Using Semantic Web Technologies for Matchmaking Software Agents Representing Web Service Description

Amer S. Al-Shaban

A Thesis
In
The Department
Of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

August 2005

ABSTRACT

Using Semantic Web Technologies for Matchmaking Software Agents Representing Web Service Description

Amer Shaban Al-Shaban

The recent growth of using agents in representing web services is causing difficulties in finding specific types of services. This problem usually arises because matchmaking techniques for services are often based on string comparison and service providers might neglect to provide enough or appropriate keywords for the matchmaking process. In this thesis, we report on an approach that makes use of formal ontologies and automated reasoning services in order to improve the matchmaking process. The suggested approach is based on the Ontology Web Language (OWL), the OWL reasoner RACER, and the agent framework DECAF. The use of OWL ontologies is twofold.

First, ontologies were used in order to express the particular knowledge of agents. These ontologies are grounded by referring to a so-called common upper ontology providing the necessary glue between the different agent domains. Second, with the help of OWL-S, a standard OWL ontology designed for specifying service descriptions, agents describe formally their offered web services. Our approach depends on a middle-ware agent called matchmaker, which will be in charge of matching required services to proper provider agents. Due to the use of OWL ontologies, the matchmaking process can be reduced to query processing and ontology reasoning implemented by the RACER system. The suggested approach has been demonstrated using a bioinformatics scenario, where several agents will take care of representing several web services. These agents will be providing composite services that the biology scientists might need. The communication protocol is based on OWL-S and allows seeker agents to adapt smoothly to dynamically changed web service descriptions of provider agents.

Acknowledgements

I would like first to express my gratitude to my supervisor Professor Volker Haarslev for without his inspiration, patience and vast knowledge and experience this thesis would not have been possible. I owe a great deal to my supervisor Professor Volker Haarslev for his confidence in me.

I would like to extend my thanks to my family in general and my mother and father in specific, who provided the item of greatest worth - opportunity. Words can not express my gratitude. Thank you for always standing by me and being there to support me.

Finally, I would like to thank all my friends and especially Ali Haidar, Osama Abdel-Mannan and Abdullah El Shazly for not only helping me in technical matters, but for their friendship and the good times we shared. Thank you all.

Amer Al-Shaban

Concordia University,

2005

To my mother and father for raising the six of us and being the parents they are…

… with all my admiration.

Table of Contents

LIST OF FIGURES.....VIII

1. INTRODUCTION 1

1.1. Description Logic 2
1.2. Semantic Web 3

2. PROBLEM 5

2.1. DECAF 5
2.2. Introduction To Agents 6
2.3. Existing Matching Techniques 8
2.3.1. UDDI 8
2.3.2. CORBA/ODP 11
2.4. Bottleneck Of Agents With Web Services 13

3. TOOLS.....15

3.1. RACER 15
3.2. OWL 18
3.3. Web Services 22
3.4. OWL-S 24
3.5. DECAF 26

4. SEMANTIC WEB IN WEB SERVICES 36

4.1. Ontologies 36
4.2. Matchmaking 39

5. DESIGN AND IMPLEMENTATION 45

5.1. Introduction To Plan Files 45
5.2. Role Of Matchmaker 49
5.2.1. Advertisement 53
5.2.2. Asking 54
5.2.3. Deeper 56
5.3. Grounding Technique 57
5.4. Role Of Provider Agent 60
5.5. Role Of Seeker Agent 63
5.6. Scenario.....66
5.7. Querying Technique Using nRQL and ConceptBased Queries.....67
5.8. Use of Java 71
5.9. Using API Of OWL-S For Execution 71
5,10. JRacer 74

6. FUNGAL WEB APPLICATION SCENARIO ……..77
6.1. Scenario **……..……77**

7. RELATED WORK ……..……...80

7.1. Configurable Matchmaking Framework for E-Marketplaces **……..……80**
7.2. Description Logics for Matchmaking of Services. **……...……82**
7.3. Software Framework For Matchmaking Based on S.W Technology**…...84**
7.4. Ontology Supported Intelligent Information Agent **……...86**
7.5. Intelligent Search Agent System For Semantic Information Retrieval.**87**

8. CONCLUSION ……..……90
8.1. Conclusion**……..……90**
8.2. Future Work**……..……...91**

9. REFERENCES ……..92
APPENDIX A: OWL-S API……97
APPENDIX B: FUNGAL WEB APPLICATION SCENARIO……100

List of figures

Figure 1.1: Description logic constructs. **2**

Figure 1.2: Example taxonomy **3**

Figure 2.1: UDDI's core data types **10**

Figure 2.2: Sequence diagram of CORBA matchmaking **11**

Figure 3.1: T-Box axioms **16**

Figure 3.2: RDF example **19**

Figure 3.3: OWL concept example. **20**

Figure 3.4: OWL individual example **21**

Figure 3.5: Uniting disparate systems via web services **24**

Figure 3.6: Top level of the service ontology **26**

Figure 3.7: DECAF architecture view **30**

Figure 3.8: Example of an ANSQuery **32**

Figure 3.9: Example of a plan file **33**

Figure 3.10: Example of an agent initialization using the GUI **34**

Figure 4.1: Example of an ontology taxonomy **37**

Figure 4.2: Protégé's user interface. **38**

Figure 4.3: String comparison matchmaking scenario **40**

Figure 4.4: Matchmaking Scenario **42**

Figure 5.1: Plan file example. **46**

Figure 5.2: Plan file for the printing example. **47**

Figure 5.3: Sequence diagram illustrating the agent communication **48**

Figure 5.4: Sketch of an upper ontology **52**

Figure 5.5: Plan file of the Matchmaker **53**

Figure 5.6: Sequence diagram for the upper ontology **59**

Figure 5.7: Plan file of provider agent. **63**

Figure 5.8: Seeker's plan file....65
Figure 5.9: ConceptBasedQuery example....69
Figure 5.10: Description logic format of the query....69
Figure 5.11: nRQL example. 70
Figure 5.12: OWL-S ontology....73
Figure 5.13: Example of using JRacer....74
Figure 5.14: Example of the send method in JRacer....76
Figure 6.1: The scenario of the fungal web project....78
Figure 7.1: Schema for Trading Intentions for Cars....81
Figure 7.2: Example of searching for a product in a certain trading 82
Figure 7.3: Service description branch of the subsumption tree....84
Figure 7.4: High level system architecture....88
Figure A.1: Example of OWL-S API....97
Figure B.1: Upper ontology....100
Figure B.2: The provider agent's ontology. 102
Figure B.3: OWL-S API Tool 103

1. Introduction

Web services have received massive attention in the internet world. The more web services exist, the more it becomes harder to find the right web service for the right user especially because web services are not centralized. That is why software agents were used to help in finding the right web services. Yet, the matchmaking techniques used by the current software agents could be improved.

In this thesis, existing matchmaking techniques are discussed and some of their issues are explained. Then the suggested solution is presented and the implementation for that solution is explained.

The structure of this thesis is as follows: First, an introduction of description logic and semantic web is given in this chapter. Afterwards, a detailed explanation of the problem is given in Chapter 2 along with an introduction of the agent's framework (DECAF). Chapter 3 introduces the tools used for the suggested solution. Later on, the contribution of the thesis is explained in Chapter 4 after explaining the components of the semantic web. In Chapter 5 the design and implementation of the contribution is explained in detail and a real world implementation is presented in Chapter 6.

The major contribution of this thesis is the implementation of a semantic web matchmaking system that uses description logics in order to match a seeker with a provider agent that represents a web service. After finding the right provider agent, the seeker agent executes the web service and returns the results.

1.1 Description Logic

Description Logics [13] are a family of knowledge representation languages which can be used to represent the terminological knowledge of an application domain in a structured and formally well-understood way. Description logics are based on the notion of concepts (classes), roles (properties) and individuals (instances). These terms will be used interchangeably throughout the text. Concepts describe the common properties of a collection of individuals which are interpreted as sets of objects. Roles are interpreted as binary relations between these objects.

In order to define new concepts and roles, a set of language constructs has to be defined and used in each description logic language; some of these constructs are intersection, union, negation…etc. as shown in Figure 1.1[13].

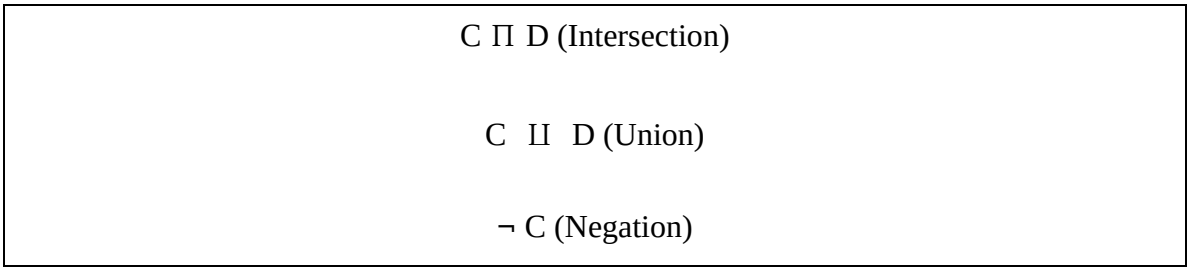

Figure 1.1: Description logic constructs. [13]

The main reasoning tasks are classification and subsumption. Subsumption, which is

written as **C** $\subseteq$ D, represents the **is-a** relation. Determining subsumption is about checking if the concept denoted by the subsumer (D) is considered more general than the one denoted by the subsumee (C). Classification is the computation of a concept hierarchy based on subsumption, see Figure 1.2.

A whole family of knowledge representation systems has been built using these languages [13] and for most of them complexity results for the main reasoning tasks are known. Description logic systems have been used for building a variety of applications [13].

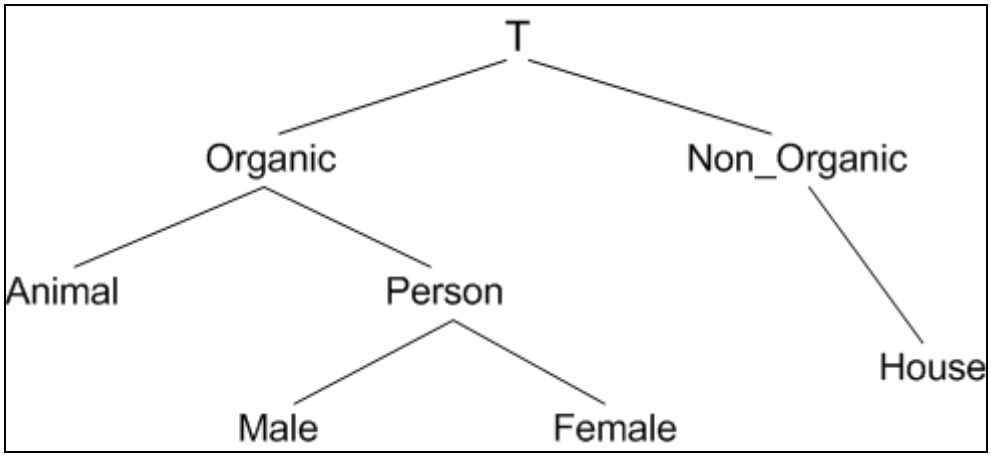

Figure 1.2: Example taxonomy

1.2 Semantic Web

"The Semantic Web is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation."

-- *Tim Berners-Lee* [51]

The Semantic Web was suggested in order to have the information on the web related in a way that it can be interpreted and used by machines. In this way this information will be understandable by machines instead of the current limited demonstration purpose for web users.

The Semantic Web can be thought of as an efficient way of representing data on the World Wide Web, or as a globally linked database [50].

The main goal of the research on the Semantic Web is to move the web from being only human understandable, to being both human and machine understandable.

The Semantic Web is generally built on syntaxes which use three URIs (Uniform Resource Identifier) to represent data. For example, many triples of URI data that can be held in databases, or interchanged on the web using a set of particular syntaxes developed especially for the task. These syntaxes are called "Resource Description Framework" (RDF) syntaxes [50].

Currently the information on the internet is considered as a series of characters and symbols for the machine. But by applying the Semantic Web on the internet the ability of having the computers “understand” the information while parsing it might become available. This task can be facilitated by using a special mark-up language such as OWL (Ontology Web Language).

2. Problem

2.1 DECAF

DECAF, Distributed, Environment-Centered Agent Framework [40], is a toolkit that allows a well-defined software engineering approach to building multi-agent systems.

This toolkit offers a stable framework in order to allow users to design, rapidly develop and execute intelligent agents in order to find a solution for complex software problems.

DECAF gives the user many services that make it possible to manage a large amount of agents easily. The services are as follows: communication, planning, scheduling, execution, and monitoring, which will be explained in Chapter 3.

DECAF could be considered as the internal operating system of the software agents because it provides the services that are provided by any operating system to software applications.

The user can control the behavior of an agent through a text based map called the plan-file. The plan-file could be created using a GUI application provided by DECAF called the plan-editor, which will also be explained in Chapter 3. The activities of these agents could contain loops and if-then-else behavior. This part is an extension of the RETSINA and TAEMS task structure frameworks [40].

The main goal of creating this architecture was to allow quick and easy development of third-party domain agents. This will lead to full multi-agent solutions. DECAF is fully based on Java and it took advantage of the object-oriented features of the language to make it easier for the user to create, edit and manipulate agents' behaviors.

What makes DECAF special is that it moves the development of software agents' a step higher; the philosophy of DECAF directs most of the effort on the behavior of the agent rather than the underlying components such as the message formatting, communication protocol and sockets creation.

Another advantage of DECAF is that it does not require from users a lot of Java network programming knowledge in order to send messages between the agents, or to be a Java database expert in order to deal with the knowledge-base of the framework. What makes this possible is the fact that DECAF handles all these small low level details and presents them by a high level definition so the user can handle the communication between agents by using simple communication protocols provided by DECAF.

Further details of DECAF are discussed in Chapter 3.

2.2 Introduction to agents

Giving an exact definition of agents in general is a challenging task. Nevertheless agents could be defined as entities that act on behalf of the user, have the capabilities of following certain schema in order to accomplish a certain goal. In the case of mobile

agents, they are considered as software agents that can move between locations. This definition implies that a mobile agent is also characterized by the basic agent model. In addition to the basic model, any software agent defines a *life-cycle model*, a *computational model*, a *security model* and a *communication model* [7]. In order for these agents to run on a certain machine, that machine needs a standardized framework that would include the capability of all the possible activities that the agent might need. These services could be categorized into three facilities. First, the life-cycle facilities, which include the services to create, destroy, suspend, stop and perform other related services. Second, the computational facility, which takes care of the computations that the agents might need such as data mining, database access, I/O action, etc. Finally, the communication facilities are needed because agents communicate with each other and with different services; therefore a framework that would allow certain agents to communicate directly or indirectly without having any security problems is necessary [7].

The main characteristic of mobile agents is mobility. Mobility means that the agent can move from a machine to another one. This facility could be achieved by any remote object, but what makes agents a better solution is the efficiency of the agent framework in the following aspects: The CPU consumption will be reduced with mobile agents, since the agent object will be executed only at one machine when needed without having to consume the CPU of the sender machine. Also the resources consumption will be reduced because it works on one node only where, on the other hand, if multiple server method is being used, then the functionality of the agent would be required to reside on both sides until it is accomplished. Regarding the network traffic, using mobile agents will require a small amount of data transfer because the agent's code is usually smaller than the data

that it processes, especially while using KQML (which is explained in Chapter 3). Taking into consideration the many causes of network failure, it will be clear that the robustness and fault tolerance is an important issue when dealing with mobile objects. That is why dealing with mobile agents would give the user the control on actions that the agents would take to reduce the probability of failing the task, which means fault detection will be improved by using mobile agents. One of the most important issues in the web is the different platforms inconsistency. By using mobile agents, which are based mostly on Java, all agents' frameworks will actually work on any platform regardless of the lower architecture. The last advantage would be the software upgrade. Upgrading a mobile agent virtually is an easy thing to do. On the other hand trying to swap functionalities on servers is difficult [11]. So these potential benefits show the importance of having mobile agents in many categories instead of the regular server functionalities.

2.3 Existing Matching Techniques

2.3.1 UDDI

The main idea behind the web services revolution is that the web will be populated with an assortment of small pieces of code, all of which can be published, found, and invoked across the Web. One key technology for the service-based Web is SOAP [52], the Simple Object Access Protocol. Based on XML, SOAP allows an application to interact with remote applications. The main question is how these applications could be found in order to be used.

One of the suggested solutions for that problem was the UDDI [41]. UDDI stands for Universal Discovery, Description and Integration protocol. It is a protocol that provides three essential functions. These functions are publishing, find and bind.

- *Publish:* This function allows a web service provider to register itself in order to be searchable.
- *Find:* The find function allows an application to search for a particular web service.
- *Bind:* It provides the ability for an application to connect and interact with a known web service.

The UDDI categorizes the available information into three main categories, just like telephone directories, white pages, yellow pages and green pages. The white pages include contact information for a given business such as the name, address, telephone number, fax …etc. The yellow pages contain information that arranges business types into different categories. The green pages contain the technical information of web services.

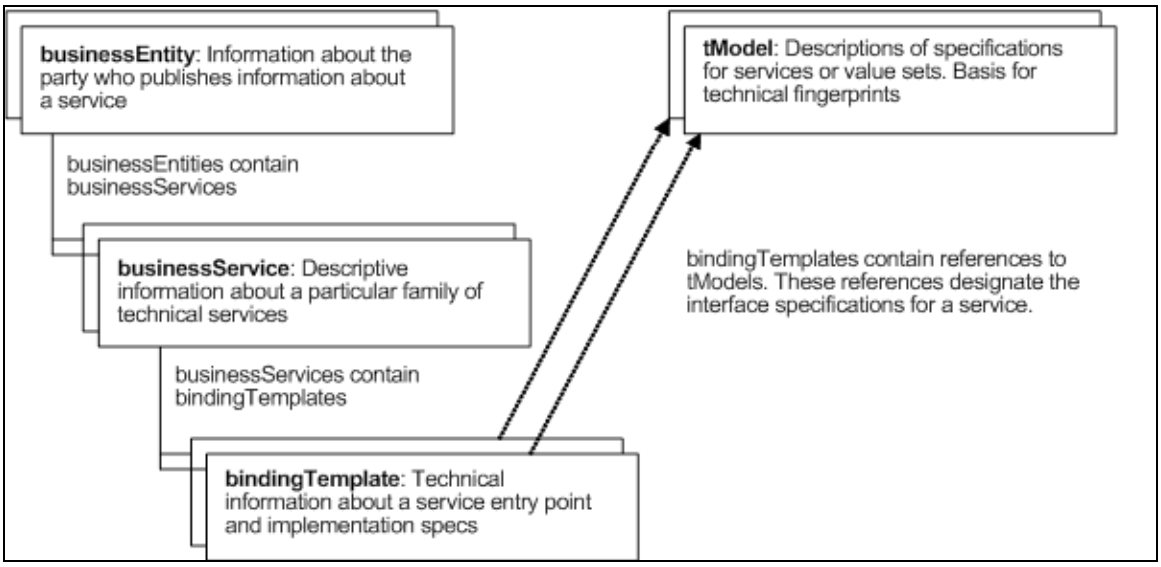

Figure 2.1: UDDI's core data types [42]

UDDI has four defined information types that could be used by users and applications in order to use a web service. These types are shown in Figure 2.1 taken from [42] and they are as follows:

- *Business information*: Contained in `BusinessEntity` object. It contains information about services, categories, contacts, URLs, and other things necessary to interact with a given business.
- *Service information*: It describes a group of web services. Service information is contained in `BusinessService` object.
- *Binding information*: It describes the necessary technical details to call a web service. This includes the address of the web service, method names, input parameters and their types, and so on. And it is represented by the object `BindingTemplate`.

- *Services' specification*: It is metadata about the implemented specifications in a given web service. It is represented by the `TModel` object.

2.3.2 CORBA/ODP

Another approach for solving the matchmaking problem is the ODP/CORBA [43, 44]. ODP/CORBA is used when the seeker does not know all the details of the server object, but knows some of the properties that describe it.

In CORBA, the provided services are usually described through the properties that the service uses. Properties have two types, either static or dynamic. Static properties are fixed at advertisement time, while the dynamic properties are updated at runtime whenever requested.

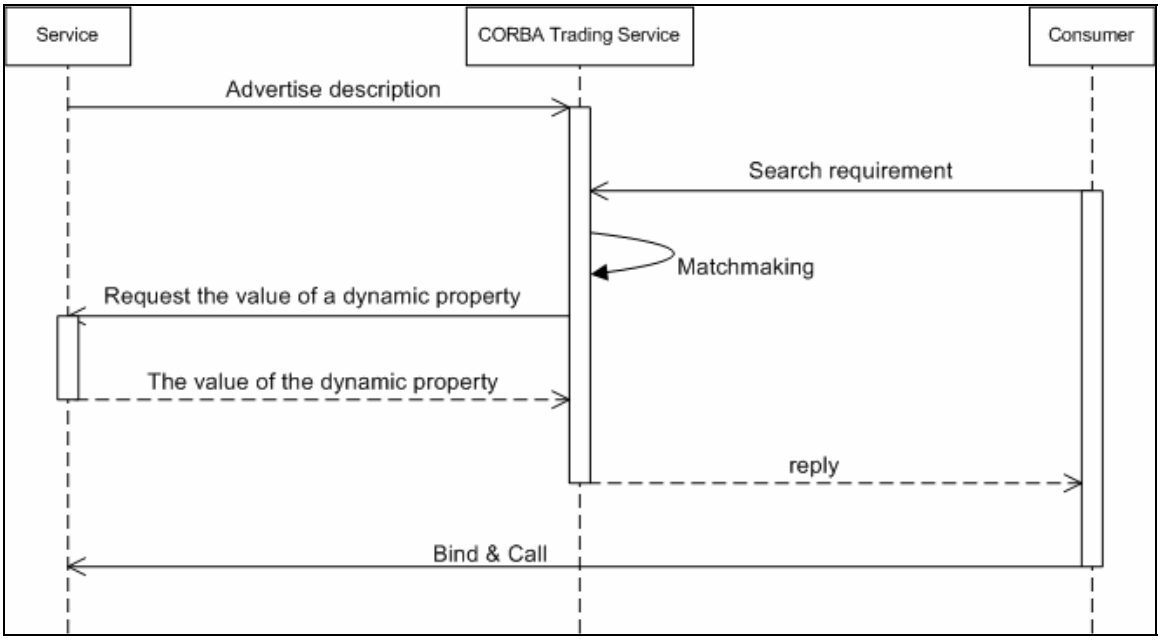

Figure 2.2: Sequence diagram of CORBA matchmaking

The procedure of the CORBA matchmaking is shown in Figure 2.2 and is explained as follows.

First, the provider advertises its service to the Match Maker Engine (MME). Then, whenever the consumer seeks a service, a search request is sent to the MME which invokes a constraint expression. When the MME receives the search request it evaluates the requirements. If the constraint expression included a dynamic property, then, a request asking for an updated value of that dynamic property is sent to the service provider. Then, the value is sent back to the MME in order to proceed with the evaluation of the constraint expression. When the MME finishes the evaluation, the reply is sent back to the consumer. Finally, the consumer has to choose a service provider in order to bind and call the service.

The CORBA trading service provides [45]:

- Asymmetric matchmaking: Where the provider publishes properties that describe the service and the consumer specifies constraint expressions in order to find the required information.
- Limited language for creating properties: the CORBA trading service supports five data types which are, integer, float, string, char and boolean.
- Language for writing constraints: The simple constraint expressions that return either a true or false answer consist of logical expression (e.g. AND, OR, NOT, <,>, ==, string …etc) and arithmetic expressions (e.g. *, /, -, +).

2.4 Bottleneck of agents with web services

As mentioned in Chapter 1, the usage of agents for matchmaking in general and for representing web services in particular is expanding. Unfortunately the matchmaking techniques that are used for finding the right web service would follow one of the techniques mentioned in the previous section. The problem with these procedures is that they are all based on string comparison. Therefore, these strings do not contain any meaning for agents. That might lead to incomplete and inaccurate results while the main point of creating agents that represent web services is to improve the quality of searching. An example will be given in order to clearly explain the problem.

Having a salesperson named John who wants to find Miss Cook that he met last year in a conference. If John followed any of the existing matchmaking techniques then he will most probably find irrelevant results. He will have information about cooking, cookies, Miss Cook Islands…etc. And John might not even find the right piece of information that will lead him to the right Miss Cook. This exemplifies the problem of string comparison. On the other hand, if John uses a search technique based on the semantic web approach, where he has to provide a concept definition for Miss Cook by stating that Miss Cook is a female person who has the name “Cook”, who is not married …etc, then, the results will be much more accurate and the searching agent will be able to avoid the irrelevant information.

Beside the problem of search quality, using string comparison for searching might cause a bottleneck. As an example, if a system has a thousand agent with 1000 keyword each

then in order to cover the thousand agents a procedure of comparing strings might be called one million time which will slow down the matchmaking. The reason of such a bottleneck is that the million keywords are not structured or pre-computed, therefore every time an agent looks for a service it has to search through the list. Where in Semantic Web the concepts are pre-computed and related with each other in a logical way that would avoid going through all the concepts.

3. Tools

3.1 RACER

RACER, which stands for Renamed ABox and Concept Expression Reasoner, is a description logic reasoner that provides a set of inference services. The most important inference services are listed as follows [46]:

- Concept consistency w.r.t. a TBox: iff there exists a model of C that is also a model of the T-Box containing C.
- Concept subsumption w.r.t. a TBox: Checks if there is a subset relationship between the set of objects described by two concepts.
- Find all inconsistent concepts mentioned in a TBox.
- Determine the parents and children of a concept w.r.t. a TBox: The parents of a concept are the most specific concept names mentioned in a TBox which subsume the concept. The children of a concept are the most general concept names mentioned in a TBox that the concept subsumes. Considering all concept names in a TBox the parent (or children) relation defines a graph structure which is often referred to as taxonomy.

In addition, RACER provides reasoning services for multiple T-Boxes and ABoxes [4].

A **T-Box** can be considered as a set of definitions of *concepts* that describe a domain under certain restrictions. These definitions are based on logical meanings and they are constructed using axioms as seen in Figure 3.1. The concepts are related with each other by subsumption, so it would be possible to reason about the terminologies on the basis of their logical meaning. [13]

$$\text{Woman} \equiv \text{Female} \sqcap \text{Person}$$

$$\text{Man} \equiv \text{Person} \sqcap \neg\, \text{Female}$$

Figure 3.1: T-Box axioms

An **ABox** is a set of assertions about named individuals referring to defined concepts in the T-Box. There are two kinds of assertions. The first kind is a concept assertion, where one states that an instance *a* belongs to the concept *C.* The second kind is a role assertion, where one states that an instance *a* fills the role *R* for the other instance *b.* For example, if there is a woman called LAURA, then the concept assertion would be as follows.

Woman(LAURA)

This states that LAURA is an instance that belongs to the concept Woman. If LAURA has a child called JOHN then there will be a relationship between each other using the property hasChild, and the role assertion is:

hasChild(LAURA,JOHN)

The following inference services are defined according to [46]. Concept consistency of C means iff there exists a model of C that is also a model of the T-Box containing C. An ABox *A* is consistent w.r.t a T-Box *T* iff *A* has a model *I* which is also a model of *T*. If an ABox or a T-Box is not consistent then it is called *inconsistent.*

RACER also provides several features that help to keep description logic representations consistent and reasonable. Whenever a T-Box states a subsumption relationship between two concept terms RACER takes two actions. First, it checks if the concepts are consistent with respect to the T-Box. Second, it determines the parents and children of the concepts in the T-Box. Also, when an ABox is given, it checks the consistency of the ABox with respect to the T-Box and confirms that they do not cause a contradiction. It applies an instance test based on the given ABox and on the existing T-Box, and it also computes the fillers of a role assertion referring to an individual. These are only the main services that RACER offers. It has to be mentioned that there exist many other actions that might be involved during successfully loading a T-Box and an ABox.

One of the most important features of RACER is the ability of querying ontologies that are based on the RDF format, which is described in Chapter 4, in order to retrieve the result of RACER's reasoning. Since RACER acts like a TCP server, these queries can be processed by opening a TCP port to the engine, streaming the commands, and then getting the reply back after RACER has finished the reasoning on the set of assertions already loaded in the RACER. The queries could query the ABox or the T-Box depending on the type of query, which will also be discussed in Chapter 5.

3.2 OWL

The Semantic Web could be seen as a vision for a web where information is given specifically with an explicit meaning associated with it. These meanings are represented in a way that would make it possible for machines to understand these meanings and be able to take actions based on them. In order for this information to be available with these features, it has to be described by a language capable of reflecting the semantics along with the concept names. Many languages were implemented for this purpose, and recently OWL was introduced by the W3C as a standard for knowledge representation in the Semantic Web.

OWL, which stands for Ontology Web Language, is a semantic markup language for publishing and sharing ontologies on the World Wide Web [14]. It is an extension of the growing standards of W3C which are RDF and RDFS. The Resource Description Format (RDF) [15] and the Resource Description Format Schema (RDFS) [16] were considered as the presenting language for description logics in the semantic web. They are based on the structure of XML tagging as seen in Figure 3.2. This makes it easy for machines to locate the required piece of information and simplify the parsing process. RDF/RDFS came with several obstacles. RDFS is too weak to represent resources as concepts and roles in adequate details. Further, it is difficult to provide reasoning support for the language (i.e. RDFS), which is important in order for the user to use the language and apply its capabilities. The basis for not supporting reasoning is that RDFS has a non-

standard semantics. Therefore, two languages were implemented that solve the problems that RDF/RDFS had, and these languages are OIL and DAML [17].

```
<rdf:Description rdf:about="http://www.w3schools.com/RDF">
      <si:author>Jan Egil Refsnes</si:author>
      <si:homepage>http://www.w3schools.com</si:homepage>
</rdf:Description>
```

Figure 3.2: RDF example

Later, DAML and OIL's features were combined into producing DAML+OIL. DAML+OIL was built from the original DAML ontology language DAML-ONT (October 2000) in an effort to combine many of the language components of OIL [18]. Then, finally the WebOnt group, the World Wide Web's Semantic Web Activity Working group on Web Ontology Language, made the effort of creating OWL based on the existing DAMIL+OIL.

OWL has three sublanguages, OWL Lite, OWL DL and OWL Full. These sublanguages are designed in order to be used by certain users and implementers [14].

- *OWL Lite* was designed for users that are mainly interested in classification hierarchies and need simple constraints only. As an example of the simple constraints that OWL Lite provides is the cardinality feature: it does not allow any value different than zero or one in order to make reasoning as simple as possible.

- *OWL DL* was designed for users who are in need of expressiveness yet want to keep computational completeness and decidability. According to [14], OWL DL includes all OWL language constructs with restrictions such as type separation. This means that a

class can not be also an individual or property, a property can not also be an individual or class. OWL DL was named due to its correspondence with description logic.

• *OWL Full* was designed for users who need the expressiveness and the freedom of RDF without any kind of description logic constraints but with no computational guarantees.

In order to make ontologies easier for the user and the implementer, OWL adopted the notion of classes and properties, which will be explained further in Chapter 4. Consequently, if one wants to declare an instance of concept A that is related to an instance of concept B, then simply a concept A would be declared as a class, and a property that relates individuals of concept A with individuals of concept B would be declared in an XML format, as mentioned before, like the example in Figure 3.3. Then, an instance of Concept A and Concept B could be declared.

```
<owl:Class rdf:ID="ConceptA" />
<owl:Class rdf:ID="ConceptB" />

<owl:ObjectProperty rdf:ID="hasRelation">
        <rdfs:domain rdf:resource="#ConceptA"/>
        <rdfs:range rdf:resource="#ConceptB"/>
</owl:ObjectProperty>
```

Figure 3.3: OWL concept example.

As shown in the example, the tag <owl:class rdf:ID=”ConceptA”> is actually declaring the concept A as a class, the same for the concept B, afterward the <owl:ObjectProperty_ rdf:ID=”hasRelation” > declares the property that would relate both individuals of concept A and individuals of concept B by specifying the domain, which is the left hand side, and the range, which is the right hand side, of the property relating individuals.

By declaring the class in OWL, this means a concept in a T-Box has been declared in terms of description logics. OWL also is capable of representing an ABox as well, and this is achieved by creating individuals from the type of the classes. For example, in order to create an individual “A” of “ConceptA”, create individual “B” from “ConceptB”, and state the relation between them. The definition in OWL syntax is shown in Figure 3.4.

```
<ConceptB rdf:ID=”B” />
<ConceptA rdf:ID=”A”>
     <hasRelation rdf:resource=”#B”/>
</ConceptA>
```

Figure 3.4: OWL individual example

Due to this method, a complete knowledge base scenario can be created based on both the T-Box and the ABox and represented it in OWL, so it will be readable and understandable by machines. Further, OWL is based on XML tag formats in order to be transferred from one place to another.

3.3 Web Services

A Web service is a software system identified by a URI, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the web service in a manner prescribed by its definition, using XML based messages conveyed by internet protocols [19].

Many organizations and firms are using web services in their daily life. It has been widely employed especially in the last two to three years. There are many practical and commercial applications based on web services that are being used by either average users or companies. At Xmethods, [20] they provide a free web service which gives a stock quote. The users simply have to put it in the symbol code, the web service processes the request, and returns the results in a XML format. As mentioned earlier, web services in the commercial field are growing dramatically. GE Global eXchange Services [21] are supporting e-commerce transactions for small and medium sized companies by providing web services capabilities. These web services will assist those companies economically by saving cost and time. Since these business documents will be sent automatically via the internet without any delay, it confirms again how web services are taking a big part of the business field. Finally, one of the most important users of web services are agents. The main concept of web services is to allow the machines to communicate with each other without any interaction from users. The combination of web services with agents will be more reliable and applicable because web services will

act on behalf of regular services, and agents will act on behalf of humans.

After discussing some of the major implementations, a big shift toward an early introduced technology, the first question that might come to mind is why web services? Firstly, web services use the already familiar URI [22], Uniform Resource Identifier, to perform endpoint mapping. Since web services are based on URI and XML, it gives it the feature of accessibility, as a result, regardless of which platform is being used, a user can simply issue a web service request and get the answer back via XML. Thus, this feature makes the clients of web services machine independent and more universal than the regular peer to peer application or server client architectures. The other advantage of using web services is the machine load. Since the web service that is requested is actually on another server, then the load of processing that service will not be on the user's machine. Consequently, if there are multiple tasks that might cause an overload, then distributing them by requesting different web services from different locations will resolve such an issue. Web services play a major role in software engineering development. Web services help in developing software for big projects by allowing software to use existing codes that are provided through web services instead of actually writing the code from scratch. In software engineering, the ideal vision of creating software is to be able to reuse as much as possible from existing programs and combine them to create the required application with less time and effort. Finally, when it comes to disparate systems that need to be united, web services could be used in order to achieve this. What is usually achieved is a web service created for each system, and then the other systems could access the services through the web. For example, in Figure 3.5, system D is communicating with system A through the web service via the internet

without the need of translating the commands to the commands of system A.

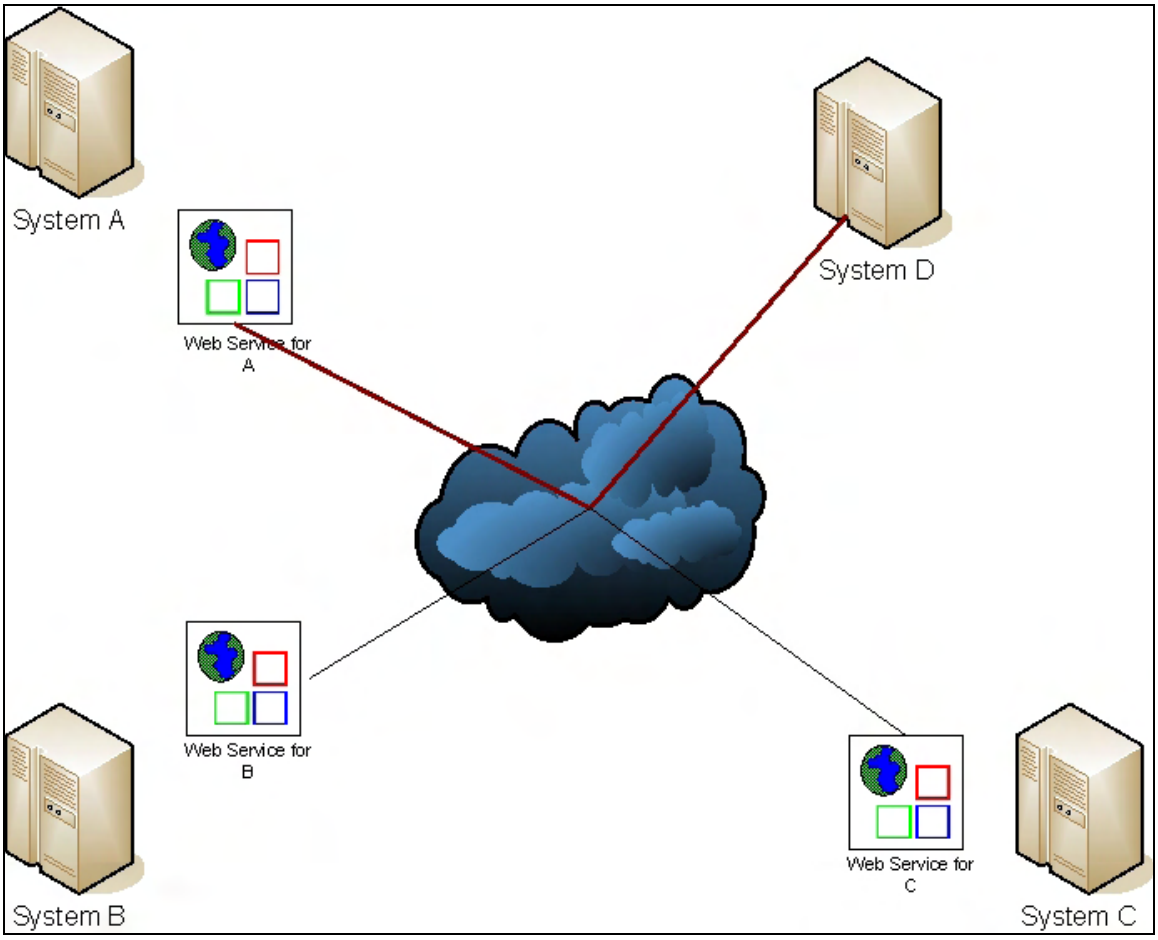

Figure 3.5: Uniting disparate systems via web services

3.4 OWL-S

The Semantic Web, as mentioned before, is not only providing access to the contents of the web but it also makes it possible to access web services that present certain tasks that can be done automatically. Ideally users and agents should have the ability to find,

compose and monitor those web services under the control of the semantic web. If desired, the agents should be able to execute these web services automatically too.

OWL-S is an OWL-based Web service ontology, which supplies Web service providers with a core set of mark-up language constructs for describing the properties and capabilities of their web services in an unambiguous, computer-interpretable form [23].

Web services are usually based on WSDL [24], which stands for Web Services Description Language, and WSDL operations are all typed as strings. So, whenever a user tries to search or execute a web service, he/she has to interpret the operations' types, input parameters, the description of the service and the returned type manually. The main reason for this is that the machine, obviously, can not use the terms on the basis of their semantics. On the other hand, OWL-S provides a language that specifies the function of the operation and the semantic types for the input and output parameters of the service. Therefore, if a software agent wants to deal with a web service, it would be able to distinguish between the size of a banana as an input and the size of an apartment which makes it possible to automatically search, execute and monitor the web services without requesting any information from the user.

After discussing what OWL-S is and how it is used, the following will describe the structure of it. OWL-S is an OWL ontology with three interrelated sub-ontologies; which are called the *profile*, process *model* and *grounding*. These sub-ontologies provide three essential types of knowledge about the service. The profile is mainly used to express and describe what the service actually does. This would be used for advertising, constructing service requests and for matchmaking purposes. The process model's task is to describe

how the service works and that will allow composition, monitoring and recovery of the service's processes. Finally, the grounding handles the constructs of the process model by mapping it onto the detailed specification of the protocol, which is normally expressed in WSDL. The combination of the three ontologies and their relationships is shown in Figure 3.6

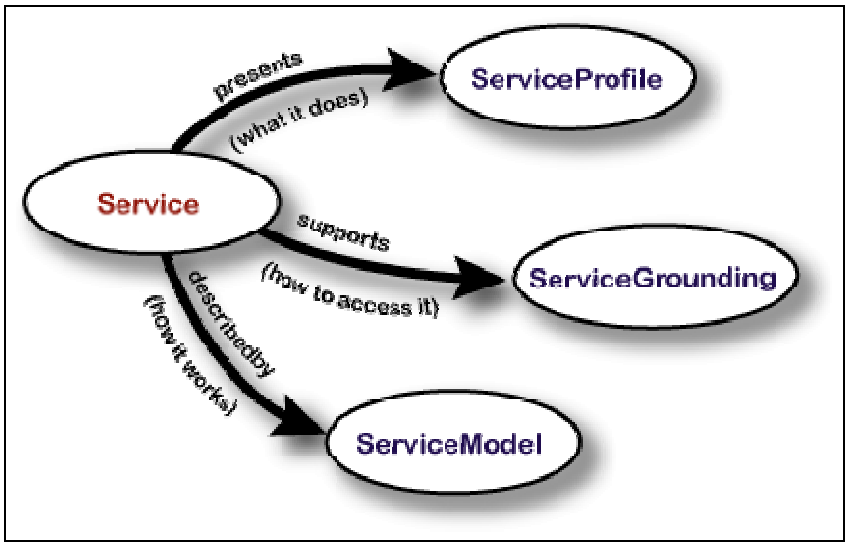

Figure 3.6: Top level of the service ontology [25]

3.5 DECAF

DECAF [26], as mentioned before, is a general purpose agent development platform which allows a well-defined software engineering approach to build multi-agent systems. DECAF provides a stable platform to design, rapidly develop, and execute intelligent agents to achieve solutions in complex software systems. DECAF provides the necessary

architectural services of a large-grained intelligent agent communication, such as planning, scheduling, execution monitoring, coordination, and eventually learning and self-diagnosis. [1]

DECAF consists of well defined control modules which are responsible for initializing, dispatching, planning, scheduling, execution, and coordination. These modules coordinate with each other in order to control the life cycle of agents. Further, each of these modules is executed as thread which makes it appropriate for applications that requires multithreading.

In this section each module of DECAF is explained, elaborating on the use of that module and how it functions [1]. Figure 3.7, gives a picture of the modules.

KQML, short for ***K****nowledge* ***Q****uery and* ***M****anipulation* ***L****anguage*, is a language and protocol used for information and knowledge exchange between programs/agents. Agents mainly use KQML for querying, stating, authenticating, requesting and subscribing.

The life cycle of agents in DECAF is explained in the following paragraphs.

a) Agent Initialization

As soon as an agent starts, the first module that will run is the initialization module in its own Java thread. It will run only once for that particular agent. The initialization module takes care of the plan file, which is explained in Chapter 5. It reads the plan file (as seen in Figure 3.7 ‘a1’), and adds each task reduction specified in there to the *Task Templates Hash table* (plan library, ‘a3’) which is a definition of the capabilities of this particular agent. Then it builds domain facts in a database or knowledgebase that might be needed

in the future by the agent (a2). It also adds the actions specified for that particular agent in a tree structure format.

b) Dispatcher

After the initialization module, the control will be passed to the dispatcher. The dispatcher will always be running in the background of the application waiting for an incoming KQML (which is also explained later in this section). As soon as any KQML message arrives, it will be queued in the *Incoming Message Queue* (b1). The dispatcher will take an action depending on the content of the KQML. First, if the KQML message tries to communicate as part of an existing conversation then it gets recognized through the in-reply-to field. In that case the dispatcher searches for the equivalent action in the *Pending Action Queue* (b3), then, it proceeds with the rest of actions for that agent. The second path shows that this KQML message initiates a conversation by not using the in-reply-to field. If so, then the dispatcher takes care of creating a new objective and placing it in the *Objectives Queue* (b2). Usually an agent has more than one active objective, but they are not all necessarily achievable.

c) Planner

Since the tasks in the *Objectives Queue* are received (c2), then a module will monitor them and attach new tasks to the existing task template stored in the plan library (c1). The planner module takes care of that. The instantiated plan will be having a copy at the *Task*

Queue area in the HTN format corresponding to that task (c3), along with a unique ID and any passed provisions from the KQML messages. If it happens that a request came for the same goal to be accomplished, then automatically the plan template will be instantiated in the task networks with a new ID. All the plans/goal structures that have to be accomplished will always be stored in the *Task Queue*.

d) Scheduler

The scheduler's main actions occur at the *Task Queue*. It stays idle until the *Task Queue* contains tasks (d1). Then it starts scheduling the tasks deciding which will be executed next, and in what order they should be executed. In order to do that the scheduler sends the tasks to the *Agenda Queue* (d2). The task of the *Agenda Queue* is setting the actions into executions. The execution part depends on the availability of the provisions for a particular module. Provisions could be available either from an incoming KQML message or, it could be given as an output from a different action. Therefore, what could be concluded is that every time a certain provision is given, the *Task Queue* is checked for any executable tasks. This part of the DECAF is capturing the attention of many researchers, where they want to add reasoning abilities for scheduling in order to select the optimal path for task completion.

e) Executor

Finally, the executor module monitors the *Agenda Queue* and starts when it is nonempty

(e1). As soon as an action is added to the queue, the Executor module executes the task. After the execution two things can occur. First, the action executes successfully and a result is returned which will be placed into the *Action Result Queue* (e3). The framework will distribute the result to downstream actions which might be waiting in the *Task Queue*. After this is accomplished, the executor will check the *Agenda Queue* searching for any other executable tasks. The executor will deal with tasks by having a thread for each action. The other path would be taken if the action is partially completed and requires further actions. In this case, the task is placed into the *Pending Action Queue* (e2) for further actions. If no such actions exist in the Pending Queue, an error message is returned to the sender.

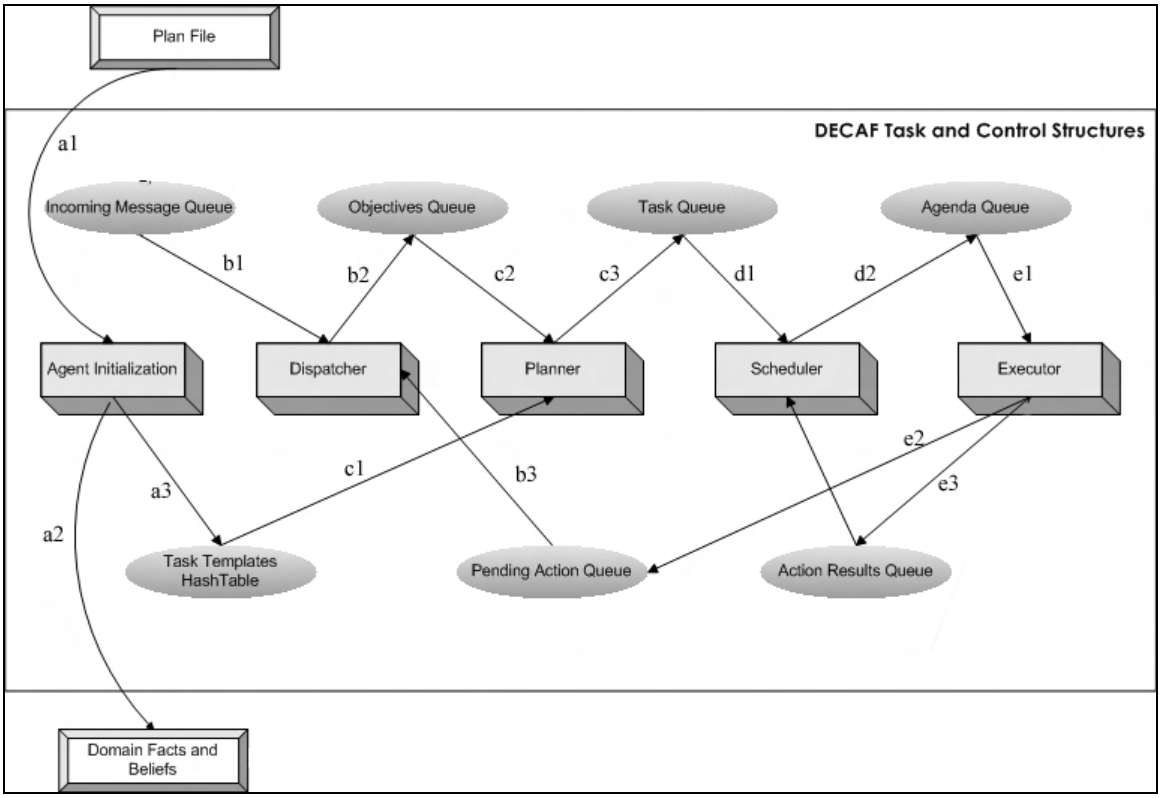

Figure 3.7: DECAF architecture view

DECAF manages the agents' actions in a way that is very similar to operating systems but with slight differences.

- The OS (operating system) handles jobs that have any type and any input where in DECAF all the jobs are agent task requests and the input is limited to KQML only.
- The OS must finish all the actions that are related to the required task but in DECAF that is not necessary.
- The OS does not have the full details of the job execution profile but this is well characterized in DECAF.
- The number and type of jobs are not known at start-up in the OS, while they are clearly stated and declared with a plan file at start time.

In order to register an agent and consider it fully running, three basic things are needed: ANS, Plan Files and the DECAF architecture itself. **ANS**, which stands for Agent Name Server, is a major component for the communication part of agents. It functions almost like the DNS, Domain Name Service, by resolving agent names to port addresses and host names, and it is also referred to as the "white pages" for the agents' world. In order to see what is registered on the ANS, the user needs a component that reflects whatever is happening at the ANS side. This component is called ANSQuery and provided by DECAF, an example of how the ANSQuery represents the ANS can be seen in Figure 3.8.

Figure 3.8: Example of an ANSQuery

The **Plan File** is the map that tells an agent what it has to do and what kind of reaction has to be created depending on the output of a certain task. Plan Files are explained in more detail in chapter 5 but in order to give an idea, an example of a plan file is shown in Figure 3.9.

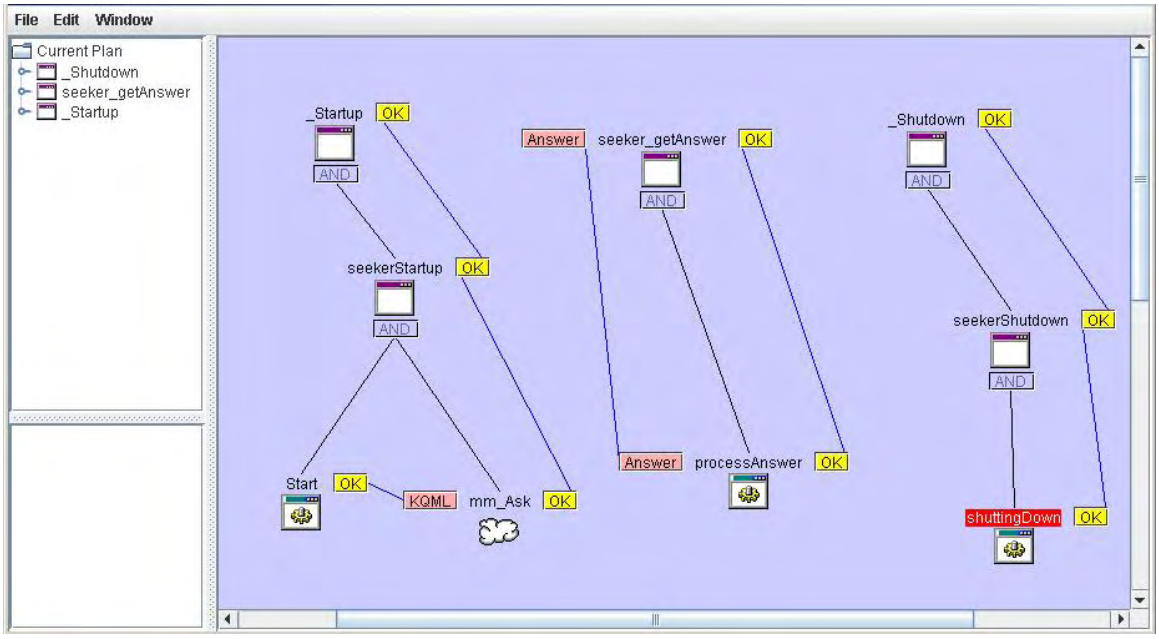

Figure 3.9: Example of a plan file

Creating an agent requires certain parameters and can be done in two ways. The first method of initializing an agent in DECAF is using a text based command that contains the required parameters. The second method is running the GUI version of DECAF and filling up the parameters through the GUI window as seen in Figure 3.10. The required parameters are the ANS host address, the ANS port which is usually 6677 (default value), Agent Name, Agent Host (which is the address of the local machine), Agent port (4000 as default value) and finally the plan file. After filling these parameters, the agent starts by itself and follows the tasks that are described in the plan file.

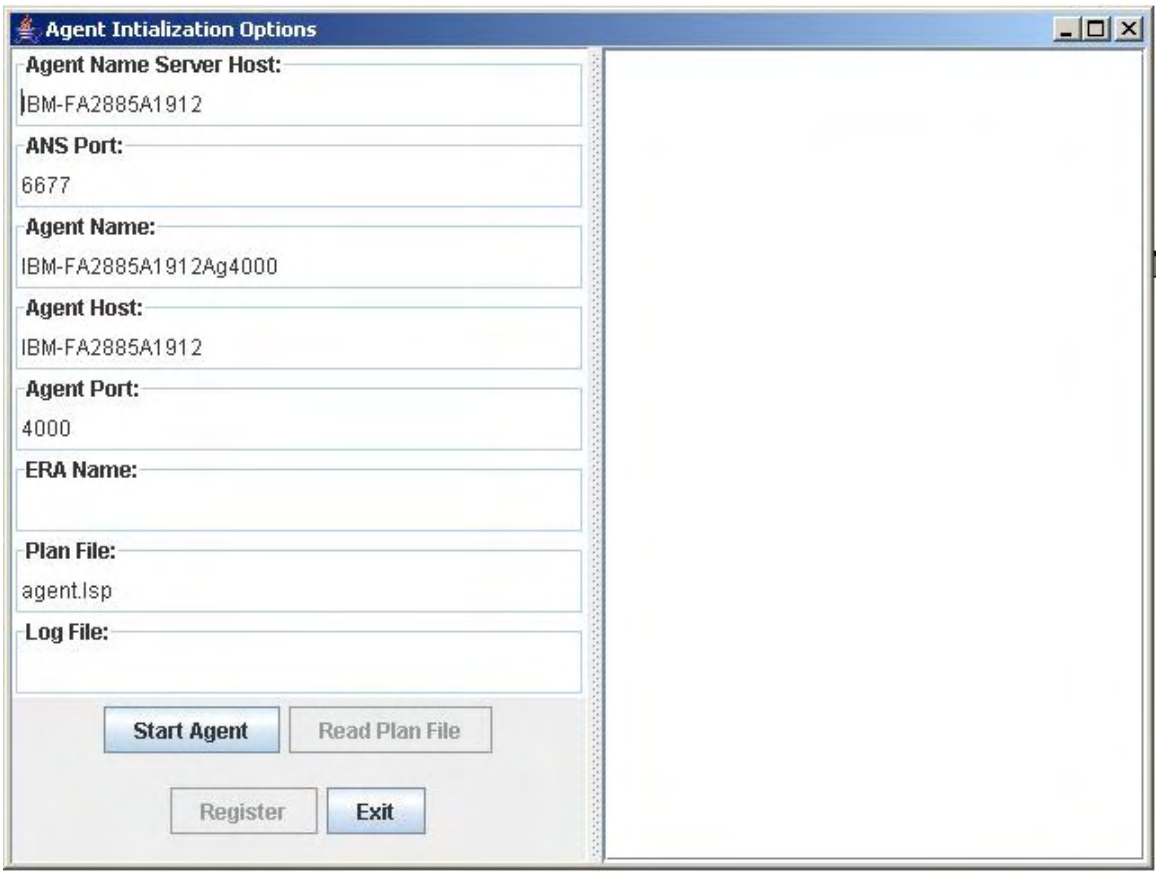

Figure 3.10: Example of an agent initialization using the GUI

Since DECAF is a multi-agent architecture kit that monitors and manages several agents, definitely the user has to face cases where an agent is looking for another agent to accomplish a certain task. That service is already provided by DECAF through the middleware agent **Matchmaker** [27]. Matchmaker is a client-driven agent that is provided by DECAF to facilitate client agents to find other agents with specified capabilities. In order to accomplish this, it stores the capabilities of the agents. When an agent provides a service for other agents, it advertises itself at the Matchmaker side by stating its capabilities in keywords as if it is creating a profile for its service. Matchmaker stores this information in its database, which is a text file named “advertisementDB.txt”.

The Matchmaker only deletes the agent from that database when it requests a un-advertisement later on. Hence, other agents can issue queries for Matchmaker in order to find the agent with the required capabilities. Afterward, Matchmaker responds by providing the name of the agents. The only agent in the current DECAF architecture that requires a unique name is the Matchmaker, so it can be recognized by other agents.

4. Semantic Web in Web Services

4.1 Ontologies

Recently, the term ontology started to spread and became more commonly used; many questions were raised about the meaning of this terminology. The term ontology has many different definitions depending on the context. In this case, an ontology [2] is a formal explicit description of concepts in a domain of discourse (**classes** (sometimes called **concepts**)), properties of each concept describing various features and attributes of the concept (**slots** (sometimes called **roles** or **properties**)) and restrictions on slots (**facets** (sometimes called **role restrictions**)). The combination of an ontology and a set of instances of its classes make up a knowledge base.

The most important component of the ontology is the set of classes. Classes in ontologies describe sets of similar individuals in a certain domain, and they are the equivalent to a T-Box in description logics as mentioned in Chapter 3. For example, a class of *THESIS* represents all theses, but when it comes to a specific existing thesis then this would be an instance from the class *THESIS*. Consider having a class *COMPUTER SCIENCE THESIS*, then the thesis that you are reading right now would be considered as the instance of that class. These classes could be divided into subclasses, which were already used, by dividing the thesis into more specific ones, by having *COMPUTER SCIENCE THESIS* which is a subclass of the class *THESIS*.

Slots give the description for the properties of the classes. So, the *COMPUTER SCIENCE THESIS* is written by a computer science graduate student. This means that the slot that describes the thesis in this example is the slot "written by" and it has the value "computer science graduate student". By having the combination of these three basic components (classes, instances and slots) a complete ontology can be defined.

In practice, in order to develop an ontology it should include the definition of classes in the ontology, the definition of slots and setting the type of their values and finally the value of these slots. Figure 4.1 shows a brief example of the ontology of thesis that will make the picture a bit clearer.

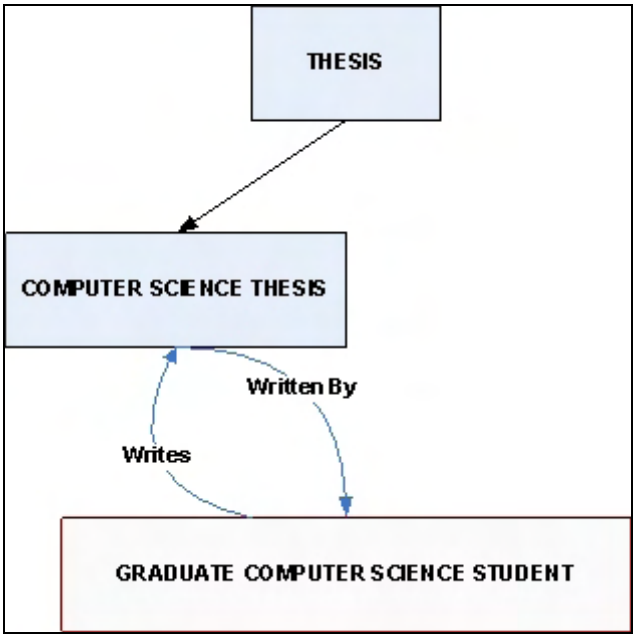

Figure 4.1: Example of an ontology taxonomy

Ontologies are used for different reasons. They can be used in order to share common understanding of the structure of information among software agents. By having this

shared infrastructure, queries between agents will be more specific and related to the domain with less sources of misunderstanding.

For this reason, and with reference to the problem described in Chapter 2, it is possible to indicate that using ontologies in such a field will be a great asset for solving this problem. In order to create these ontologies a tool called Protégé [28] was used. Protégé is one of the most popular ontology and knowledge-base editors because it provides powerful functions to design and implement ontologies. Protégé allows authors to create and import contents. It also allows authors to edit both the original and the imported contents using a GUI (see Figure 4.2) [29].

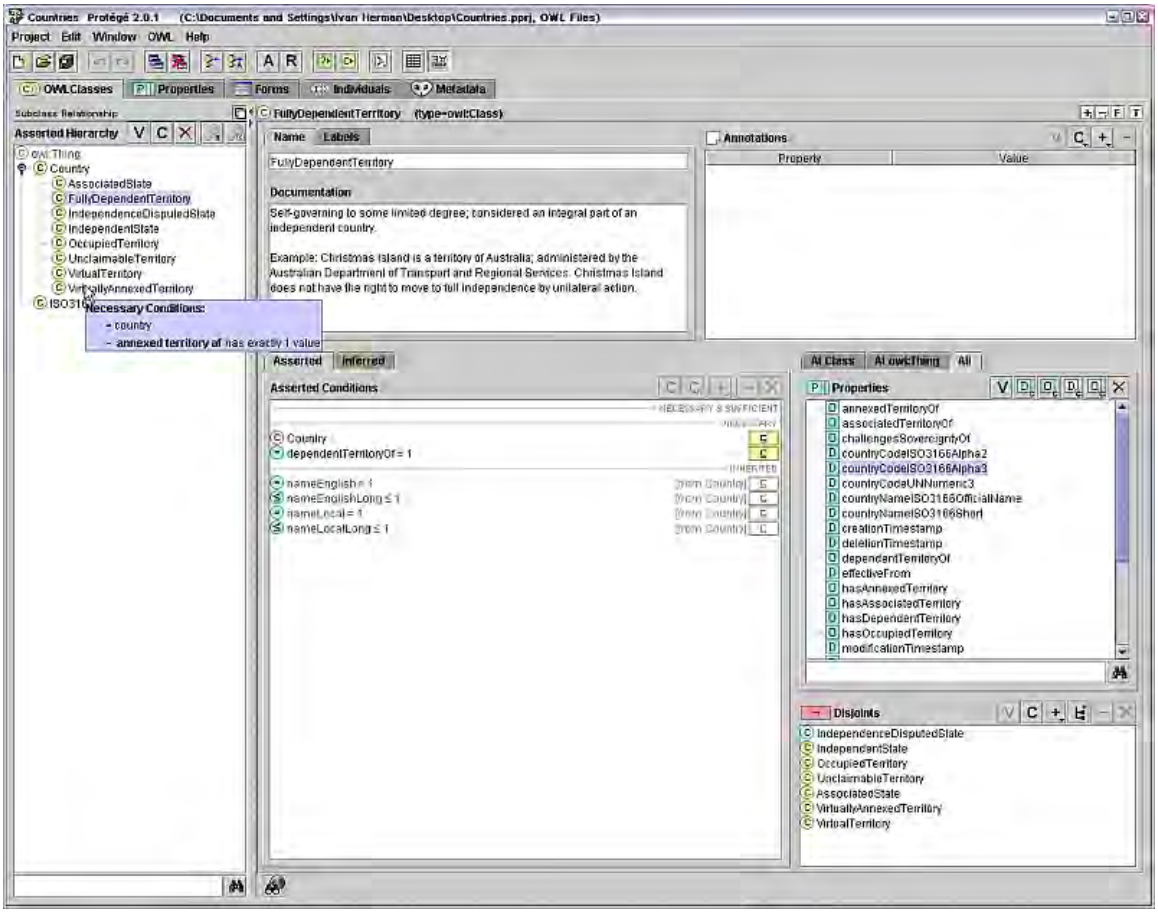

Figure 4.2: Protégé's user interface.

4.2 Matchmaking

Matchmaking, in general, is the process by which parties that are interested in the exchange of economic value are put in contact with potential counterparts.

The matchmaking process is usually carried out by toning together features that are required by a party and provided by another. In the traditional way of accomplishing that, this process is achieved either by brokers, where they continuously seek counterparts in directory services such as the yellow pages, or by looking at advertisements on media.

With the possibilities opened by e-commerce on the internet, the number of potential counterparts dramatically increased [30].

As mentioned before most of the existing matchmaking techniques, such as UDDI and CORBA/ODP are based on string comparison; so if service providers neglect to provide sufficient or appropriate terms for the matchmaking process, the search techniques will return incomplete results as seen in Figure 4.3, which will be described in detail in Chapter 5.

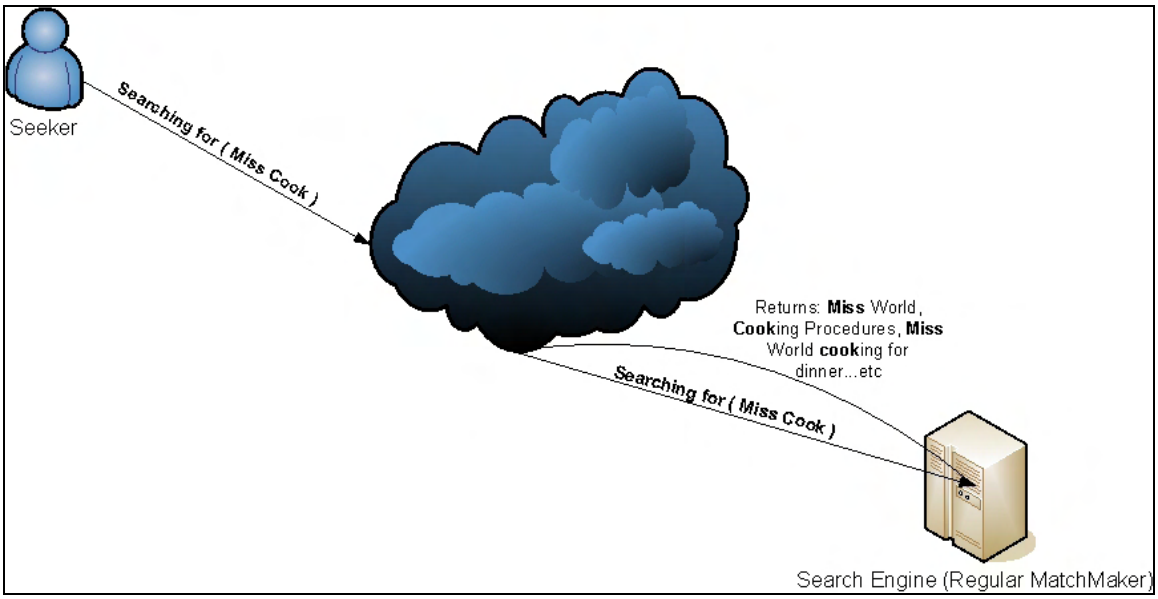

Figure 4.3: String comparison matchmaking scenario

In order to avoid these problems, the main issue that will lead to such a problem has to be located. From the previous example we can conclude that the problem is caused by the string comparison technique. The problem is that the advertised terms at the Matchmaker side are treated as a combination of letters in a certain format without any meaning for the machine. But if the concepts were loaded in a way that carry their definition or meaning with them in a certain format, then by looking for Miss Cook, the machines will understand that the user is looking for a person, who is a female, who is not married and has the name Cook. This example shows that the best idea for avoiding such dilemma is using semantic web techniques.

As defined before, the ontologies are used to define concepts in a semantic way that would make them machine understandable. To avoid the problem of irrelevant and missing results, concepts could be represented in an ontology with a concept definition

based on other relevant concepts. In order to create an ontology a user has to use the standardized Ontology Web Language (OWL) with the help of a tool such as Protégé [20].

Having these concepts represented in ontologies requires an assistant from a third party. In order to avoid having human interference, software agents could be used in order to make it as automated as possible. The overall picture could be to have an agent for every web service, and these web services are described by an ontology represented in OWL.

The described scenario is shown in detail in Figure 4.4. It shows how the seeker agent communicates with the Matchmaker who, in turn, communicates with RACER in order to query the ontologies and communicate with the provider agent, who is representing a web service through OWL-S. The communication between the agents is based on KQML due to the architecture of DECAF.

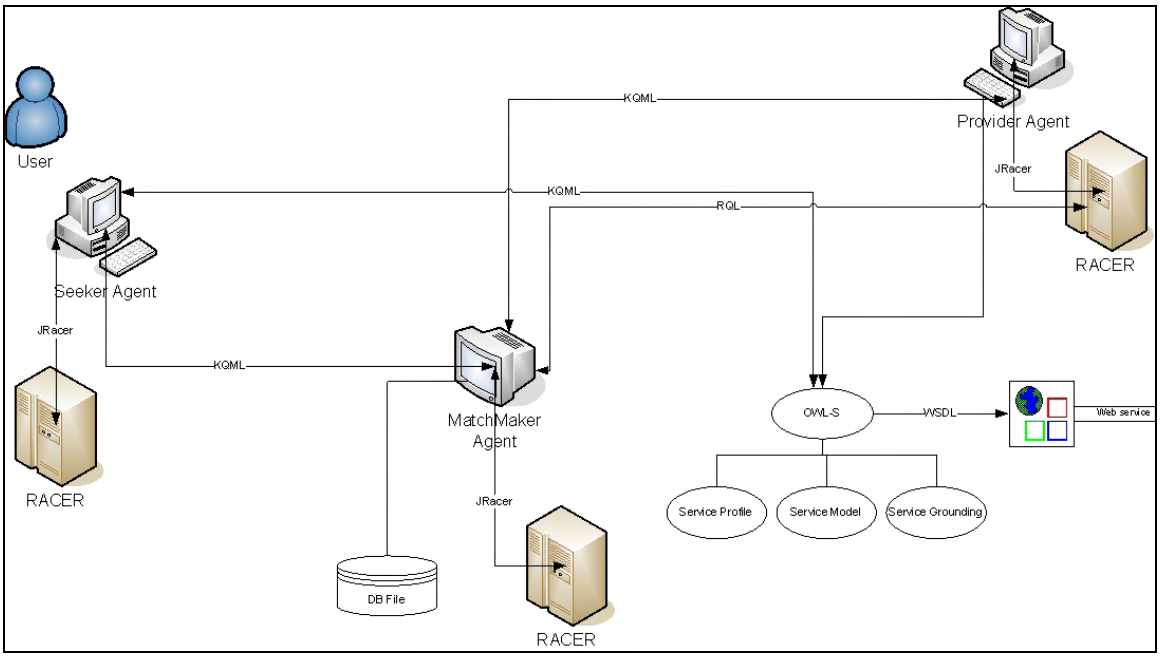

Figure 4.4: Matchmaking Scenario

As also shown in Figure 4.4, the scenario includes three major parts: the seeker, the Matchmaker and the provider agent. The seeker agent searches for the right provider agent with the right service, by sending the request to the Matchmaker in an ontology based query format. The provider agent is the agent that provides the services that the seeker agent is looking for. The provider agent communicates with the Matchmaker by advertising itself under the right category in an upper ontology (which will be explained further in Chapter 5). It supports the web service using OWL-S. Finally, the Matchmaker, which is the most important part of the scenario, provides the matchmaking service in order to allow the seeker agent find the right provider agent using ontological definitions and queries.

That was a brief description of those components, a detailed description of their functionality and services is explained in Chapter 5.

The matchmaking procedure is divided into two main steps: **filtering** and **confirming**. The filtering part is based on a concept-based query, a query that will query the T-Box of the upper ontology, and the result of this query will be a number of agents that are related to the required service by subsumption. The main reason of doing this filtering part is to narrow down the number of provider agents in order to speed up the querying part for each one of them. If in a certain scenario 100 provider agents are advertising their services at the Matchmaker side, and a seeker agent is requesting a certain service, then, without the existence of the filtering step, the Matchmaker has to create and send 100 query commands to the 100 agents. On the other hand, by having the filtering procedure, it might help in decreasing the number of related agents. After cutting down the number of provider agents the next phase is confirming. This phase is based on nRQL (Racer Query Language) which can be used to mainly query the A-Box of the provider agent. The Matchmaker searches for a specific and detailed service and gets back the instance of that service if that agent satisfies the query, otherwise it would discard it and move on to the next agent, and so on.

The reason for not only employing a concept-based query or a nRQL query alone is that in case of using the concept-based query alone, then the agent will get all the agents that are related to the required service regardless if these agents are what the user required or not, because by querying the T-Box the reasoner is dealing with the taxonomy of the ontology and using the subsumption facility, so by getting up one level in the taxonomy (e.g. querying for the concept-parents), the range of agents will increase dramatically,

where on the other hand having the nRQL it becomes very specific, so if the user missed a slight property of the required service, this might lead into missing the agent that the user really wants at the Matchmaker side. In other words, using the suggested solution will allow combining the general range search in the beginning, which acts like agent filter, then the specific search to eliminate the unnecessary agents.

5. Design and Implementation

This chapter discusses in detail the design and implementation phase of the complete system. First an introduction of Plan Files is discussed and how they are used in the system. Then the role and implementation of each objective in the system is explained in detail.

5.1 Introduction to Plan Files

Agents in DECAF are controlled by an ASCII file that is called the *Plan File*. The plan file is written in the DECAF programming language as shown in Figure 5.1, where an example of the advertisement service provided by the Matchmaker agent is provided.

These plan files can be created easily using a Graphical User Interface tool called Plan Editor. Plan editor is provided with the DECAF kit. In the plan editor, everything can be built by dragging objects from a toolbar and dropping them into the panel. Actions are treated similar to building blocks, where one simply drags an action element and connects it to another action to achieve larger and more complex tasks. The structure of these actions is a hierarchical basis, and it is based on the style of HTN (Hierarchical Task Network) [53].

```
(defaction
     :name(“loading_ontology”)
     :parent(“Matchmaker_advertise”)
     :children(“NONE”)
     :parameters(“NOPROV”)
     :provisions(“keywords” “ontology”)
     :outcomes(“fail” :behaviour_profile
                (:cost “0” :quality “0”
                 :duration “0” :density “0”)
                 “OK” :behaviour_profile
                (:cost “0” :quality “0”
                 :duration “0” :density “0”)
               )
     :deadline(“0”)
     :earliest_start_time(“0”)
     :caf(“AND”)
     :utility_function(“NONE”)
)

(provides :from(“Matchmaker_advertise.ontology”)
          :to(“loading_ontology.ontology”)
)
```

Figure 5.1: Plan file example.

Items in the plan file are divided into three categories: task, action and non-local task. Tasks are the upper level of the actions, so, if an user, for example, wants to print a paper, the task will be to print the paper in general regardless of the little details afterwards. Actions are listed under the task most of the time. If a set of actions is satisfied that means their task is done, so, considering the example of printing the paper, the actions would be to get the text, confirm the format and send the text to the printer. By satisfying these actions the task is accomplished. Finally, the non-local-task communicates with other plan files that belong to other agents. Of course this part will require extra

information leading to the right agent and the right task at that agent, so in the example of printing, the text could have been received from another agent that provides news articles, and afterwards, when the text has been received it get passed on to the next action. The printing example is shown in Figure 5.2 with a plan file defined and created using the plan editor.

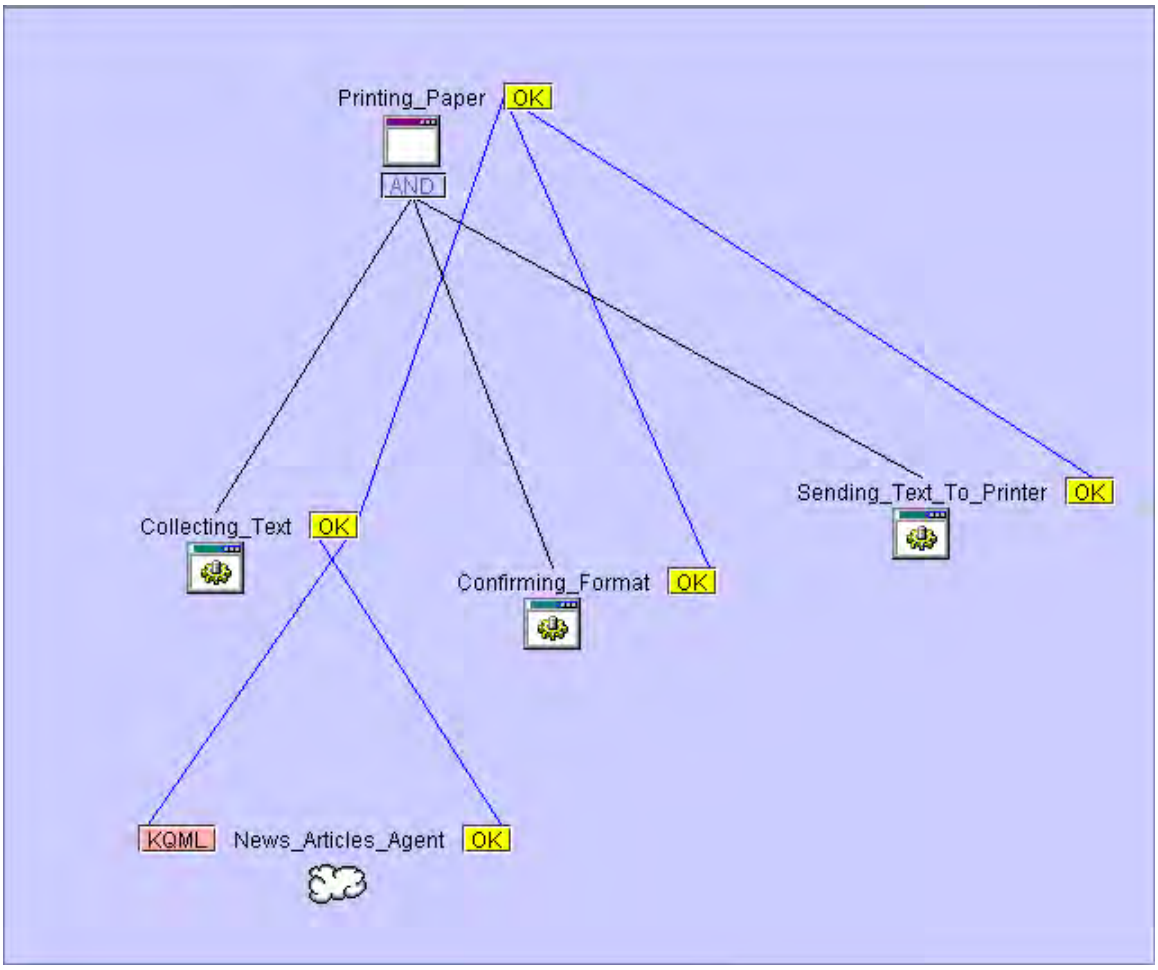

Figure 5.2: Plan file for the printing example.

After having explained the used tools and the main problem, the suggested solution, as shown in Figure 5.3, is discussed. The main scenario that was described as solution basically consists of agents that represent different parties, where a seeker informs his/her agent to seek for a certain product or service. Using the task of “reading apartments ads” as an example, the seeker’s agent will be able to find the appropriate provider agent, if available in the market, through certain techniques and procedures that are explained later in this section. In general, the presented solution can be divided into three main steps: an agent providing Web services (*Provider Agent*), an agent requesting Web services (*Seeker Agent*) and an agent which takes care of matching the seeker agent to correct provider agents (*Matchmaker*).

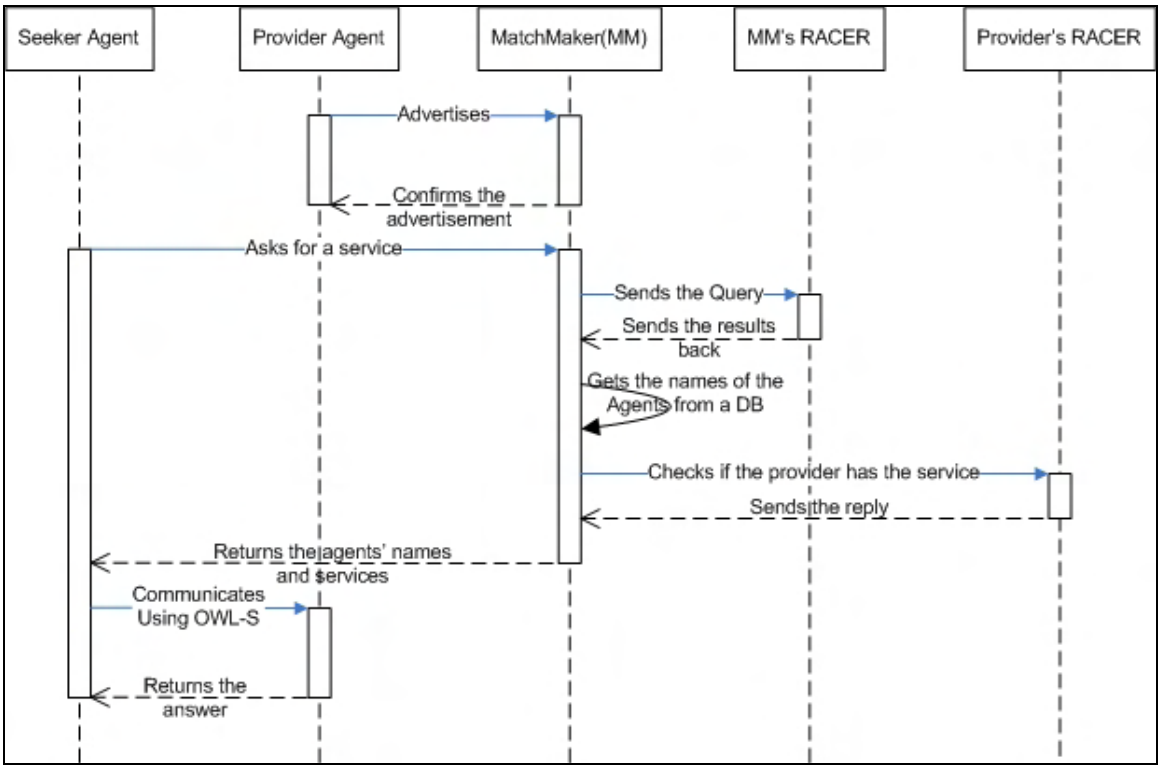

Figure 5.3: Sequence diagram illustrating the agent communication

5.2 Role of Matchmaker

As mentioned before, description logics are a family of knowledge representation formalisms. description logics are based on the notion of concepts and roles, and they are characterized by building complex concepts and roles from atomic ones [47]. This feature, that allows one to define complex concepts based on atomic ones using subsumption, is what makes description logics a good choice for matchmaking. Because of subsumption detection, when a certain concept is being looked for using description logics, one could get as result:

- Concepts with an equivalent definition.
- Concepts that are directly subsumed by the definition searched for, in case the exact concept could not be found.

Those results, in general, can not be achieved with the string comparison search techniques since the string-based definition of concepts is not based on subsumption in a hierarchical model. Therefore, in case the exact concept that the user is looking for does not exist, but a concept that is very similar to the search concept exists, it can not be found using the string comparison search technique.

In order for a seeker agent to find a certain service from a provider agent, there should be a third party which takes care of that specific issue. That third party is called the Matchmaker. The Matchmaker, as mentioned before, is a middle-ware agent that is provided with DECAF. It is considered as a middle-ware agent because it acts as a coordinator between the seeker and the provider agent. The Matchmaker is in charge of

matching requested services to proper provider agents. The current version of the Matchmaker has a problem, its search technique is based on string comparison which, as mentioned before, can cause performance problems and produce incomplete results for seeker agents. So the main part of the provided solution in this thesis is to design and implement a new Matchmaker by replacing string comparison with reasoning based on ontologies. The new Matchmaker offers several new features. The main feature of the new Matchmaker is the ability to distinguish between the different domains agents belong to. That feature was accomplished by having an upper ontology, which will be explained in detail in this Chapter. It is an ontology that includes everything related to the different agent domains and connects them in a reasonable way. This ontology is basically universal. All the Matchmaker agents have to attain one in order to ground their domains and make the matchmaking procedure easier for the Matchmaker agents. Thus it will be easier for the seeker and provider agents to find their category without any conflicts or arising problems. That upper ontology acts as make-shift glue which connects all the agents' domains. For example, Figure 5.4 shows a sketch of a part of an upper ontology that basically emphasizes on the "apartment" concept. The connection between the apartment and the other concepts can be seen in a very general way. It also shows the location of the apartment in the hierarchy of the ontology. And by having the apartment in such a general ontology, it would be possible to relate it to other existing concepts in the ontology even if they are not directly related. This means under this upper ontology the related and non-related concepts can be distinguished. As a result, when a seeker searches for a service, the upper ontology is used as grounding for communication and for getting only the related agents. What makes the general ontology a better technique

than a regular database, which includes all the concepts of all the domains and relates them to the agents, is its ability to reason with ontologies. Another feature that was improved in the new Matchmaker is the use of a database to store information about agents (e.g. their names and offered services), which will be discussed in Chapter 5. In comparison, the past models used to store the information in a text file instead of a database. The database is used also to keep track of requests while performing the search for matching agents. For each request there is a unique ID number that is passed with the on-going event that is made to satisfy the seeker agent's request. At the end when the events are terminated and the results are returned to the Matchmaker, the Matchmaker will point out the seeker agent, who sent this request.

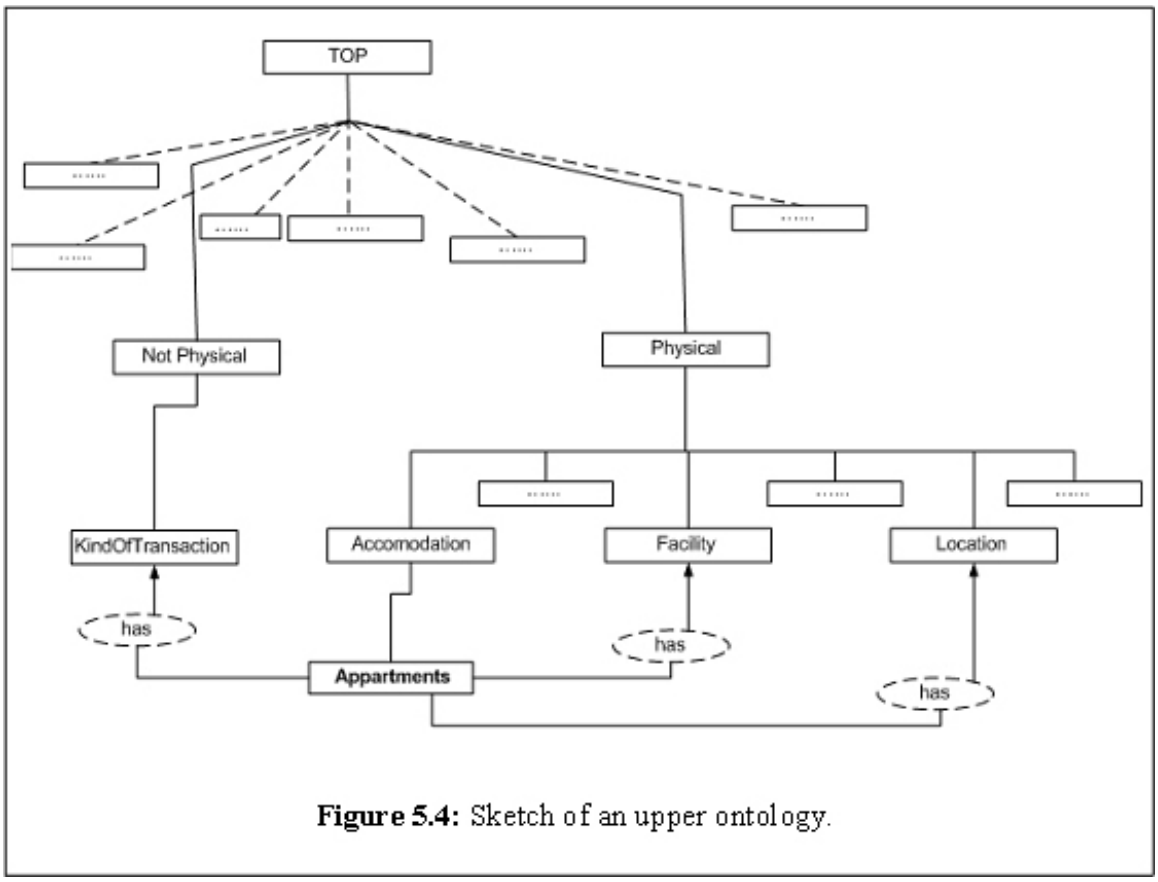

Figure 5.4: Sketch of an upper ontology.

Three basic services are provided at the Matchmaker in order to communicate with other agents. These services are *advertisement*, *asking* and *deeper*, as seen in Figure 5.5, and they are explained as follows.

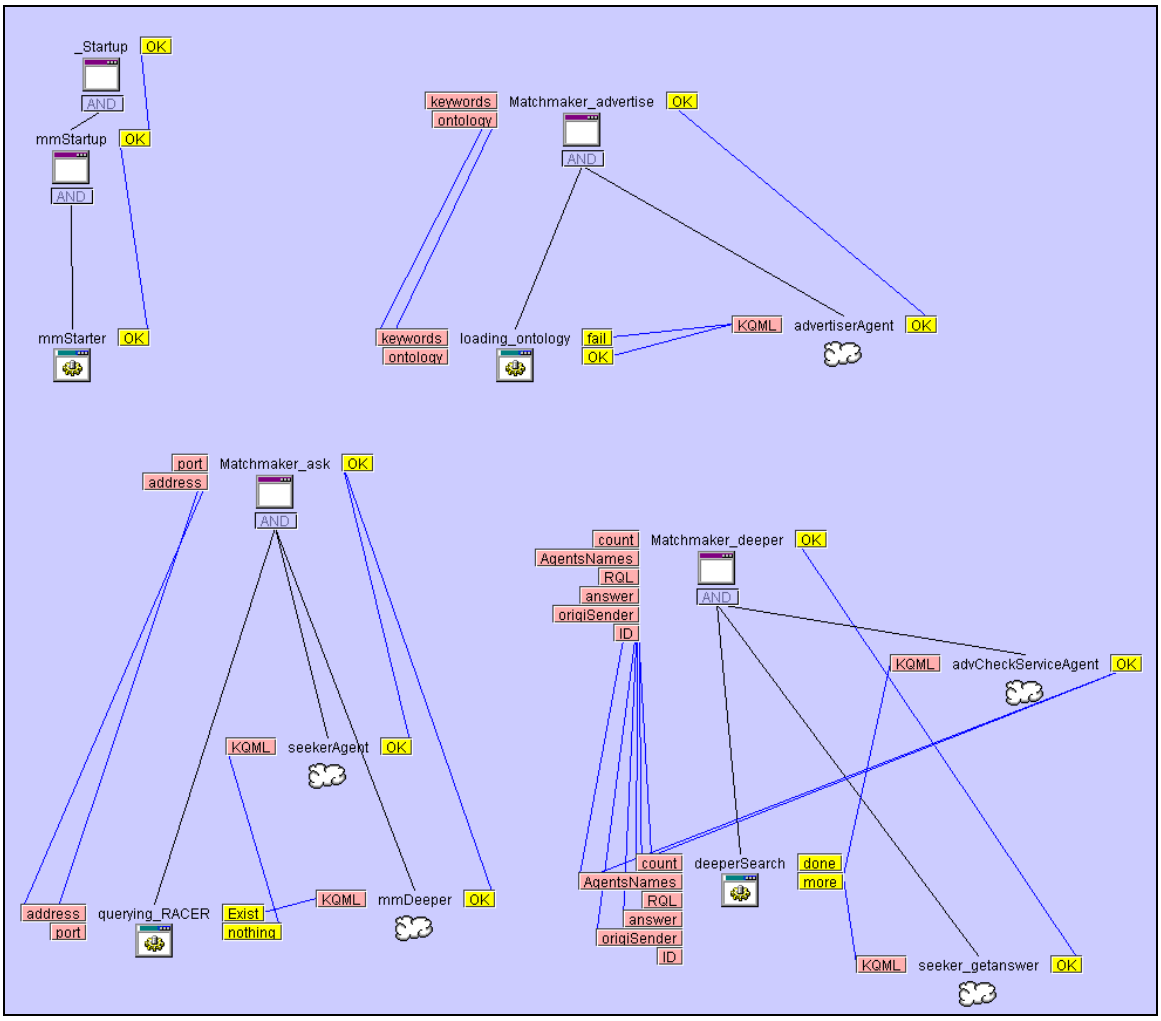

Figure 5.5: Plan file of the Matchmaker

5.2.1 ADVERTISEMENT

The provider agent has to register at the Matchmaker side and advertises itself in order to allow other agents to use its services, and to be known in the e-market. For that reason the service Advertisement was provided, and it expects two parameters. The first one is a set of **concepts**, which includes all the related concepts for that particular service. These

provided concepts must occur as concepts in the upper ontology (e.g., if an agent is providing apartments, and the upper ontology includes housing as a concept name, then the agent will provide housing as a concept for that service in order to register under it). The concepts provided by the provider agent will be the key to the appropriate place in the taxonomy. The second parameter is the **category** which will be used exclusively by DECAF, usually the category is the same as the agent's name.

5.2.2 ASKING

The second service is *asking*, which fundamentally addresses the search part in the Matchmaker. Whenever a seeker agent starts searching for a certain service, it calls the asking service at the Matchmaker. The asking service expects two parameters. The first parameter is **ConceptBasedQuery** which is a general query that will be used as a filter on the upper ontology at the Matchmaker's side to get all the related provider agents. This query does not actually deal with the A-Box of the ontology; it basically queries only the T-Box. The Matchmaker executes the query based on subsumption, so if the exact request was not available in the upper ontology it will return the parent concept of the existing term in the T-Box. In this way the system can usually guarantee not to reach to the top of the ontology. By these means the Matchmaker will have a general picture of the request since the upper ontology is being dealt with at the Matchmaker side only. The second parameter is named **nRQL** (new Racer Query Language) which contains a more specific query that will be used at the provider agent's side to check whether the filtered agents truly provide the requested service. What makes the nRQL special is the ability of

representing complex OWL queries, which is not possible in the case of regular queries or keyword matching techniques. The nRQL deals directly with the instances in the A-Box, therefore, the results of the query return the exactly wanted service at the agent's side which satisfies all the requirements, if available. On the other hand, using the ConceptBasedQuery might return the agents that do not match the exact requirements that the seeker agent was primarily seeking for. However, at least it will return to the originator all the agents that could serve those services. In other words the ConceptBasedQuery can be considered as a filter that retains only the related agents regardless of the details. Thus the solution will be more efficient since the number of agents that have to be checked will decrease to the related agents only. Then, the nRQL can be considered as the main query that retrieves only the exactly matching agents with all the details since it is more specific than the ConceptBasedQuery and furthermore because it deals with the A-Box at the agent's side.

Precision and **recall** are the basic measures used in evaluating search strategies. These terms will be used for evaluating the two query techniques used in matchmaking. Precision is the ratio of the number of relevant records retrieved to the total number of irrelevant and relevant records retrieved. It is usually expressed as a percentage. Recall is the ratio of the number of relevant records retrieved to the total number of relevant records in the database. It is usually expressed as a percentage.

By using the ConceptBasedQuery, the solution is guaranteeing a precision of 100% at all cases because the ConceptBasedQuery is querying the T-Box of the ontology and no irrelevant concepts would be retrieved from the concepts in the T-Box. Regarding the

nRQL query it will improve the search done by the ConceptBasedQuery. The estimated recall percentage based on the nRQL query was not calculated in this research because of the required resources of populating a big number of agents with ontologies.

5.2.3 DEEPER

As mentioned above, the Asking procedure has two parameters, one for the general filter, and another one for the specific search. In order to accomplish a more specific search, the Matchmaker has to take the list of agents from the first query and communicate with each one of them, and confirm the existence of that service based on the second query (i.e. the nRQL). This procedure makes use of the service *deeper*. This service in particular will never be called by any agent but the Matchmaker itself. The parameters of this service are:

- **count** (which contains the number of the agents to be checked).
- **AgentsNames** (the list of filtered agents from the first query).
- **nRQL** (which will be used in order to query the specific services in the search).
- **Answer** (is what will be returned from the provider agent that the Matchmaker was checking in the first place).
- **origiSender** (the name of the agent that made the search request).
- **ID** (which will be the unique ID that will be explained later in this section).

The *deeper* service takes the list of filtered agents from the *asking* service and sends the nRQL query to each of them and obtains the answer from them. Since the deeper search procedure is being requested from the Matchmaker only, and since more than one agent

can initiate the service at the same time the *deeper* request that is initiated from the Matchmaker has to be kept track of because the *deeper* service deals with more than one agent at the same time. Therefore, it has to be known which request belongs to which agent. In order to solve this problem, a database is used that keeps track of the requests by giving each of them a unique number. It also provides a service that activates each time a service returns with an answer from a filtered provider agent. The answer will be saved in the database in the same field of that unique number and the number will be passed on to the next agent. After checking on all the filtered agents, it returns the matched agents' names with their offered services to the seeker agent.

5.3 Grounding Technique

One of the major problems encountered in implementing this system was the performance of the matchmaking. If the search technique were left to be peer-to-peer, meaning that the Matchmaker acts as a redirector only, then, whenever a seeker agents starts searching, it asks the Matchmaker for all the advertising agents and communicates with each one of them and checks whether it offers the provided service. But of course this is time consuming and not feasible in general. Also, if the matchmaking technique were left the way it was provided by DECAF, then it will be based on keywords, so that will lead back to the problem of string comparison which is the main issue that has to be avoided in the first place, and the semantic web concept will be lost in the first place. Therefore, it was decided to create an upper ontology that will handle the grounding

between the agent domains.

The upper ontology (Figure 5.4) is a representation for the general (upper) concepts that can be used in a certain domain, and it can contain many general concepts if it has to be generalized, so it can be also considered as an indexing system and a categorizer. Since the upper ontology contains concept names only without instances, it is represented by a T-Box only.

As mentioned before, the upper ontology acts like a categorizer. It categorizes the provider agent into the corresponding subcategory of the general concepts, e.g., in the apartments scenario if some provider agent has apartments or accommodation that it wants to sublease, rent, sell, …etc, then the provider agent would be listed by subsumption under the concept apartments. Another feature of the upper ontology is that it illustrates the relationships between different concepts. If two concepts are subsumed by common concepts, it will be logically shown in the upper ontology through subsumption (e.g. *Accommodation* and *Facility*, they are both *Physical* therefore they are both subsumed by *Physical*), and if they are not related then it will also show the common ancestors (e.g. *KindOfTransaction* and *Location* yet they do not have a direct subsumption relationship between each other). In this way the categorizing feature will not mislead the provider or the seeker agent because the Matchmaker will be able to distinguish subsumption between related and non-related concepts.

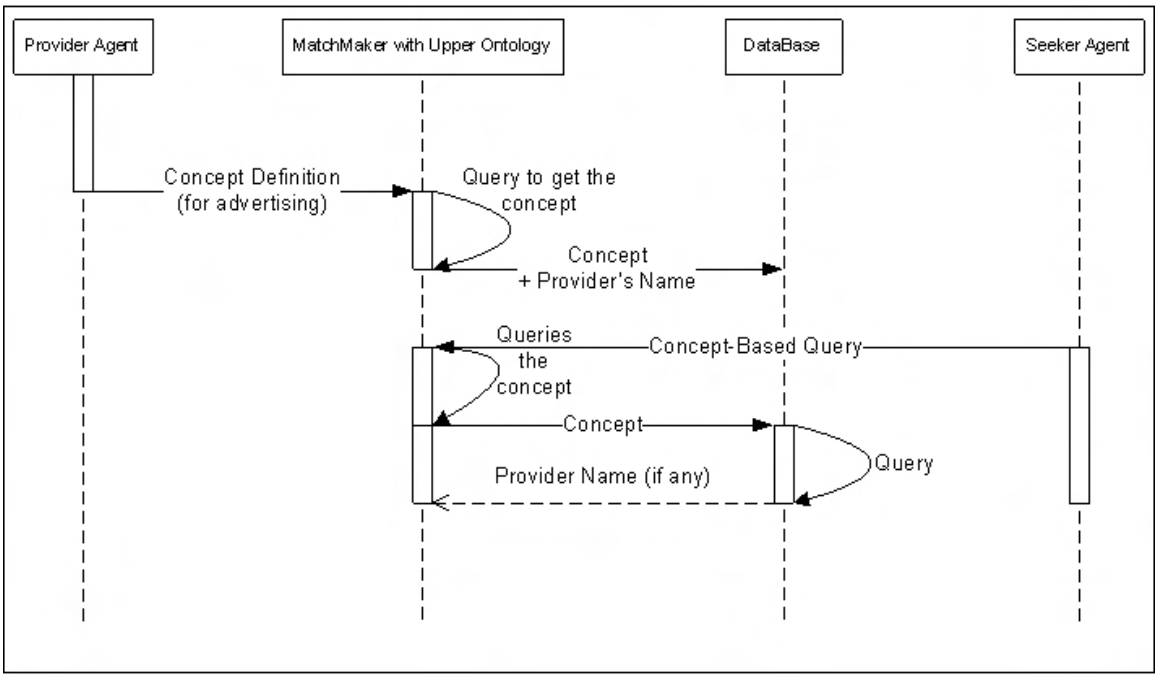

Figure 5.6: Sequence diagram for the upper ontology

After describing the upper ontology and its functionality, this section describes how the upper ontology functions according to the sequence diagram in Figure 5.6.

The upper ontology is used in two different ways. The first way is when a provider agent advertises its service, and the second way is when the seeker agent searches for a service. When the provider agent advertises its service it sends a concept-based query, which is usually the concept definition of the service, to the Matchmaker in order to categorize itself. So the Matchmaker receives the query, sends it to RACER, and returns the most-specific subsumer concept for that definition in the upper ontology. Afterwards the Matchmaker sends the returned concept name along with the provider's name to the database. Then the database binds both the concept with the provider name for future use.

If a seeker agent wants to find an agent with a service that satisfies its requirements then it sends the Matchmaker a concept-based query for the service that it is looking for. Most of the concept-based queries are in the format of retrieving either the (concept-ancestors) or (concept-descendants) for the definition of the required concept. In this format the user will be able to narrow or widen the number of filtered agents depending on the requirements. The Matchmaker, in turn, queries RACER in order to get the most related concepts, which lead to the right category, from the upper ontology. After it gets these concepts, it queries the database looking for the provider agents that are advertising their services under those categories. Next, the database returns the provider agents, if any, back to the Matchmaker in order to proceed with the rest of the matchmaking technique.

5.4 Role of Provider Agent

As mentioned before, Web services are software systems identified by URIs. And most of the web services advertise themselves like any other website, using a regular search engine, so in this way, finding the web service will not be as easy as required. That is a reason why web services are usually represented by agents.

A provider agent represents a web service and makes it available for any seeker agent that seeks for a web service matching the specified requirements. It can be considered as the supplier in the real market, but in this situation the agent is the supplier in the e-market and the web services are the resources for the market, where it can be used by any other agent as long as it satisfies the seeker’s requirement. This agent was implemented

with two main features, *deeperSearch* and *activateService*, one is used in order to make it searchable and the other one to make it directly executable (as seen in Figure 5.7). The deeperSearch service is the service that communicates with the Matchmaker only, and, to be more specific, with the deeper service at the Matchmaker side. The main function of the deeperSearch is to confirm whether the required service actually offered by this agent, the filtering and the confirming, and this part is the confirming. If the Matchmaker communicates with a provider agent and poses a nRQL query to it, that means the agent already matches some of the seeker's requirements, but by applying the deeperSearch it verifies whether the service is exactly what the seeker is looking for.

The provider agent takes the nRQL query, which queries its ABox, and poses it to RACER by referring to the corresponding ontology. Afterwards it replies back with the service name, if available, or with none if no matching services could be found on the basis of this ontology. There are four parameters for this service:

- The first parameter is **count,** which as previously explained will be used when it sends the reply back to the Matchmaker, in order to keep track of how many other agents are left in the deeper search procedure before termination.
- The second parameter is **AgentsNames** which contains the names of the other provider agents that might satisfy the requirements and offer the right service as well.
- The third parameter is **ID** which is the identifier for this specific search.
- The fourth parameter is **origiSender** which contains the name of the seeker agent.

The other service, activateService, is used by the seeker agent after receiving the results

from the Matchmaker. This service makes use of OWL-S by representing the existing web service in an ontology format, so that a seeker agent can retrieve the parameter profile of the service and executes the web service call. Using OWL-S makes the representation of web services more flexible and dynamic because the interface of an agent might change at any given time. The service description is compatible with WSDL and agents can adapt to changed services at runtime without the need to code any of the involved agents or to even temporarily shutdown any service in order to change any kind of parameters. This procedure is done automatically as soon as the WSDL is changed from the provider's side. The OWL-S description specifies a communication protocol between the seeker and provider agent for executing the web services successfully. In order to make the changes refreshable at run time an API of OWL-S was used which translates the WSDL file into OWL-S, so whenever a change occurs in the web service that API is applied and as a result the newest web service will be active online based on OWL-S.

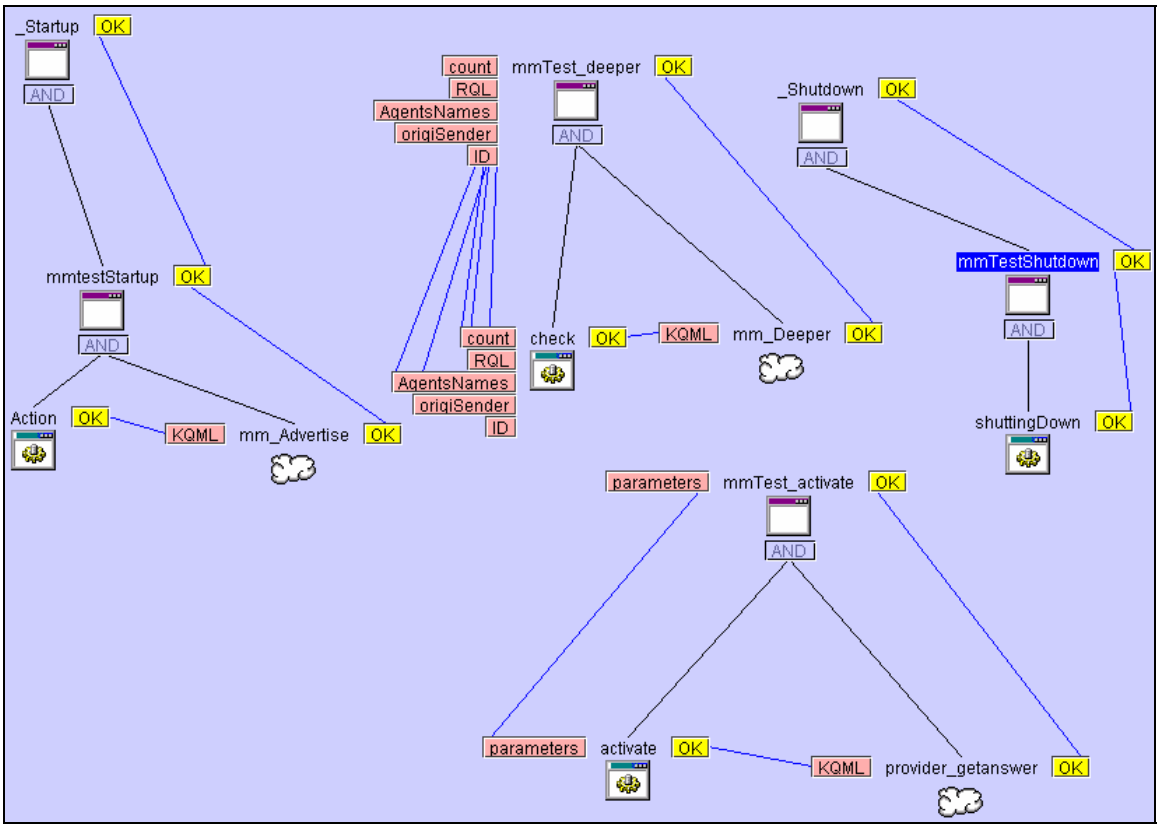

Figure 5.7: Plan file of provider agent.

5.5 Role of Seeker Agent

Presently, when a user wants to use a web service, he/she searches for it using a search engine, which is most probably based on string comparison. Afterward he/she checks the input parameters manually, writes an application that would use a certain protocol to communicate with the web service based on the WSDL and then executes the web service. This means that whenever one wants to use a web service he/she has to dedicate a professional programmer that has to create the interface if an online interface was not provided by the webservice provider. In order to avoid this problem a third party, which

replaces the programmer, would be the software agent and it could be called the seeker agent.

The seeker agent in this scenario acts as the consumer in the real market, where it has two main functionalities according to its plan file in Figure 5.8. The first function is the need to communicate with the Matchmaker, in order to retrieve the names of the matching provider agents that have the required service. In this service it passes the required parameters at the Matchmaker side, which are the nRQL, Racer Query Language, and the ConceptBasedQuery, and it expects the answer back through another service called getAnswer. GetAnswer takes one parameter only which is the parameter named answer. The other feature is named **activateService**, which communicates directly with the web service of the matching agent in order to activate that specific service. The special characteristic about this is its use of OWL-S. By using OWL-S, it is guaranteed that the input parameters of the web service are captured instantly as soon as it needs to activate the service. It asks the user to fill in the missing information, noting that if there is already provided information, the seeker agent can use it without requesting any further information from the user. In the case of missing parameters it will ask the user to fill them in, and after that the service will be executed and it will return a reply confirming that the request went through or an error if something went wrong. Therefore, this is a regular activity of an agent using web services, but with the ability of instant and immediate changes on the web service, and a better matchmaking procedure.

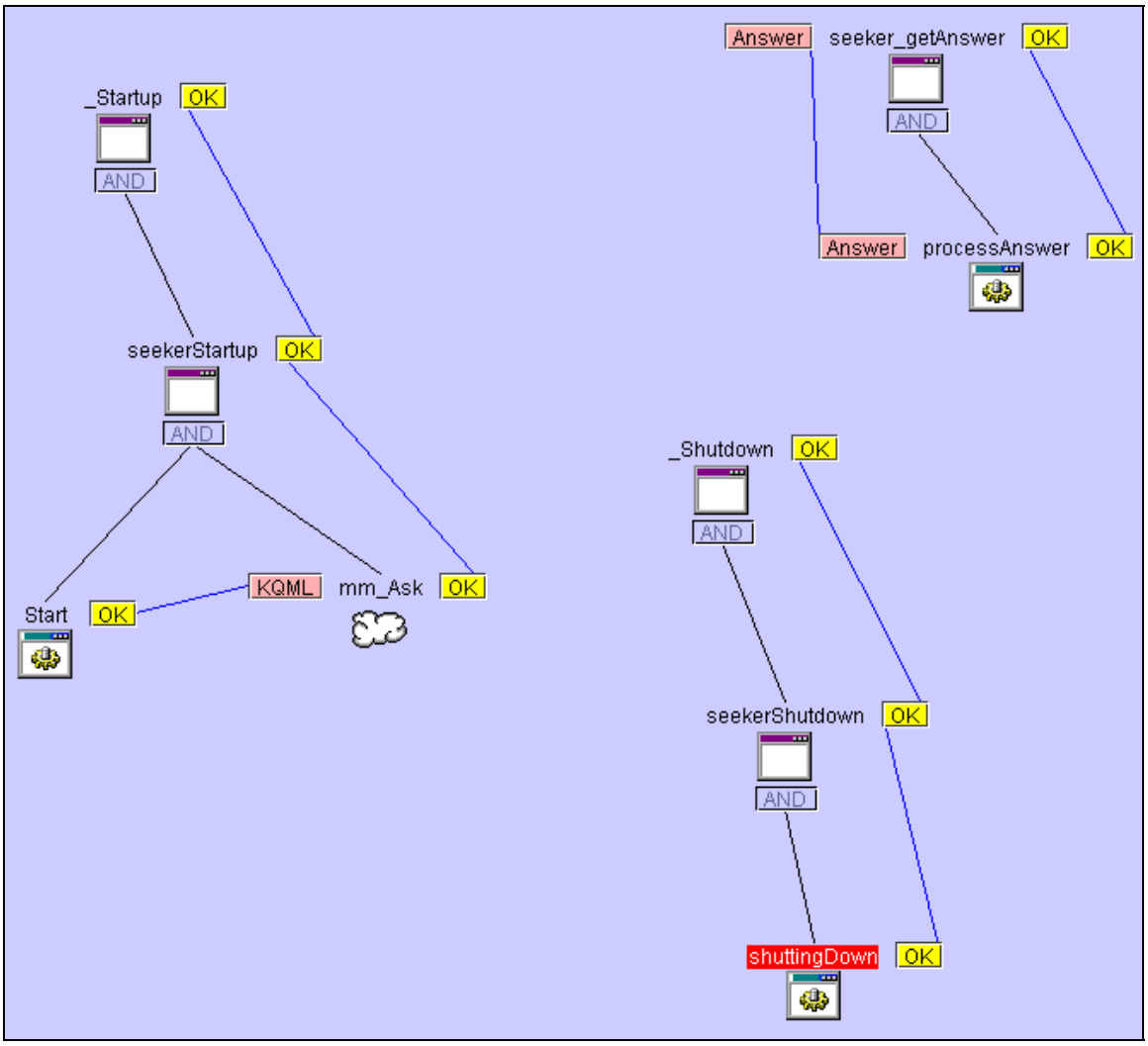

Figure 5.8: Seeker's plan file

The seeker agent should be knowledgeable of the upper ontology at the matchmaker agent in order to be able to register under the right concept. Also the seeker agent should have a general idea of the concepts at the provider agent in order to get a more precise results when the A-Box query occurs. Notice also that the ontologies at the matchmaker, provider and seeker agents should be designed and created by expert ontology designers not by amateurs.

5.6 Scenario

To make the picture clearer, a real world example will be explained, based on the scenario in Chapter 2, which has already been implemented in Java with the mentioned tools and framework. The scenario is about an agent providing a web service for renting out apartments (apartment_Agent). It is providing apartments in the Downtown area of Montreal city, and there exist two types of apartments: a one and half and a two and half. Another agent is providing a web service that sells bananas (banana_Agent).

Finally there exists a seeker agent that is searching for a one and half apartment for rental in downtown Montreal. Initially the two provider agents register their web services with the Matchmaker agent. They provide a concept based definition of the concepts that are the guide for the Matchmaker in order to fit this service in a certain category of the upper ontology with the same concepts. Along with the concept definition the provider agents provide an ontology, which is a term that is used for DECAF, in order for agents to be able to communicate with each other. After the registration and advertising procedure, the seeker agent will start communicating with the Matchmaker by sending two parameters: nRQL and ConceptBased (see Figure 5.5 for the plan file of the Matchmaker agent). The Matchmaker will send the ConceptBasedQuery, which requires apartments in general, to RACER which contains the upper ontology. RACER will query the T-Box returning all the registered agents that are related (by subsumption) to apartments in general, which is apartment_Agent in the scenario.

After filtering the registered agents and getting agents which provide the most compatible web services to the requested one, the Matchmaker will send the nRQL query to the apartment_Agent which will query the A-Box of the apartment_Agent returning the name

of the web service that exactly supports what the seeker agent is asking for. By having the name of the web service, the Matchmaker returns the name of the service to the seeker agent. Afterward, the seeker agent communicates directly with the provider agent through the OWL-S interface for the web service, by getting the required parameters for activating the service, filling up these parameters from the user, then finally executing the web service.

5.7 Querying Technique Using nRQL and Concept Based Queries

As mentioned before, the main problem was that seekers might miss matching providers. The main reason of that problem is the existing search techniques were based on string comparison. That is why the main contribution is to improve those search methods along with using agents in representing web services. Therefore two main querying techniques that fit in the semantic web model were used. Those methods are nRQL and ConceptBasedQuery. These two methods were briefly discussed in Chapter 4. In this section the reason for using nRQL and ConceptBasedQuery will be explained, how they work and what is the format of their syntax.

It was already mentioned that the ConceptBasedQuery queries the concepts in the T-Box of the ontology and the nRQL queries the instances of the ABox. It was also stated that the ConceptBasedQuery is used as a filter for provider agents trying to retrieve as many agents related to the query as possible, where the nRQL gets the specific agent that is being searched for from the filtered provider agents.

ConceptBasedQuery is best used at the Matchmaker side, the center of the search. The main reason of choosing the ConceptBasedQuery as the first filter is for the flexibility that can be achieved from using the ConceptBasedQuery. It provides the ability of alternating between the levels of the taxonomy; ConceptBasedQuery is based on subsumption, so it does not actually search for a specific instance. On the other hand it tries to fit a posed query to the closest concept that exists in the taxonomy. Therefore, if a user wants to get more specific or more general results, this could be achieved by slightly changing the query in a way that retrieves a concept in a lower or higher level of the taxonomy is retrieved. For example, if the target was a one and a half apartment located in downtown Montréal, then, the ConceptBasedQuery could have been very general by retrieving a concept name in a high level of the taxonomy like “residency” or it could have been more specific, if some properties were added leading to a concept name in a lower level like “apartments”. This will depend on the need of the users.

On the other hand, it was decided to utilize the Racer Query Language at the provider’s side. The service of deeperSearch as seen in Figure 5.7 will be the confirmer of whether the required exact service exists. Since the nRQL is based on ABoxes the user will either get the right answer, which is the name of the required service if it exists, or NULL, meaning there is no such service in this agent, although the service is most probably related to the query that the provider agent is looking for. So, in the scenario where the seeker agent looks for the apartment, the Matchmaker will find out by seeking in the T-Box with the ConceptBasedQuery that the apartment agent is in the required category. When the Matchmaker sends the nRQL query to the apartment agent, the apartment agent sends the nRQL query to Racer, checks the existence of the service, and returns the name

of the web service to the Matchmaker. The nRQL query is very precise in results, and it can guarantee to the seeker agent that the apartment agent is the one it is looking for.

After explaining how and why the ConceptBasedQuery and the nRQL query are used, the following part illustrates a couple of examples that show the syntax of both queries and what they exactly mean.

Since the ConceptBasedQuery's syntax is flexible, the user could choose whatever is convenient. For example, in the query shown in Figure 5.9 the command of *concept-ancestors* was used. *concept-ancestors* returns all atomic concepts of a T-Box, which subsume the specified concept [4].

```
(concept-ancestors (SOME |has_Location| |Montreal_Downtown|))
```

Figure 5.9: ConceptBasedQuery example.

The query in Figure 5.9 is in ConceptBasedQuery format, the equivalence of it in the DL is shown in Figure 5.10.

$$\exists has\_Location.Montreal\_DownTown$$

Figure 5.10: Description logic format of the query

nRQL is a standardized semantic web query language that is provided by RACER. nRQL Consists of a query head and body, for example, in Figure 5.11 the query head is (?x) and

the query body is (**AND** (?x |DownTownOfMontreal| |has_Location|) (?x |AirConditioner| |has_Facility|))). It returns all the, mother-child, pairs from the A-Box which is queried. Complex and simple nRQL queries can be generated. Simple queries are basic conjunctive queries (e.g. (and (?x Woman) (?x ?y has-child))) and complex queries are a combination of simple conjunctive queries connected using the boolean constructors *and, neg* and *union* [5]. As seen in Figure 5.11, the query is retrieving any instance that has a property called “has_location” which is related to an instance called “DownTownOfMontreal” and has a second property called “has_Facility” which is also related to an instance called “AirConditioner”. In other words, retrieve an instance that is located in Montreal, specifically downtown, and has air conditioning as a facility. What distinguishes nRQL from the ConceptBasedQuery is that if there exist no apartment with both of those options then nothing will be returned by nRQL. On the other hand, by using an equivalent query in ConceptBasedQuery then the result might contain an apartment that has one of the two options or both. That is why the nRQL is more specific than the ConceptBasedQuery.

```
(retrieve (?x) (AND (?x |DownTownOfMontreal|  |has_Location|)
(?x |AirConditioner| |has_Facility|)))
```

Figure 5.11: nRQL example.

5.8 Use of Java

Java is the main language that is used in the implementation of the suggested solution. DECAF is, as mentioned before, based on Java. All the components such as ANS, Matchmaker, agents, even the interface is based on Java. The main API that makes it possible to communicate easily and directly with Racer is called JRacer and it is also based on Java, JRacer will be described in this chapter. Finally most of the web services that were used in the implementation are Java based.

5.9 Using the API of OWL-S for execution

As mentioned before, OWL-S is a service description language for web services based on OWL. Represented web services are specified semantically by defining the web service, the type of it, the input parameters and their types. Therefore in order to deal with such web services OWL-S has to be parsed, the right concepts have to be categorized and the method of using the web service has to be determined. These tasks can be applied using OWL-S API.

The OWL-S API is, from its name, an Application Programming Interface used to interact with OWL-S. OWL-S API was implemented by mindswap [32].

OWL-S API provides a Java API for programmatic access to read, execute and write OWL-S service descriptions. Many versions of the OWL-S descriptions are supported by this API, such as OWL-S 0.9 and OWL-S 1.0. One of the most important services

provided by this API is the ability of executing the web service. That is accomplished through the ExecutionEngine which is included in the API. The ExecutionEngine invokes the Atomic Processes which have WSDL groundings.

Mindswap takes into consideration the data structure of the existing OWL-S ontology. That is why Mindswap implemented the APIs in a way that matches the definition of OWL-S's data structures.

Mindswap also takes into consideration the name of the packages, interface and methods in the Java implementation to match the names of ontologies, classes and properties of OWL-S ontology with slight differences as shown in Figure 5.13.

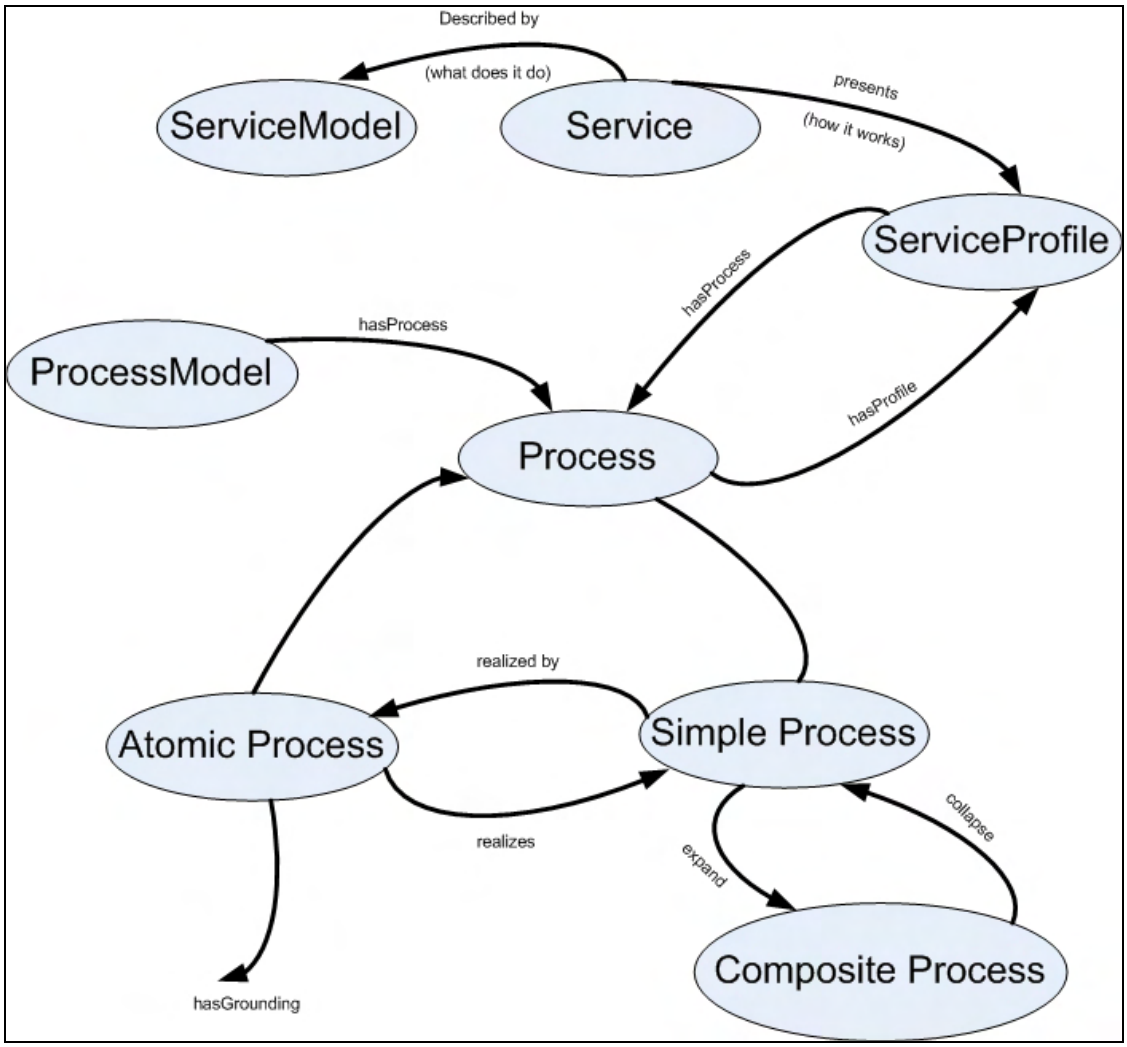

Figure 5.12: OWL-S ontology

A description of the usage of the OWL-S API is explained in Appendix A where an example of a web service that is already translated into OWL-S is presented. Then, the code is explained step by step showing what are the important factors, how are they initialized and how to use them at the right time.

5.10 JRacer

Racer has API's for languages such as C++, Common Lisp and Java. Since the main programming language was Java it was decided to choose the Java API which is called JRacer. JRacer is a TCP-based API for RACER. It is implemented in Java.

The functionality of JRacer is straightforward. It simply opens a TCP port to the specified communication port of Racer and starts streaming the commands that are created. What makes JRacer special is that since it is based on Java all the commands are object oriented. Therefore it categorizes the regular Racer commands into different objects and uses object oriented features such as inheritance.

```
RacerClient Rclient = new RacerClient("127.0.0.1",8088);
try {
  Rclient.openConnection();
  try {
      Rclient.owlReadFile(filename);
  }
  catch(RacerException e) {
    this.message(e.getMessage());
  }

  Rclient.closeConnection();

}catch(Exception e) {
  this.message(e.getMessage());

}
```

Figure 5.13: Example of using JRacer

In Figure 5.13 an example on how to open a connection with Racer and load an OWL file is shown. First of all a variable from type RacerClient is created which is a class that

provides a full Racer client. It provides Java methods that help accessing Racer primitives and perform simple parsing of the results from Racer. The constructor of RacerClient takes two variables, the IP address of the Racer engine and the port where Racer is expecting the communication. After opening this port and establishing the communication, the OWL file is loaded into Racer using the method owlReadFile which is part of RacerClient. owlReadFile is equivalent to the Racer command “owl-read-document”. It loads and represents a file in OWL format as a T-Box and an ABox with the appropriate declarations.

The rest of methods are used in the same way that owlReadFile method was used. For example, in order to get the concept definition of a certain concept, the method *getConceptDefinition* is used which is equivalent to the Racer command get-concept-definition. In case the user can not find the requested command, then the method *send* which is part of the class RacerSocketClient can be used. The user passes the command line, which he/she wants to send to the Racer engine as a parameter to this method and get the answer as a string without parsing. Afterwards, the user can parse the answer as shown in Figure 5.14.

```
try {
     Rclient.openConnection();
     try {
       Answer = Rclient.send(nRQL_Query);
          }
          catch(RacerException e) {
             this.message(e.getMessage());
               }
      Rclient.closeConnection();
    }catch(Exception e) {
     this.message(e.getMessage());
    }
```

Figure 5.14: Example of the send method in JRacer

6. Fungal Web Application Scenario

The Bioinformatics research field is having a lot of attention and effort spent toward it. That is why the suggested solution was applied to a Bioinformatics field. A proof-of-concept was realized by building the provider and seeker agents based on the required tasks that the biologists would need.

6.1 Scenario

The main user who will be using this system for this scenario is the biologist. The scenario will contain an agent that provides a service that represents two different web services. This provider agent would provide the protein sequence in a SRS format [48] for a protein from a specified database. Afterwards it would use that retrieved protein sequence by supplying it to another webservice in order to get the rest of information about that protein sequence in Blastp algorithm format [49]. Finally the provider agent returns both, the SRS and Blastp, to the seeker agent. The sequence of the scenario is shown in Figure 6.1 and is described as follows.

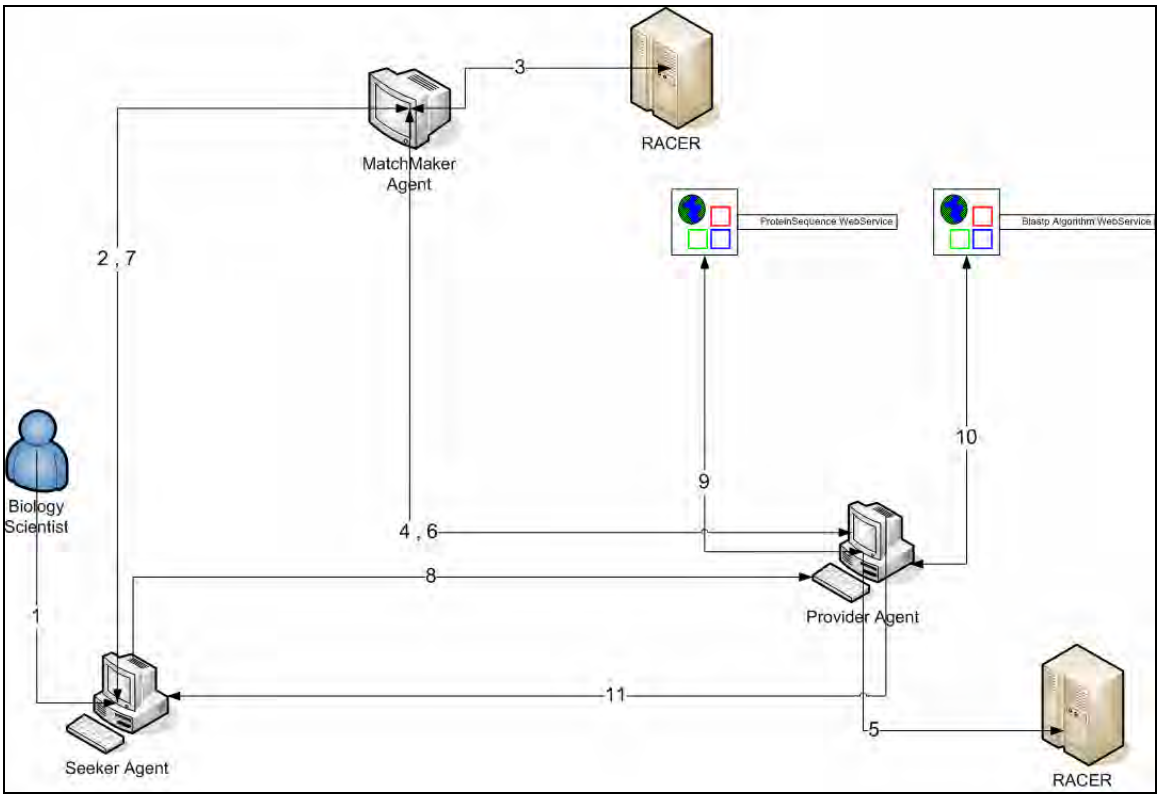

Figure 6.1 The scenario of the fungal web project

As mentioned before, the main user for the scenario will be the biologist's agent. The seeker agent will be passed two queries of the formats ConceptBasedQuery and nRQL, as shown in the Figure 6.1 (1). The ConceptBasedQuery will be taking care of filtering the number of nominated seeker agents. The nRQL will confirm the existence of the required service. Both of the queries will contain the name of the protein about which the scientists requests the information. Afterwards, the seeker agent will pass those queries to the Matchmaker agent (2). Next, the Matchmaker will check the upper ontology using the ConceptBasedQuery (3) where it will return the name of the provider agent. Subsequently, the Matchmaker confirms the existence of the required service in the provider agent by passing the nRQL query to the provider agent (4). The provider agent

poses the nRQL query to RACER that includes its specific ontology (5) and sends the name of the service back to the Matchmaker (6). The Matchmaker then finishes by sending the seeker agent both the name of the provider agent and the required service. Next, the seeker agent starts communicating with the provider agent by requesting the required input parameters for the service (8). The seeker agent sends the required parameters to the provider agent (8), which are the name of the protein and the database in this specific scenario. After the provider agent receives the parameters it sends the name of the protein and the database to the web service (9) and it returns the protein sequence. Afterwards, the provider agent uses the retrieved protein sequence by sending it as an input parameter to the second web service (10). The second web service returns the information about the required protein sequence in a BlastP algorithm (10). Finally, the provider agent sends back the protein sequence and the information in BlastP to the seeker agent (11).

The whole scenario was successfully implemented. The two webservices were found and their web service description language, WSDL, was translated into OWL-S. The upper ontology and the specific ontology for the provider agent were implemented based on what the provider agent does as mentioned before. It took a lot of effort structing the ontologies for the very specific task implemented in the solution, and a lot of help was requested from biologists in order to create the logical structure of the ontologies. Finally, the provider agent's actions and responses were programmed under DECAF's protocol. A detailed description about the implementation of this scenario is shown in Appendix B.

7. Related Work

7.1 Matchmaking for Electronic Marketplaces

In [34] an implemented prototype for matchmaking, done using Java language, is presented that can be used in the e-marketplace. It uses the technology of web services for the communication part between agents. The server that takes care of the matchmaking procedure, which will be explained later at this section, runs as a web service on top of a SOAP-enabled application server.

These web services use WSDL to describe their operation. The approach that is used to design such a framework, is to declare concepts and functionalities that are common to different types of matchmaking servers, and to define a reference model capturing these commonalities. The developer specifies and creates a set of entity types and relationships, and by combining them the developer gets the mentioned reference model. The reference model is created to fit the matchmaking modalities of a given e-marketplace. Beside the reference model, a processing model is created that defines the sequence of steps that the Matchmaker will follow when it receives a trading intention.

The authors took in consideration that each different e-marketplace domain has different schemas and complexities. In order to support that, they categorized the configurable matchmaking framework into trading domains. Each of them defines a given set of matchmaking modalities. To make the scenario complete it is suggested to have roles and

actors in the trading domain. The roles of the trading domain decide the rights of the actors registered in the domain.

Each trading domain comes with a different schema. Figure 7.1 [34] shows a schema for trading intentions for cars.

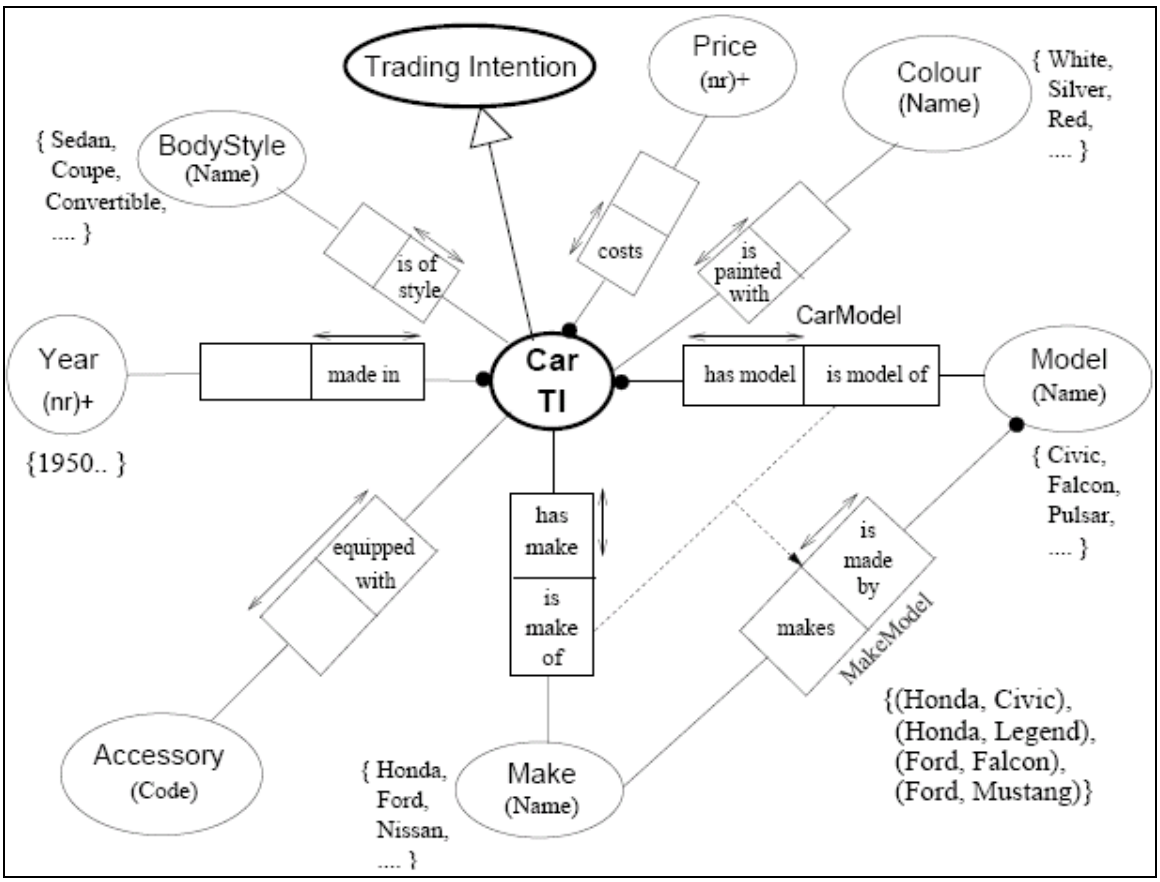

Figure 7.1: Schema for Trading Intentions for Cars [34].

In order to express certain products, or to search for one, expressions are used. Those expressions are built using literal values, properties and the following operators:

- Arithmetic operators (+, -, x, /). Expressions must be linear in the sense that a property cannot be multiplied or divided by another property.

- The “dot" operator which allows us to access the attribute values. This allows one to write path expressions when the schema involves tuple types such as Contact Information or Address. An expression that involves a sequence of applications of the “dot” is called as traversal path (e.g. contactInfo.address.streetName).
- The “includes" operator which tests for set membership.

For intance, Figure 7.2 shows an example of how a trading intention lodged by a buyer can be associated.

(Model = “Toyota Corolla" or model = “Nissan Pulsar") and year > 1995
and price < 3000 and accessories includes “CD player"

Figure 7.2: Example of searching for a product in a certain trading

7.2 Description Logics for Matchmaking of Services

The authors of [35] based the solution on Semantic Web technologies. The language used to apply the semantic web was the DAML+OIL. It was used to express service descriptions using description logic reasoners such as RACER and FaCT. The concept of match was based on using subsumption in order to find the general description matches, the more specific description matches and the compatible services. This means that the approach is totally based on T-Boxes of ontologies either in RACER or FaCT. The

matches for service description S are:

- Equivalent concepts to the service description (S)
- Sub-concepts of S
- Super-concepts of S
- Sub-concepts of any direct super-concept of S.

The suggested matchmaking service provides three basic functionalities.

The first one is *advertising*. It basically publishes the service description (as seen in Figure 7.3 [35], or the advertisement, into a matchmaking service. A check on all the concepts, confirming that all the concepts are satisfiable, has to be done before including the advertisement in the knowledge base. That action is taken in order to avoid realizing a non satisfiable service later on. When the advertisement occurs, a new set of concepts are added to the subsumption tree representing the advertised concepts.

The second functionality is *querying*. The querying functionality is very similar to the advertising functionality. The only difference is that the description submitted to the Matchmaker is not persistent; it is only used to check existing similar concepts in the tree.

The last functionality is *browsing*. It is a simple functionality that allows parties to explore the published advertisements. This functionality helps parties to modify and tune their advertisements or queries before they submit them in order to maximize the probability of matching.

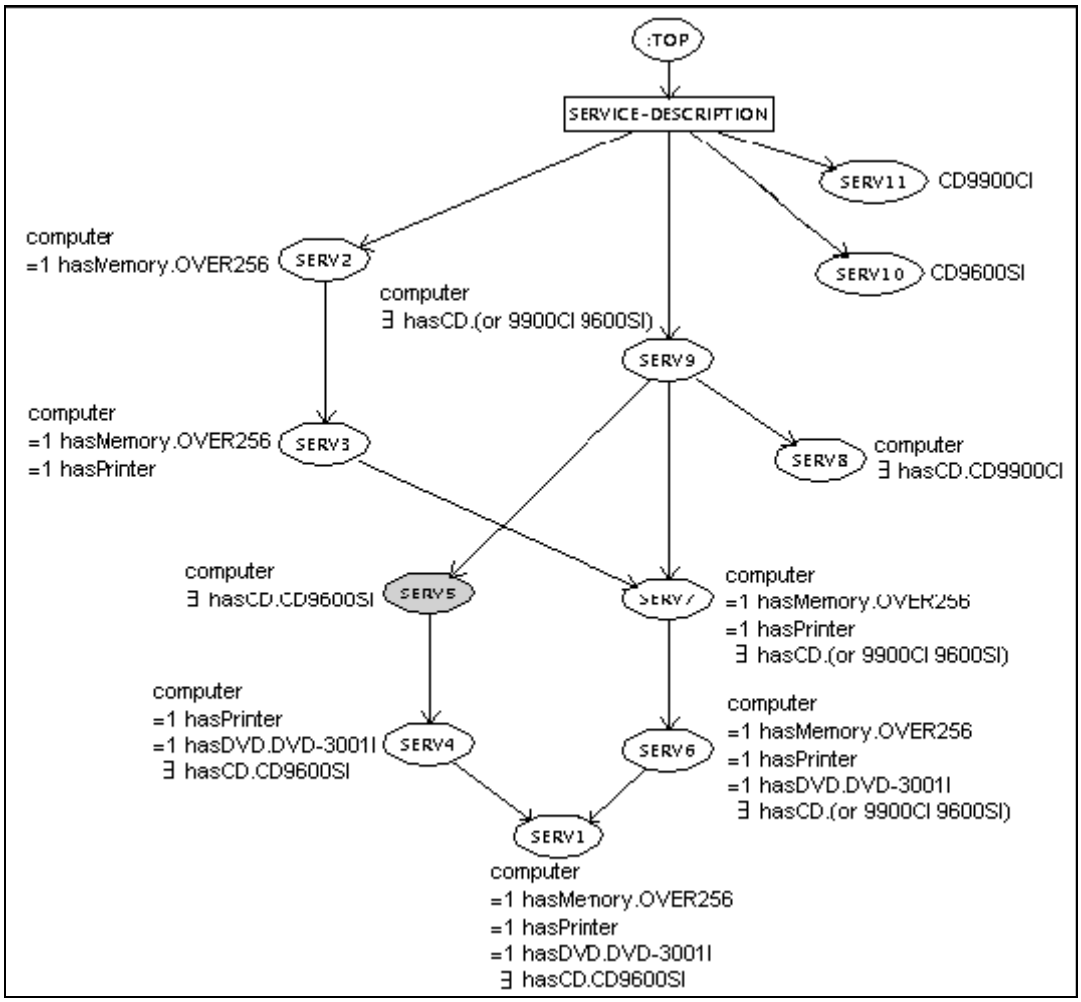

Figure 7.3: Service description branch of the subsumption tree [35].

7.3 Framework for Matchmaking Based on Semantic

In [37] service matchmaking in e-commerce is introduced, the requirements for a service description language and ontology is assessed, and the idea that DAML+OIL and DAML-S fulfill these requirements is argued. The design and implementation of the prototype Matchmaker is based on a DL reasoner to match service advertisements and requests based on the semantics of ontology based service descriptions. By representing

the semantics of service descriptions, the Matchmaker enables the behaviour of an intelligent agent to approach more closely the behaviour of a human user trying to locate suitable web services (assuming that a suitable ontology has already been developed and deployed.)

The suggested matchmaking technique was fully based on subsumption reasoning; the T-Box is queried based on the concepts of each advertising agent without querying the ABox. Therefore the level of matching had to be divided into five levels. The first level, *exact*, occurs when the advertisement A and request R are equivalent concepts. Second level, *plugIn*, occurs if the request R is a sub-concept of advertisement A. Third level, *subsume*, occurs when the request R is super-concept of advertisement A. The fourth level is *intersection*, which means that the intersection of advertisement A and request R is satisfiable. Finally the fifth level is *disjoint*, which is the opposite of *intersection*; if the intersection of advertisement A and request R is not satisfiable.

The design of the prototype Matchmaker revealed a problem with the use of DAML-S in matchmaking: DAML-S service profiles contain too much information for effective matching. This problem was solved by separating various components of the description; in particular the description of the service being provided was separated from the descriptions of the providing and requesting “actors”.

Finally, the performance of the prototype implementation was evaluated using a simple but realistic e-commerce scenario. This revealed that, although initial classification of large numbers of advertisements could be quite time consuming, subsequent matching of queries to advertisements could be performed very efficiently. On the basis of these

preliminary results, it seems possible that DL reasoning technology could cope with large scale e-commerce applications [37].

7.4 Ontology Supported Intelligent Information Agent

The authors of [38] also try to combine the power of semantic web technique in general, and ontologies in specific, in order to make the web services semantically approachable. The techniques are based on knowledge base and machine learning approaches; the agent is given a minimum of background knowledge, and the agents count on learning the appropriate behavior from the user in the long run. Another benefit from the learning approach is that it allows the agent to provide explanations for its reasoning and behavior in a familiar language for the user.

The agent in the suggested solution is also based on the mechanisms of observation, training and feedback. The observation mechanism keeps track of the documents or web services that the user usually visits in order to be used later on. The training mechanism trains the agent by inserting the subjects that the user is interested in. Those two mechanisms could make it possible for the agent to create a user profile that matches the user's request. The feedback mechanism basically takes care of directing the feedback from the user to the agent either directly or indirectly and collects the data that the agent needs to learn.

The agent records the data into a knowledge base. This knowledge base is organized

based on an ontology so it could be used for further requests.

The main implementation of the system is based on Java and OIL. It was taken in consideration that the most common way of developing a system with web services is via the Common Gateway Interface, in which procedural code is written to invoke various functions of the web protocols. Based on that, it is known that the feature of a web site that contains a CGI entrance has an “action” tag in the HTML document. In case no CGI entrance was found the suggested solution may check if that web page has a hyperlink that points to another web page. The feature of a hyperlink is having a tag that consists of “<A” and “href” attribute and ends with a closure tag “</A>” in the HTML document.

Each HTML document could have a different format from the other. Therefore it might be difficult to find the appropriate CGI entrance since some CGI programs could add different attributes such as “?keyword=” and so on. That is why the authors are still working on that problem.

7.5 Search Agent Systems for Semantic Information Retrieval

The paper [39] describes a system that retrieves information over the internet. The main idea is to be able to search the web semantically and syntactically at the same time.

Figure 7.4 [39] shows the high level architecture of the proposed prototype.

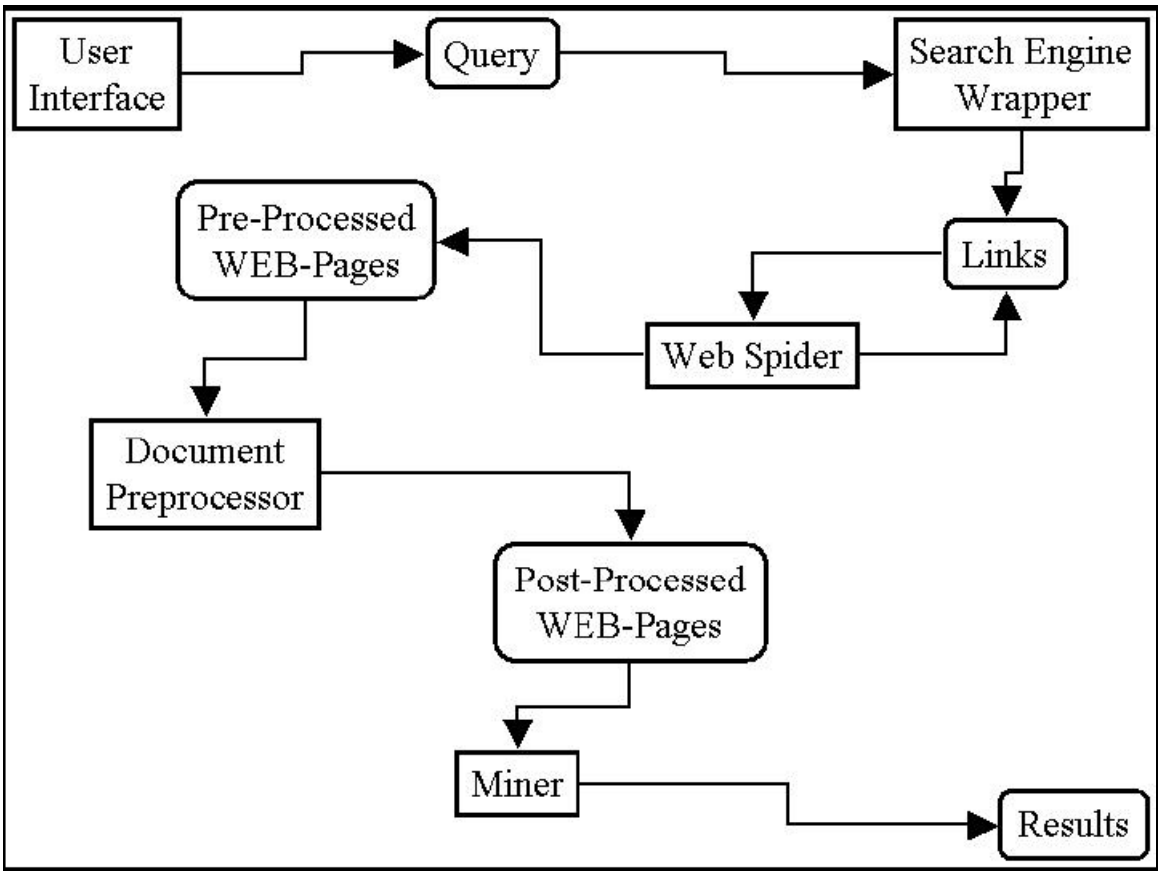

Figure 7.4: High level system architecture [39].

The architecture is based on intelligent agents. It consists of four agents, the Search Engine Wrapper (SEW), Web Spider, Document Preprocessor and the Miner.

The user submits a query to the system, using the graphical interface, apart from the keywords and the sentences that would be searched for as a whole. The user also states the depth of which each site has to be searched along with the type of language and the search area too. This query will be translated in order to be applied to many search engines through the search engine wrapper. These search engines are queried and the

required results collected and parsed in order to have the required links. Afterward the web spider downloads the collected pages, checks the links of each found web page in order to find the pages, if any existed, in case if the pages were not found by the search engines. This task continues until the search reaches the user depth specified by the user. After this step the downloaded pages are submitted to the document preprocessor where they are processed by extracting the useful data within a single document. Afterward the miner agent rates the retrieved pages with respect to their semantic similarity based on the definition of context that was retrieved from the user using a semantic knowledge base. Finally it returns the results to the user.

Each of the described works has its own unique approach. All of them follow the same principle of using the semantic web in order to improve the Matchmaking technique. In the same way, the suggested solution in this thesis has its own unique method. The suggested solution uses the combination of querying both the T-Box and A-Box at two different levels. The advantage of doing so is increasing the quality and the performance of the Matchmaking. Because by querying the T-Box and using the benefit of subsumption of concepts the solution is filtering the number of candidate agents, instead of having to ask every agent advertised in the Matchmaker. Beside that, since the T-Box is a set of definition of concepts structured in a logical way then the query does not need to have the concept being looked for as part of the query, the query can have the logical parents and features of the service and the T-Box will return the relevant concept. Querying the A-Box is an action of confirming if the candidate agent is what the seeker agent is looking for. Since querying the A-Box retrieves instances of the ontology the results will be more accurate than those retrieved from the T-Box.

8. Conclusion

8.1 Conclusion

In this thesis, a bottleneck of using web services with agents based on the existing matchmaking techniques was explained in section 2.4. The main problem of the current matchmaking techniques is that most of them are based on string comparison, which might lead to incorrect and inaccurate results. The proposed solution was to use description logic, OWL in specific, in the matchmaking procedure. By using OWL, the searching technique will not be based on string comparison; it will be based on the semantics of concepts and the relationships between concepts (i.e. if it was a subsumption relationship or a role-based relationship).

The suggested solution was fully implemented under the architecture of DECAF in Java. The description logic part was implemented in OWL using Protégé. Moreover, RACER was used for querying the designed ontologies both in the T-Box and ABox.

A real world scenario was implemented for the fungal web project. A provider agent (agent A) represented two web services and provided a service that takes a protein name as an input and returns the SRS sequence of that protein and the BlastP algorithm information. A seeker agent was searching for such a service, and by using the implemented Matchmaker, the seeker agent was able to find agent A among several other

agents based on the query and retrieved the results by executing the web services using OWL-S.

Based on the implementation that was done and from the results that were made, it makes the concept of using semantic web and description logic along web services and agents for matchmaking purposes promising.

8.2 Future Work

There are two points that could be considered as future work for this thesis. First a friendly user interface that functions as nRQL and ConceptBasedQuery generator based on simple user input taken from that interface. In that way the user would not need to have an advanced knowledge in these query languages, therefore, the solution will be easier to use and the number of users that can use it will increase.

The second point is adding the ability for the Matchmaker to use the common search engines, such as Google or Yahoo, to search for the required service in case the Matchmaker was not able to find any registered provider agent that provides that service. The queries for the common search engines could be automatically generated from the ConceptBasedQuery and nRQL query.

9 References

[1] *Tools for Developing and Monitoring Agents in Distributed Multi Agent Systems*. J. R. Graham, Daniel McHugh, Michael Mersic, Foster McGeary, M. Victoria Windley, David Cleaver, K. S. Decker. University of Delaware Newark, DE, 19716, USA, publisher: Springer-Verlag London, UK, 2000.

[2] *Ontology Development 101: A Guide to Creating Your First Ontology*. Natalya F. Noy and Deborah L. McGuinness. Stanford Knowledge Systems Laboratory Technical Report KSL-01-05 and Stanford Medical Informatics Technical Report SMI-2001-0880, March 2001.

[3] *Creating Semantic Web (OWL) Ontologies with Protégé*. Holger Knublauch, Mark A. Musen, Natasha F. Noy. Sanibel Island, Florida, USA, October, 2003. Appeared in 2nd International Semantic Web Conference (ISWC2003).

[4] *RACER User's Guide and Reference Manual Version 1.8*. Volker Haarslev and Ralf Möller. Available at http://www.racer-systems.com/ (last accessed August 14, 2005).

[5] *A High Performance Semantic Web Query Answering Engine*. M. Wessel and R. Möller. In I. Horrocks, U. Sattler, and F. Wolter, editors, Proc. International Workshop on Description Logics, 2005.

[6] *Java-based mobile agents.* Wong, D., Paciorek, N., and Moore, D. Communications of the ACM Volume 42, Number 3 Pages 92-102, Mar. 1999.

[7] *Mobile Agents for Network Management* .A. Bieszczad, B. Pagurek, and T. White IEEE Communications Surveys, vol. 1, no. 1, pages 2-9. 1998.

[8] *Concordia*. Koblick, R., Communications of the *ACM (CACM),* Vol 42, Issue 3, pps 96-97, March 1999.

[9] *An HTTP-based Infrastructure for Mobile Agent.* Anselm Lingnau, Oswald Drobnik, Peter Dömel. Fourth International World Wide Web Conference, December, 1995.

[10] *Network Modeling for Management Applications Using Intelligent Mobile Agents* White T., Pagurek B, and Bieszczad A. Journal of Network and Systems Management, September 1999.

[11] *Software Agents: A review, Technical Report*. Green, S. et al., Department of Computer Science, Trinity College, Dublin, Ireland. May 1997.

[12] *Real-Time Scheduling in Distributed Multi Agent Systems*, John Graham, Ph.D. thesis. Dissertation, University of Delaware, January, 2001.

[13] *An Introduction to Description Logics*. D. Nardi, R. J. Brachman. The Description Logic Handbook, edited by F. Baader, D. Calvanese, D.L. McGuinness, D. Nardi, P.F. Patel-Schneider, Cambridge University Press, 2003, pages 1-40.

[14] *OWL Web Ontology Language Overview.* Mc Guiness D. & ZSPLITZvan Harmelen F. (2004). W3C Recommendation http://www.w3.org/TR/owl-features/ (last accessed August 14, 2005).

[15] *Resource description framework (RDF) model and syntax specification.* O. Lassila, and R. Swick, W3C Recommendation, World Wide Web Consortium http://www.w3 .org/TR/REC-rdf-syntax, 1999.

[16] *RDF vocabulary description language 1.0: RDF Schema.* D. Brickley and R. V. Guha. W3C Recommendation 10 February 2004, 2004. available at http://www.w3.org/TR/rdf-schema/. (Last accessed August 14, 2005).

[17] *The DARPA Agent Markup Language.* J. Hendler and D. McGuinness. In IEEE Intelligent Systems Trends and Controversies, November/December 2000.

[18] *DAML+OIL (March 2001) Reference Description.* Connolly, D., van Harmelen, F., Horrocks, I., McGuinness, D. L., Patel-Schneider, P. F., und Stein, L. A.: 2001. http://www.w3.org/TR/daml+oil-reference. (last accessed August 14, 2005).

[19] *Web services glossary of W3C.* W3C Web Services Architecture Working Group. working draft 14 November 2002. Available at "*http://www.w3.org/TR/ws-gloss*/". (last accessed August 14, 2005).

[20] *XMethods* , http://www.xmethods.net/. (last accessed August 14, 2005).

[21] *GE Global eXchange Services*, http://www.gxs.com.(last accessed August 14, 2005).

[22] *Universal Resource Identifiers in WWW.* T. Barners-Lee. A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World-Wide Web. Internet Requests For Comments (RFC) 1630, June 1994.

[23] *OWL-S: Semantic Markup for Web Services*, version 1.1 D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. Mcllraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, K. Scycara: available at http://www.daml.org/services/owl-s/1.1/overview/ (last accessed August 14, 2005).

[24] *Web Service Description Language (WSDL)* Moreau, J.-J.; Schlimmer, J.: Version 1.2: Bindings. W3C Working Draft 11 June 2003. World Wide Web Consortium, Boston, USA, 2003.

[25] *OWL-S Semantic Markup for Web Services (Version 1.0),* Ankolenkar, A., Paolucci, M., Srinivasan, N., Sycara, K., Solanki, M., Lassila, O., McGuinness, D., Denker, G., Martin, D., Parsia, B., Sirin, E., Payne, T., McIlraith, **S**., Hobbs, J., Sabou, M.,

and McDermott, D.: OWL Services Coalition. 2003.

[26] *DECAF - A Flexible Multi Agent System Architecture*, John R. Graham, Keith S. Decker, and Michael Mersic. Autonomous Agents and Multi-Agent Systems, 7(1/2):7— ACM publication. 27, July-September 2003.

[27] *Requirement Specification for the DECAF Matchmaker*, Mikko Laukkanen, Jukka Eskelinen, May, 1999 Updated by Foster McGeary, June, 2000,University of Delaware, Department of Computer and Information Science

[28] *Creating semantic web contents with Protege-2000*, Noy NF, Sintek M, Decker S, Crubzy M, Fergerson RW, Musen MA. IEEE Intelligent Systems. 2001;16(2):60-71 (March/April 2001). Available from: http://computer.org/intelligent .

[29] *W3C Offices' Overview Slides*, http://www.w3.org/Consortium/Offices/Presentations (last accessed August 14, 2005)

[30] *A Semantic Web Approach to Service Description for Matchmaking of Services*. D. Trastour, C. Bartolini, and J. *Gonzalez-Castillo:* In Proceedings of the Semantic Web Working Symposium, Stanford, CA, USA, July 30 - August 1, 2001.

[31] *JavaBean-Based Simulation with a Decision Making Bean*. Miki Fukunari, Yu-liang Chi, Philip M. Wolfe. Department of Industrial and Management Systems Engineering Arizona State University U.S.A. Proceedings of the 30th conference on Winter simulation, 1998.

[32] *OWL-S API*, Maryland information and network dynamics lab semantic web agents project (mindswap) available at http://www.mindswap.org/2004/owl-s/api/, last accessed (August 14, 2005).

[33] *Class overview of OWL-S-1.0.1* . (mindswap) available at: http://www.mindswap.org/2004/owl-s/api/doc/javadoc (last accessed August 14, 2005)

[34] *A Configurable Matchmaking Framework for Electronic Marketplaces.* Dumas, Marlon and Benatallah, Boualem and Russell, Nick and Spork, Murray (2004). *Electronic Commerce Research and Applications* 3(1):95-106.

[35] *Description Logics for Matchmaking of Services*. Gonzalez-Castillo, J., Trastour, D., Bartolini, C.: In: Proc. of the Workshop on Applications of Description Logics at KI-2001, Vienna, Austria (2001)

[36] *Semantic Description of Location Based Web Services Using an Extensible Location Ontology*. Lemmens, R.L.G. and de Vries, M. (2004). In: Proceedings of Münster GI-days, 1-2 July 2004 : Geoinformation and mobility. pp. 261-262.(IfGI prints ; 22).

[37] *A Software Framework For Matchmaking Based on Semantic Web Technology*. Li, Lei and Horrocks, Ian (2003). In *Proceedings International WWW Conference,*

Budapest, Hungary.

[38] *Ontology Supported Intelligent Information Agent*. L. Weihua. In Proceedings on the First Int. IEEE Symp. on Intelligent Systems, pages 383--387. IEEE, 2002.

[39] *An Intelligent Search Agent Systems for Semantic Information Retrieval on the Internet*. Carmine Cesarano, Antonio d'Acierno, Antonio Picariello.In Proceedings of the 5th ACM international workshop on Web information and data management November 7–8, 2003, New Orleans, Louisiana, USA. Copyright 2003 ACM 1-58113-725-7/03/0011

[40] *Towards Distributed, Environment Centered Agent Framework.* John Graham and Keith Decker, Appearing in "*Intelligent Agents IV, Agent Theories,* Architectures, and Languages," Springer-Verlag, 2000, Nicholas Jennings, Yves Lesperance.

[41] *UDDI4J: Matchmaking for web services* Doug Tidwell, UDDI4J: http://www-128.ibm.com/developerworks/library/ws-uddi4j.html, IBM Developerworks, January 2001. (Last accessed August 14, 2005)

[42] *Introduction to UDDI: Important Features and Functional Concepts*, October 2004, Oganization for the Advancement of Structured Information Standards. Available at: http://uddi.org/pubs/uddi-tech-wp.pdf (last accessed August 14, 2005).

[43] *CORBA Trading Object Service*. OMG, Object Management Group and X/Open Standard, Document orbos/96-05.6, 1996.

[44] *ODP, Open Distributed Processing Reference Model*. RM-ODP. International Standard 10746-2/ITU-T Recommendation X.902.

[45] *Web services and matchmaking*. Simon Field, Yigal Hoffner. Matching Systems Ltd., Zurich, Switzerland. IBM Zurich Research Laboratory, Zurich, Switzerland Journal: International Journal of Networking and Virtual Organisations 2003 - Vol. 2, No.1 pp. 16 – 32.

[46] *RACER System Description*. Volker Haarslev, Ralf Möller. Proceedings of International Joint Conference on Automated Reasoning, IJCAR'2001, R. Goré, A. Leitsch, T. Nipkow (Eds.), June 18-23, 2001, Siena, Italy, Springer-Verlag, Berlin,pp. 701-705.

[47] *Practical reasoning for expressive description logics.* I.Horrocks, U.Sattler and S. Tobies. Proceedings of LPAR'00, vol. 1705 of LNAI, Springer, 1999

[48] *EBI Workshop for Health and Life Sciences Online.* University of Nottingham Medical School at Derby, 18/03/04 Taken from The Joint Information Systems Committee.

[49] *Basic local alignment search tool.* S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Journal of Molecular Biology, 215(3):403--410, October 1990.

[50] *The Semantic Web: an introduction*. Sean B. Palmer,2001-09. Article available at: http://infomesh.net/2001/swintro, last accessed (August 14, 2005).

[51] Tim Berners-Lee, James Hendler & Ora Lassila: "The Semantic Web". Appeared in: Scientific American 284(5):34-43 (May 2001) available at http://www.lassila.org/publications/2001/SciAm.shtml last accessed (August 14, 2005).

[52] *Simple Object Access Protocol (SOAP) 1.1*, Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, Dave Winer, Editors. World Wide Web Consortium, Note 08 May 2000. Available at http://www.w3.org/TR/2000/NOTE-SOAP-20000508/ last accessed (August 14, 2005)

[53] *Semantics for hierarchical task-network planning*. Kutluhan Erol, James Hendler, and Dana S. Nau. Technical Report CS-TR-3239, Computer Science Department, University of Maryland, 1994.

APPENDIX A: OWL-S API

```
public class RunService {

  Service service;
  Process process;
  String outValue;
  ValueMap values;
  OWLSReader reader;
  ProcessExecutionEngine exec;

  public RunService() {
    // create a generic OWL-S reader
    reader = OWLSFactory.createOWLSReader();

    // create an execution engine
    exec = OWLSFactory.createExecutionEngine();

    // Attach a listener to the execution engine
    exec.addExecutionListener(new ProcessExecutionListener() {
      String TAB = "   ";
      String indent = "";

      public void setCurrentExecuteService(Process p) {
        System.out.println(indent + "Start executing process " + p);
        indent += TAB;   }

      public void printMessage(String message) {
        if(message.equals("[DONE]"))
          indent = indent.substring(0, indent.length()-TAB.length());
        System.out.println(indent + message);      }

      public void finishExecution(int retCode) {
        System.out.println("Finished execution");   }   });    }

  public void executeService() throws Exception {
    service = reader.read(URI.create("http://WebServiceServer.com/Service.owl"));
    process = service.getProcess();

    // initialize the input values to be empty
    values = OWLSFactory.createValueMap();

    // Adding the Value "Hello World" to the first parameter of the service.
    values.setValue(process.getInputs().parameterAt(0), "Hello World");

    // Executing the Service and putting the result in "values"
    values = exec.execute(process, values);

    // get the output param using the index
    outValue = values.getValue(process.getOutputs().outputAt(0)).toString();
  }

  public static void main(String[] args) throws Exception {
    RunService Service1 = new RunService();
    Service1.executeService();
  } }
```

Figure A.1: Example of OWL-S API

As seen in Figure A.1, first of all a variable for Service, Process, ValueMap, OWLSReader and ProcessExecutionEngine has to be created [33]. The Service variable represents the OWL-S service as seen in Figure A.1. The process variable is acting as the Process in the regular OWL-S and it takes its value from the Service. ValueMap is the interface that provides a way to assign values to OWL-S parameters. When a certain process needs to be executed the values for the input parameters are specified using this interface. Also the result of that execution is given by ValueMap. OWLSReader is an interface which takes its value from the OWLSFactory, it has several methods but the most important one is the read method. The read method in OWLSReader reads the OWL-S description of the service from the given URI. It is better to have one service for every file in order to avoid any inconsistency because if the OWL-S file has more than one service the read method will randomly return the service description of one of them. In case the user wants to retrieve all of the services in a file then the method returnAll has to be used. ProcessExecutionEngine has only two methods, one is *addExecutionListener* where it gives the user the chance to add a certain behavior whenever an execution is done. The other method is *execute* which executes the OWL-S process with the given input value bindings. In case the user did not specify a ValueMap, the default value of the processed will be used if it exists.

In case of having a web service that has to be translated it into OWL-S, the user can use an existing method in OWL-S API where it takes the WSDL description of the web service and the user specifies which service he/she wants to translate, in case of having more than one service in the same WSDL file, then it automatically creates an OWL file with the OWL-S service description.

This section will show how to execute the web service based on OWL-S using the OWL-S API by explaining the simple code in Figure A.1. First of all an OWLSReader and a ProcessExecutionEngine is initiated using the OWLSFactory interface by creating a generic OWL-S reader and an execution engine. Afterward the execution listener that gets attached with any kind of execution is specified. This behaviour will occur whenever an execution command is sent. After that the service description of the OWL-S is read using the method read in OWLSReader as seen in Figure A.1, passing the location of the OWL-S OWL file. The processes are obtained from the service and specified in the process variable. By declaring these variables the users have initialized the service. What has to be done next is to initialize the input values by creating a value map that can bind an input value with the input parameter using the method setValue. Finally the service gets executed by sending the process along with the ValueMap which includes the input values with the corresponding input parameter.

APPENDIX B: FUNGAL WEB APPLICATION SCENARIO

Upper Ontology At Matchmaker

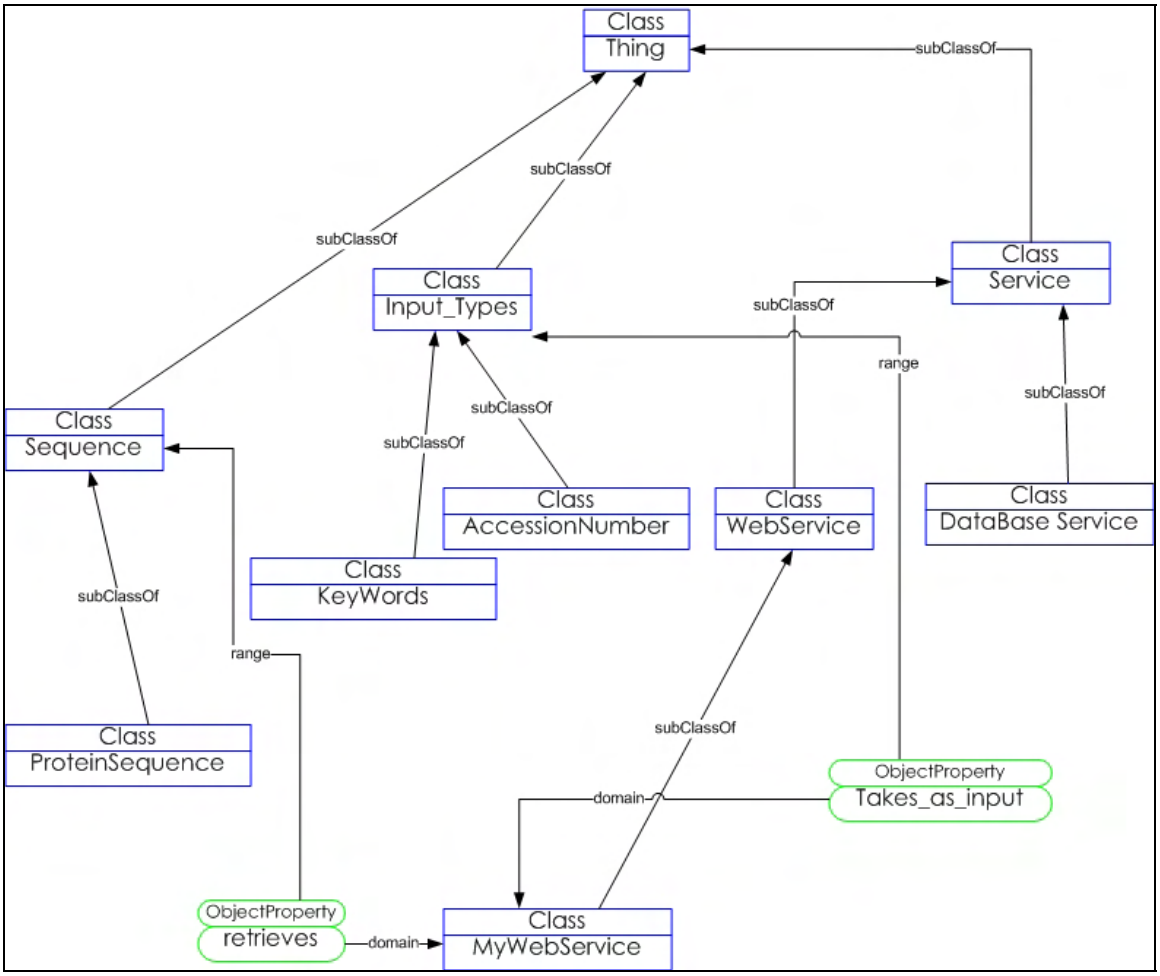

Figure B.1. Upper ontology

Figure B.1 shows the Upper ontology that was used for the Fungal Web application scenario. It shows the structure of the used concepts. The most important concept used is

“MyWebService”. “MyWebService” is the concept that the advertised agent will reach when advertising at the Matchmaker. The reason is that it subsumes the concepts that the provider agent is seeking. When the Matchmaker queries the ConceptBasedQuery ,provided by the seeker agent, it will retrieve “MyWebService” concept. Afterwards, the Matchmaker will check in its DataBase any provider agent advertising its service under that concept, which will be the SRS provider agent in our example.

Specific Ontology At Provider Agent

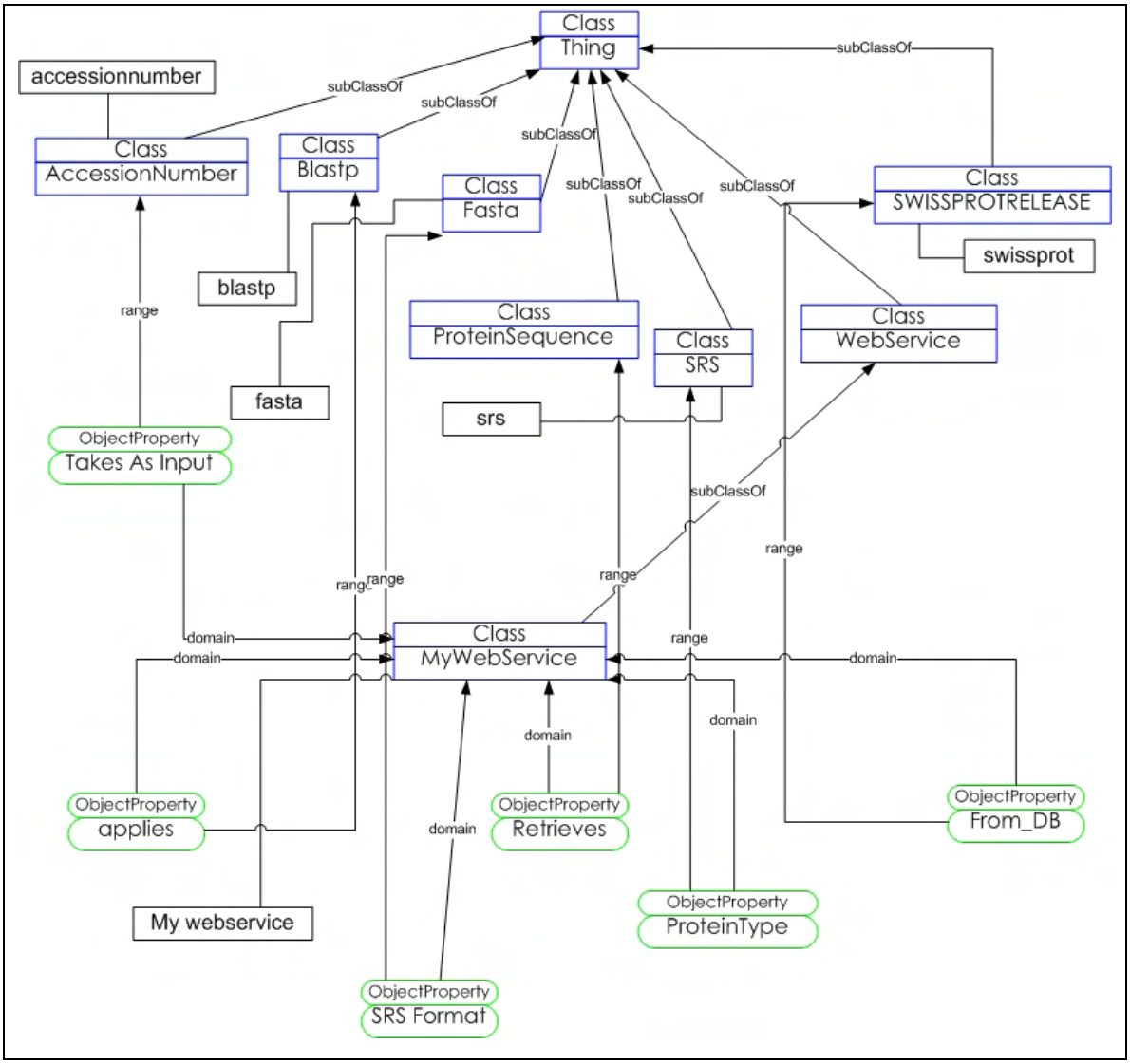

Figure B.2. The provider agent's ontology.

Figure B.2 shows the provider agent's ontology, notice the existence of instances in this ontology since the querying in this ontology will be based on the A-Box. In our scenario when the Matchmaker finds the provider agent advertising under the web service retrieved from the previous Figure. Matchmaker will send the nRQL query to the

provider agent. By its turn the provider agent will send the nRQL query to the RACER that contains this ontology and will retrieve the instance “My webservice”.

OWL-S

OWL-S is generated automatically using the API provided by mindswap. It is basically a Graphical User Interface that takes the WSDL of the web service as an input, shows the user the available web services in that WSDL file, the user chooses the required web service, and the API tool automatically generates the OWL file that represents the web service in OWL-S. Figure B.3 shows the API tool.

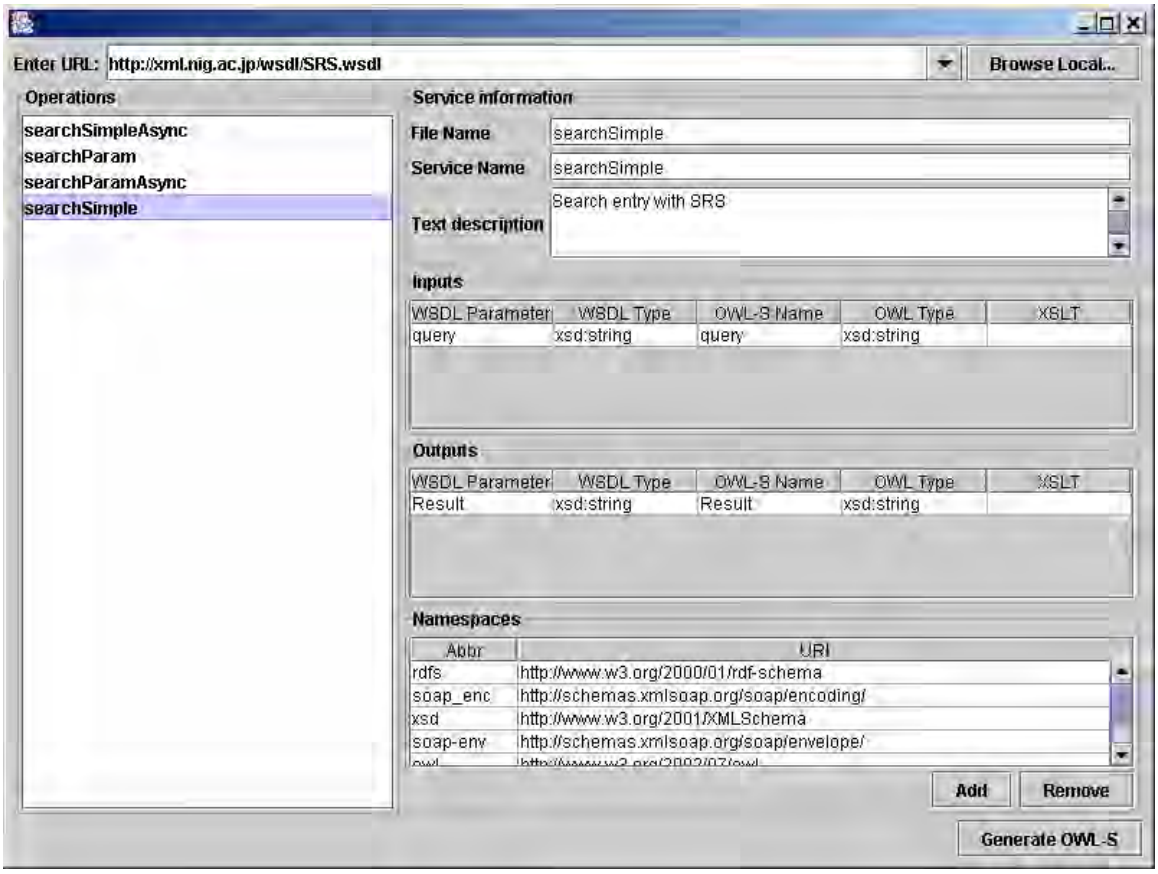

Figure B.3. OWL-S API Tool

Web Service Used

The web services that were used to achieve the required tasks were:

1) http://xml.nig.ac.jp/wsdl/SRS.wsdl

The SRS web service takes as an input the SRS query and returns the protein sequence.

2) http://xml.nig.ac.jp/wsdl/Blast.wsdl

The Blast web service takes the output of SRS web service and returns the information of the protein sequence in BlastP format.

Results Format

The format of the results is shown in the following two pages. This output is a result from an input of Accession Number = “P56547” to the agent “Fungal”. Note that the output format is only part of the complete results for demo purpose.

Beginning of Output Format

==========

| Provider executeWebservice Starting |

==========

Creating a Generic OWL-S Reader

Creating an Execution Engine

RECIEVED PARAMETERS == P56547

searchParam

Start executing process Process

[DONE]

Executed service 'searchParam'

The Result is : null>AZUR1_ALCXX Azurin I (AZN-1). 129
bpAECSVDIAGNDGMQFDKKEITVSKSCKQFTVNLKHPGKLAKNVMGHNWVLT
KQADMQGAVNDGMAAGLDNNYVKKDDARVIAHTKVIGGGETDSVTFDVSKLA
AGEDYAYFCSFPGHFALMKGVLKLVD

Sending the sequence to Blastp...

searchSimple

57:49

Start executing process Process

[DONE]

The Results is : BLASTP 2.2.6 [Apr-09-2003]

Reference: Altschul, Stephen F., Thomas L. Madden, Alejandro A. Schaffer,

Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman (1997),

"Gapped BLAST and PSI-BLAST: a new generation of protein database search

programs", Nucleic Acids Res. 25:3389-3402.

Query=

(4 letters)

Database: swiss_all.seq

188,477 sequences; 68,230,664 total letters

Searchingdone

Database: swiss_all.seq

Posted date: Aug 16, 2005 5:32 AM

Number of letters in database: 68,230,664

Number of sequences in database: 188,477

Lambda K H

0.328 0.154 0.395

Gapped

Lambda K H

0.267 0.0410 0.140

Start executing process Process

[DONE]

The Results is : BLASTP 2.2.6 [Apr-09-2003]

Reference: Altschul, Stephen F., Thomas L. Madden, Alejandro A. Schaffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman (1997), "Gapped BLAST and PSI-BLAST: a new generation of protein database search programs", Nucleic Acids Res. 25:3389-3402.

Query=

(150 letters)

Database: swiss_all.seq

188,477 sequences; 68,230,664 total letters

Searching.....done

Sequences producing significant alignments:	Score (bits)	E Value
sp|P56547|AZUR1_ALCXX Azurin I (AZN-1).	266	2e-71
sp|P00279|AZUR_ALCSP Azurin.	263	2e-70
sp|P0A320|AZUR_BORPE Azurin precursor.	214	8e-56
sp|P0A322|AZUR_BORPA Azurin precursor.	214	8e-56
sp|P0A321|AZUR_BORBR Azurin precursor.	214	8e-56
sp|P00283|AZUR_PSEDE Azurin.	199	2e-51
sp|P00282|AZUR_PSEAE Azurin precursor.	197	6e-51
sp|P00280|AZUR_ALCDE Azurin precursor.	189	2e-48
sp|P56275|AZUR2_ALCXX Azurin II (AZN-2).	186	2e-47
sp|P00286|AZUR_PSECL Azurin.	183	1e-46
sp|P00285|AZUR_PSEFC Azurin.	180	1e-45
sp|P80546|AZUR_PSEFA Azurin.	177	7e-45

…….

End of Output Format