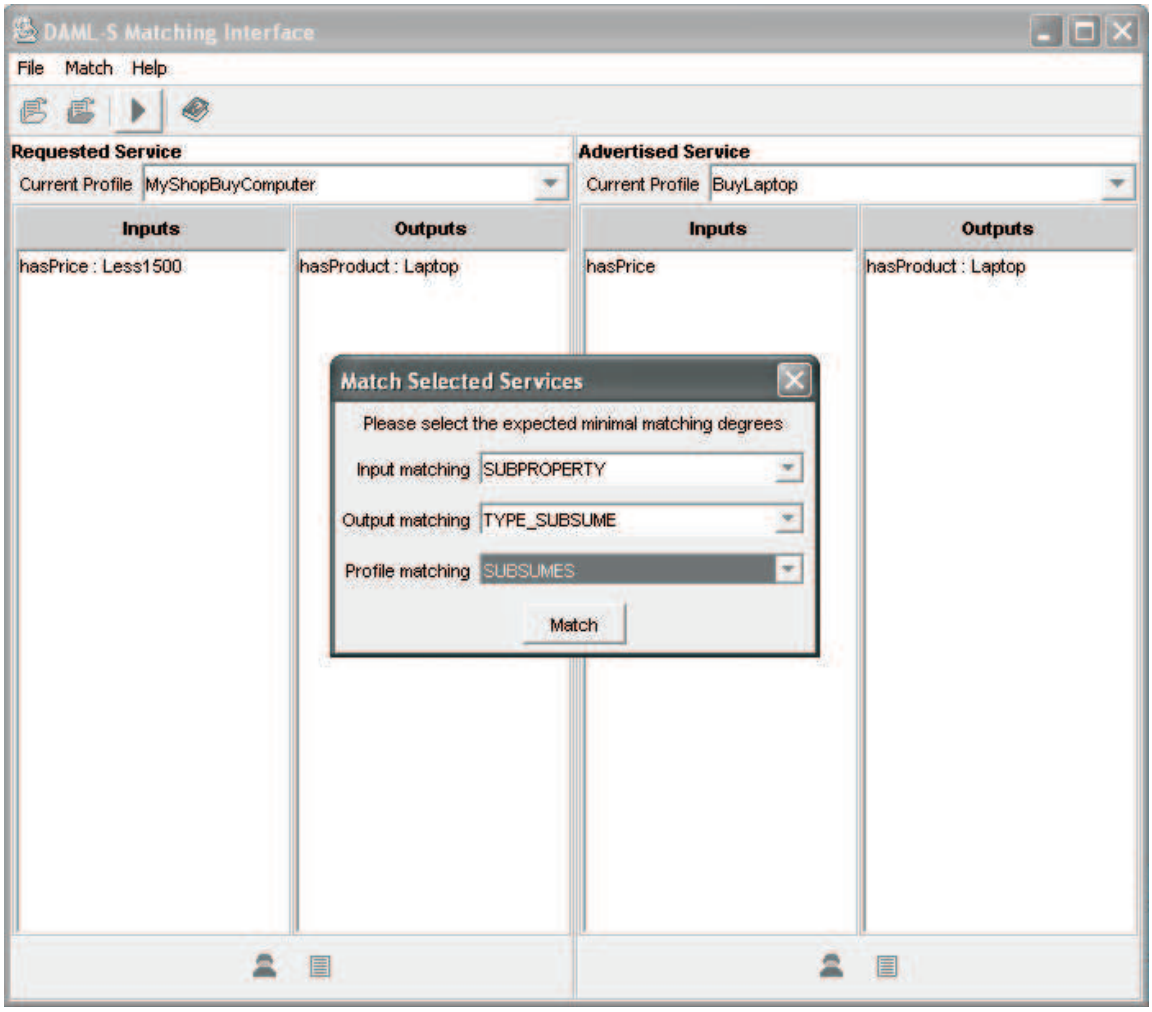

Figure 6.1: A snapshot of the Graphical User Interface.

Chapter 7

A Simple Example

We consider a very simple example that will demonstrate the usage of the matching algorithm.

We will look at a small fictitious business called *MyShop*, which sells laptop computers to its clients. As *MyShop* always tries to offer a broad range of laptop computers at low prices to its customers, it buys the hardware from different retailers. To achieve a high degree of flexibility, the list of retailers is not fixed, and every time *MyShop* makes a new order to refill its inventory, it queries a DAML-S database and tries to find a Web Service that matches its need. There are two other businesses which sell computers, and thus are potential business partners of *MyShop*. These businesses include *LaptopInc* and *PCLand*, each of which provides a semantic description of their offered services.

7.1 Necessary Classifications and Ontologies

The first step for a successful utilization of the matching algorithm is to define ontologies such that we are able to classify services in a useful way. The first classification is a small ontology that defines some concepts from the computer domain, shown in Figure 2.4. As we can see it defines a concept `Computer`, and subdivides computers into Apple computers and personal computers, the latter one subdivided again into Laptop computers and Desktop computers. Also, a PC has at least one `CPU`, either

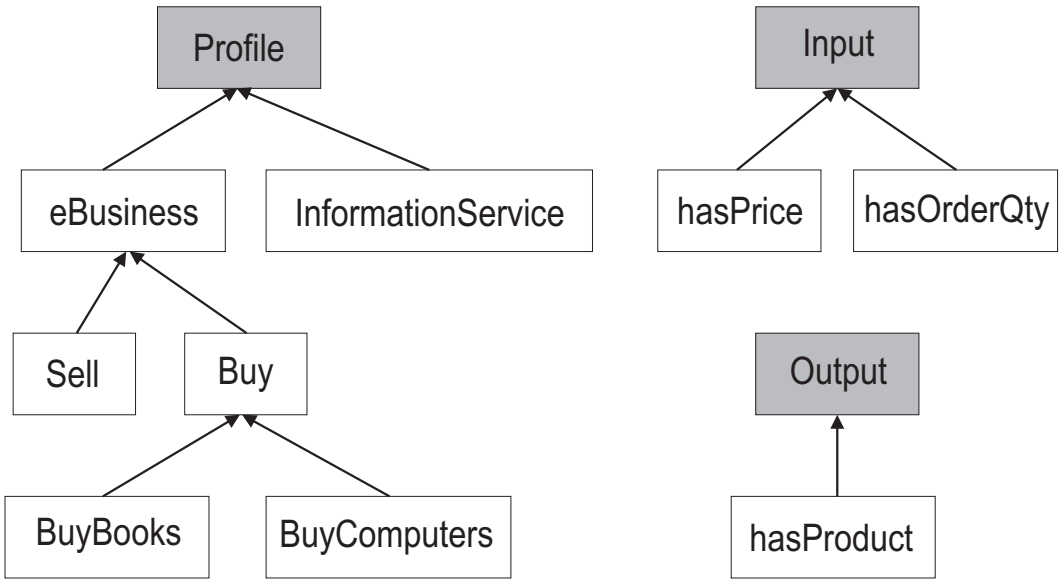

Figure 7.1: Profile, Input and Output classifications.

an `Intel` or an `AMD` processor.

Next we define service categories and input and output classifications, these are shown in Figure 7.1. The class `Profile` is splitted into the concepts `eBusiness` and `InformationService`, so that services can classify themselves as either services that do business on the web or that provide information (such as weather reports, stock quotes etc.). In addition, for the purpose of *MyShop*, a web business can either sell or buy things; in particular, services that allow their clients to buy things through their businesses can classify themselves as computer selling services (via the concept `BuyComputers`) or book buying services (`BuyBooks`).

As for the inputs, we define two subproperties: `hasPrice` and `hasOrderQty`. The first property classifies an input parameter as the price, and the latter defines an order quantity. We only classify one output parameter: `hasProduct`. This property classifies one output parameter as one that returns a product. Concepts and properties shaded in grey were originally defined in the DAML-S service ontology.

As we deal with commercial businesses, we also need to define the concept of a price. Figure 7.2 shows a small sample ontology that defines the concept of a price. The concept `Price` has two properties: a data-type property called `hasValue`

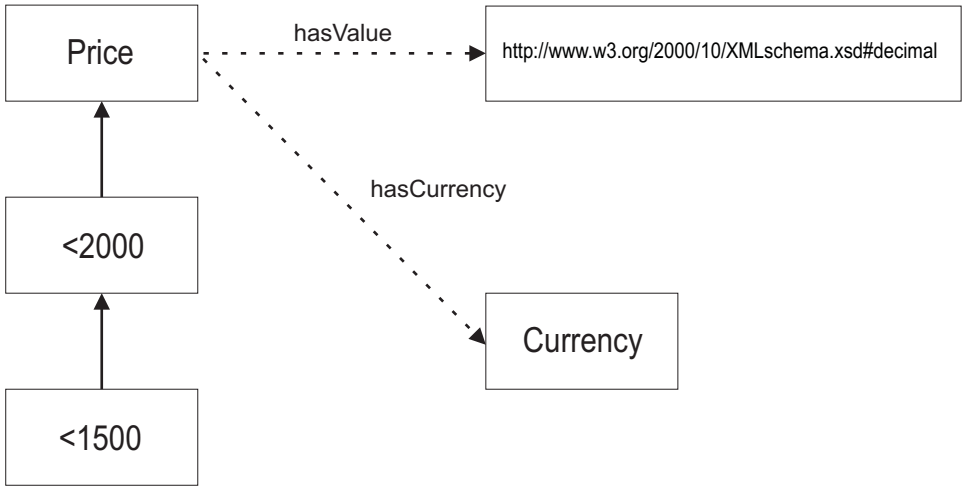

Figure 7.2: Ontology defining the concept of a price.

referring to an XML Schema data-type denoting the actual price and a property called `hasCurrency`, which specifies the currency of the price via the concept `Currency`. We note that it is not possible to define a numerical restriction on the actual value of a datatype property at runtime. We therefore introduce the concepts `<2000` and `<1500` to denote prices that are less than 2000 units and 1500 units, respectivly.

We assume that all three ontologies are domain-independent, i.e. they are publicly available and issued by some central authority, and thus accessible by all involved parties.

7.2 Service Definitions

We now look at the service descriptions of all three businesses. Every service definition imports the three previously defined ontologies. All definitions are represented graphically in Figure 7.3. The service itself is represented as a grey box; the name of the box denotes the service category this service is classified into. The inputs of the service are coming in on the left and the outputs are going out on the right.

Service definition for *MyShop* *MyShop* provides an abstraction of an ideal Web Service that it is looking for. As *MyShop* wants to buy laptop computers, it specifies

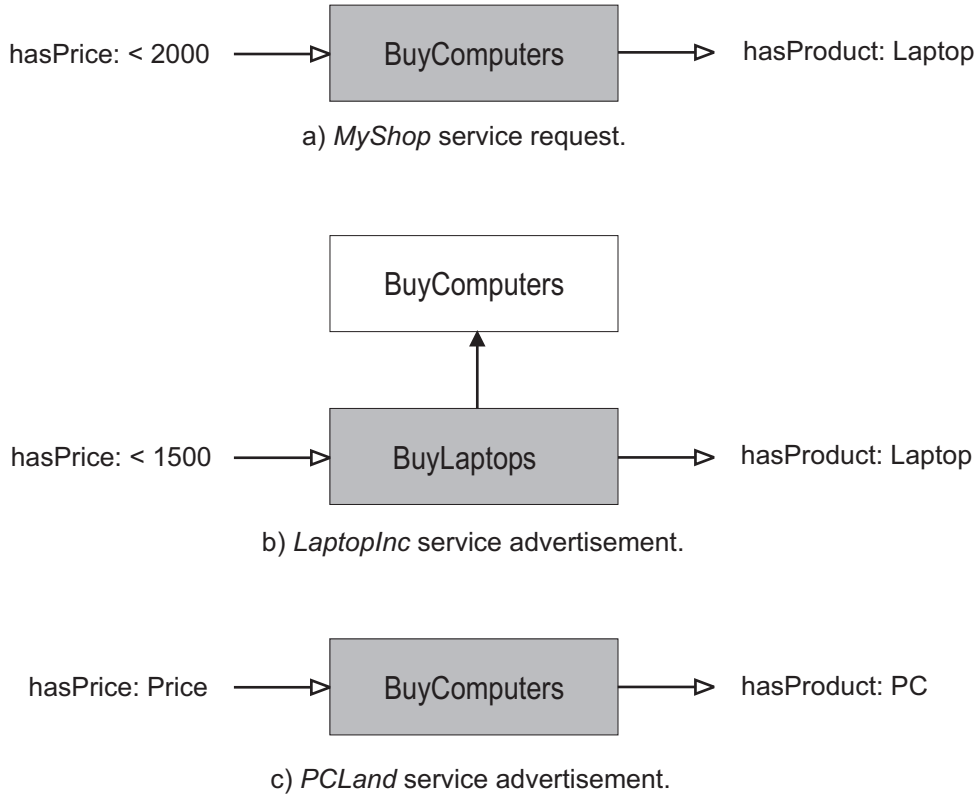

Figure 7.3: Service requests and advertisements for the involved parties.

that the requested service should be classified into the category `BuyComputers`. The only input parameter it will supply is the price, thus the lone input will be classified as `hasPrice`. The type restriction is that the price can not exceed 2000 units. As a return value it expects a product of type `Laptop`. The service definition is shown in Figure 7.3 a).

Service definition for *LaptopInc* As the name suggests, *LaptopInc* specializes in selling laptop computers. As the service category `BuyComputers` is too general, it defines a new category `BuyLaptops` as a subconcept of the category `BuyComputers`. In general, this new category is a local classification, and one can not expect that other services have access to the local extended ontology. This service also specifies one input parameter, `hasPrice`, and one output parameter classified as `hasProduct` with type `Laptop`. It also states (via its type) that the price will not exceed more

than 2000 units. The service definition is shown in Figure 7.3 b).

Service definition for *PCLand* *PCLand* sells personal computers in general, both laptop and desktop computers. It therefore classifies its service with the category `BuyComputers`. It also specifies the `hasPrice` input property and as expected an output parameter of type `PC`. The service definition is shown in Figure 7.3 c).

7.3 Application of the matching algorithm

We now assume that *MyShop* queries a DAML-S providing entity, such as a central database as explained in Section 4.1, where all three service advertisements are registered. This entity implements the matching algorithm. *MyShop* thus has to provide a specification of its service requests and the expected minimal matching degrees for the service classification, and the input and output properties. Note that the domain-independent ontologies as stated in Section 7.1 do not need to be explicitly specified, as they are imported by the service definition and thus will automatically be evaluated by the matching algorithm. *MyShop* specifies the following minimal matching degrees:

- Profile matching: *SUBSUMES*. *MyShop* expects that the advertised service sells computers. If it does in fact sell a more specific type of computer, it might still be of use.

- Input parameter matching: *TYPE_SUBSUME*. It is expected that the advertised service offers exactly the same input parameter (namely `hasPrice`), and that they can differ by at most the type it specifies. If any service specifies an input parameter whose type is subsumed by `<2000`, then it is assumed that this type will denote a smaller price type and *MyShop* thus could use this service, paying a lesser price than expected.

- Output parameter matching: *TYPE_SUBSUME*. Again, the output parameter has to match and at most the type can differ. It is usually assumed that a

service which returns a type that is more general than the concept `Laptop` sells computers, and that will also include laptops.

Now we match the requested service profile of *MyShop* with every advertised profile according to these minimal matching degrees.

Matching the *MyShop* service request with the service advertisement of *LaptopInc* yields in a *MATCH*, i.e. the matching algorithm returns *true*, as we have the following results:

- The profile matching degree returns *SUBSUMES*, as `BuyLaptops` is a subconcept of `BuyComputers`.
- The input parameter matching degree returns *TYPE_SUBSUME*, as the concept `<2000` is a superconcept of the concept `<1500`.
- The output parameter matching degree returns *MATCH*, as both services specify exactly the same property and type for all outputs.

Based on the minimal expected matching degrees, *MyShop* can make use of the service offered by *LaptopInc*, as every partial matching degree results in a higher rank than expected.

If we now match the *PCLand* service advertisement against the *MyShop* service request, the matching algorithm returns *FAIL*, or *false*, for the following reason:

- The input parameter matching degree is *TYPE_INVERT*, as the concept `<2000` is subsumed by the concept `Price`.

This means that *MyShop* would have to make use of a service whose price might exceed the limit of 2000 units.

In the case that *MyShop* does not obtain enough matches to fulfill its task, it could now weaken its minimal matching degrees towards the advertised services. For example, the input parameter requirement could be set to *TYPE_INVERT* so that services which offer Laptops for a higher unit price will also be considered.

Of course, these examples are very simple, but nevertheless demonstrate how a service requester can find appropriate services by specifying the expected degree of connectivity. Although the presented sample services provide only one input and output parameter each, the matching procedure works in the same way for multiple parameters. All ontologies in this example were kept simple, however the real benefit of semantic reasoning becomes apparent once the ontologies grow in size and concepts and properties start cross-referencing among many domain independent ontologies, such that the relationship among concepts and properties are not as evident as in this example.

Chapter 8

Conclusions

In accordance with the goal of this work, a matching procedure has been developed that assists service requesters finding compatible Web Services based on semantic descriptions. We first explored the necessary background in Description Logics and DAML-S, and then considered the design of the matching algorithm. In addition to the implemented matching of the core information such as profile classification, input and output matching, the matching algorithm has been designed to a very flexible degree, as it can be extended in many possible ways through plug-ins.

We also looked at possible scenarios in which the matching algorithm can be applied, and developed a mechanism that allows for the integration of the matching procedure into existing infrastructure components such as UDDI.

As for verification and testing purposes, a simple GUI has been programmed that allows the easy and convenient application of the matching algorithm to a set of services.

A splitted algortihm as we proposed in this work has the huge advantage that we expect a higher degree of positive matches, as explained in Section 5.1.1. By allowing different rankings for each stage of the matching procedure, the user can shift the priorities of each matching result. We conclude this work with the note that in general, any specific defined ranking for matching degrees might appear to be somewhat arbitrary, i.e. other users might feel that different ranking schemes

might be a better approach. However, a reasonable solution is being provided by the rankings defined in this work, as it was demonstrated with the simple examples in the previous Chapter.

8.1 OWL-S

Just before this thesis was finished, a successor to DAML-S has been recommended by the W3C. This new standard is called OWL-S [10] and builds on top of OWL (Ontology Web Layer) [28]. Like DAML+OIL, OWL is a language for defining web ontologies. The semantics in OWL differ from DAML+OIL, as various constructs have been eliminated (such as qualified restrictions) while others have been added.

In addition to the fact that OWL-S differs from DAML-S in that it is based on OWL, a few classes have been eliminated (such as the class `Actor`), and some name changes were made. For example, the property `input` changed to `hasInput`, `output` to `hasOutput` and so on.

8.2 Outlook

Now that it seems that DAML-S has been discontinued, the next step would be the migration of the existing implementation of the matching algorithm to the latest OWL-S standard. The basic design of the matching algorithm would essentially be left unaltered, however, as OWL has different semantics than DAML+OIL, the whole underlying knowledge base and its querying techniques need to be replaced.

Also, for a full appreciation of the designed matching procedure, a set of useful plug-ins needs to be developed that would allow to process every available information that is provided with a service profile.

Bibliography

[1] *The Apache Software Foundation.* http://www.apache.org.

[2] *The DAML Coalition.* http://www.daml.org.

[3] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider. *The Description Logic Handbook: Theory, Implementation and Applications.* Cambridge University Press, January 2003.

[4] S. Bechhofer, I. Horrocks, C. Goble, and R. Stevens. Oiled: a reason-able ontology editor for the semantic web. In *Proceedings of KI2001, Joint German/Austrian conference on Artificial Intelligence*, pages 396–408, Vienna, September 19-21, 2001. Springer-Verlag LNAI Vol. 2174.

[5] Dave Beckett. RDF/XML Syntax Specification (Revised). Technical report, W3C, http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/, February 2004.

[6] Roberto Chinnici, Martin Gudgin, Jean-Jacques Moreau, Jeffrey Schlimmer, and Sanjiva Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. Technical report, W3C, http://www.w3.org/TR/wsdl20/, November 2003.

[7] John Cowan. Extensible Markup Language (XML) 1.1. W3C Candidate Recommendation 15 October 2002. Technical report, W3C, http://www.w3.org/TR/2002/CR-xml11-20021015/, October 2002.

[8] Anupriya Ankolenkar et al. Daml-s: A semantic markup language for web services. In *Proceedings of 1st Semantic Web Working Symposium (SWWS' 01)*, pages 411–430, Stanford, USA, August 2001. Stanford University.

[9] Bill Shannon et al. Java 2 Platform Enterprise Edition Specifiaction, v1.4. Technical report, Sun, http://java.sun.com/j2ee/j2ee-1 _4-fr-spec.pdf, November 2003.

[10] David Martin et al. OWL-S: Semantic Markup for Web Services. Technical report, Daml consortium, http://www.daml.org/services/owl-s/1.0/owl-s.pdf, February 2004.

[11] Tom Bellwood et al. UDDI Version 2.04 API Specification. Technical report, Oasis, http://uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.htm, July 2002.

[12] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext Transfer Protocol – HTTP/1.1 . Technical report, June 1999.

[13] Charles L. Forgy. The rete algorithm. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[14] Ernest Friedman-Hill. *Jess in Action.* Manning Publications Company., October 2003.

[15] Michael R. Genesereth and Richard E. Fikes. Knowledge interchange format version 3.0 reference manual. Technical report, Stanford University, 1992.

[16] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification.* Addison-Wesley Publishing Company; 2nd Edition, June 2002.

[17] Hugo Haas. Web Services activity statement. Technical report, W3C, http://www.w3.org/2002/ws/Activity, 2001.

[18] Ian Horrocks, Frank van Harmelen, and Peter Patel-Schneider. DAML+OIL. Technical report, Daml Consortium, http://www.daml.org/2001/03/daml+oil-index.html, March 2001.

[19] Jason Hunter and Brett McLaughlin. JDOM API Javadoc specifiaction. Technical report, http://www.jdom.org/docs/apidocs/, 2004.

[20] Ian Jacobs. About the World Wide Web Consortium (W3C). Technical report, W3C, http://www.w3.org/2003/01/Consortium.pdf, March 2000.

[21] Joseph Kopena and William Regli. DAMLJessKB: A tool for reasoning with the semantic web. In *IEEE Intelligent Systems*, volume 18, pages 74–77, May/June 2003.

[22] Lei Li and Ian Horrocks. A software framework for matchmaking based on semantic web technology. In *Proceedings of the Twelfth International World Wide Web Conference (WWW 2003)*, pages 331–339, ACM, 2003.

[23] Brian McBride. Jena: Implementing the RDF Model and Syntax Specification. In *Proceedings of the Semantic Web Workshop (WWW2001)*, Hongkong, China, May 2001.

[24] Eric Miller. Semantic web activity statement. Technical report, W3C, http://www.w3.org/2001/sw/Activity, 2003.

[25] Nilo Mitra. Soap version 1.2: Primer. w3c recommendation 24 june 2003. Technical report, W3c, http://www.w3c.org/TR/2003/REC-soap12-part0-20030624/, June 2003.

[26] Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, and Katia Sycara. Semantic matching of web services capabilities. In *Forthcoming in Proceedings of the 1st International Semantic Web Conference (ISWC)*, pages 333–348, Sardinia, Italia, June 2002. IEEE.

[27] Dennis Sosnoski. Apache Axis SOAP for Java. In *Java-XML SIG*, Seattle, October 2002.

[28] Zuo Zhihong and Zhou Mingtian. Web ontology language OWL and its Description Logic foundation. In *Proceedings of the 4th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT 2003)*, pages 157–160. IEEE, August 2003.

Appendix A

List of Abbreviations

API	Application Programming Interface
DAML	Darpa Agent Markup Language
DAML-S	Darpa Agent Markup Language for Services
DAML+OIL	Darpa Agent Markup Language with Ontology Inference Layer
DL	Description Logics
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
J2EE	Java 2 Platform, Enterprise Edition
JESS	Java Expert System Shell
JDOM	Java Document Object Model
KB	Knowledge Base
KIF	Knowledge Interchange Format
OWL	Ontology Web Language
OWL-S	Ontology Web Language for Services
RDF	Resource Description Framework
SN	Semantic Network
SOAP	Simple Object Access Protocol
SVO	Subject-Verb-Object Form
SW	Semantic Web
UDDI	Universal Description, Discovery and Integration

W3C	World Wide Web Consortium
WS	Web Service
WSDD	Web Service Deployment Description
WSDL	Web Service Descriptions Language
XML	Extensible Markup Language

Appendix B

Abstract - German

Mit der Erscheinung des Semantic Web ist es jetzt möglich dass Web Resourcen und Inhalte eindeutig für Computerprogramme interpretierbar werden. Dies geschieht mit Hilfe von sogenannten Ontologien, die Konstrukte zur Konzeptualisierung und Klassifikation von Konzepten bieten, und eine Art semantisches Markup bereitstellen. Ein Standard namens DAML-S ermöglicht nun die Anwendung von Ontologien auf Web Services. Diese Arbeit untersucht auf welche Art und Weise dieses semantische Markup der Automatisierung und Interoperabilität zwischen Web Services nützlich sein kann. Nachdem wir die Grundlagen erforscht haben, entwickeln wir einen sogenannten Matching Algorithmus, welcher die Semantiken von zwei gegebenen Web Services untersucht und daraufhin die Kompatibilität zwischen diesen feststellt.
Weiterhin werden Anwendungsgebiete erforscht, speziell die Integration in existierende Infrastrukturen. Abschliessend wird die konkrete Implementierung untersucht, und anhand eines einfachen Beispiels wird die Funktionsweise illustriert.