An agent-based system for Web Services provision and selection, using semantic markups

W I L L I A M   G R O L E A U

Master of Science Thesis
Stockholm, Sweden 2005

IMIT/LECS-2005-72

An agent-based system for Web Services provision and selection, using semantic markups

William Groleau

Examiner
Assoc. Prof. Vladimir Vlassov
(IMIT/KTH)

Master of Science Thesis
Stockholm, Sweden 2005

IMIT/LECS-2005-72

Abstract

We have seen the number of available web-services on the web dramatically increasing for the past few years, as well as the growth of public interest in accessing these services. Internet is hence becoming a marketplace where providers and requesters need to be able to work altogether for the sake of each other interests. Furthermore, providers and requesters do not need to be necessarily humans, but can also represent automats looking for services to build composite services. A system, *"An agent-based system for Grid services provision and selection",* has been brought up, fulfilling two purposes: bringing requesters and providers together in a marketplace and achieving a certain degree of automatic interoperability between services for automatic composition purposes. The system consists of an agent architecture working as a marketplace where service providers and requestors can meet and negotiate about services. Users specify their requirements to agents, which start negotiating with other agents provisioning services. Nonetheless, the services used by the system cruelly lack semantic markup, which can be added with the use of OWL-S which allows unambiguous interpretation of web resources and content through the use of shared web ontologies. In this work, we see how semantics have to be added to the system, *"An agent-based system for Grid services provision and selection",* by the use of OWL-S, and we do not restrict anymore the system to Grid Services. We also particularly study the matchmaking of services issue, by introducing new and more powerful procedures allowing finer matchmaking of services, designed to allow requesters to find Web Services satisfying precise requirements.

The main result of this work is thus a prototype of the upgraded version of the mentioned system. Two matchmaking methods have been implemented in order to assess which of those is the most appropriate. Our proposed prototype also includes new working modes: one to ensure the requester's privacy and a second to improve the efficiency of the system. The description of the implementation is also given in this work.

Contents

Chapter 1. Introduction ... 1
1.1. Preliminary definitions & concepts ... 1
1.1.1. Web services ... 1
1.1.2. Semantics ... 1
1.1.3. Requesting & Providing services : a motivating example ... 2
1.2. Project goals ... 3
1.3. Outline ... 3
Chapter 2. Web services & Grid Services ... 5
2.1. Web-services basic activities ... 5
2.1.1. Description (WSDL) ... 5
2.1.2. Publishing & Discovery (UDDI) ... 5
2.1.3. Binding & Evocation (SOAP) ... 6
2.2. Grid services & Stateful web-services ... 7
2.2.1. Stateful Web-services vs. Stateless Web-services ... 7
2.2.2. Grid Services by OGSI ... 7
2.2.3. From OGSI to WS-RF ... 8
2.2.4. Globus Toolkit 4 ... 8
Chapter 3. Semantics ... 11
3.1. Introduction ... 11
3.1.1. Background ... 11
3.1.2. OWL & OWL-S ... 12
3.1.3. Reasoning ... 14
3.2. Reasoning ... 14
3.2.1. JESS approach ... 14
3.2.2. JENA approach ... 15
3.2.3. Comparing approaches ... 16
Chapter 4. Services discovery approaches ... 18
4.1. Matchmaking ... 18
4.1.1. Matching service profiles ... 18
4.1.2. Matching service models ... 20
4.1.3. Comparison ... 21
4.2. Frameworks ... 22
4.2.1. Introduction to Agents & Multi-Agents Systems ... 22
4.2.2. Agent-based system for Grid services provision and selection ... 23
4.2.3. UDDI-based discovery systems ... 24
4.2.4. Others ... 25

Chapter 5. Proposed Solution – Design 26
5.1. Preliminary Considerations 26
5.1.1. Terminology 26
5.1.2. Scenario of system use 26
5.1.3. System needs 28
5.2. Model of the application 28
5.2.1. Use Cases 28
5.2.2. Interaction between parts (sequence diagrams) 29
5.3. Algorithms 33
5.3.1. Profile matching 33
5.3.2. Model matching 36
5.3.3. Filtering SPAs & Secure mode filter 39
Chapter 6. Implementation 41
6.1. Agents 41
6.1.1. SPA 41
6.1.2. SSA 42
6.1.3. EDF 44
6.2. Other packages 45
6.2.1. matcher 45
6.2.2. reasoner 46
6.2.3. storage 46
6.2.4. content 47
6.3. Manual 48
6.3.1. Required software and libraries 48
6.3.2. Configuration 49
6.3.3. GUIs & Application example 50
Chapter 7. Proposed Solution – Evaluation 55
7.1. Time performances 55
7.1.1. Matchmaking 55
7.1.2. Application 58
7.2. Evaluating accuracy 59
Chapter 8. Conclusion & Future Works 60
A. Bibliography 62
B. Abbreviations 63
C. OWL services used in the application example of Section 6.3.3 64

List of Figures

Figure 2-1 Web-services lifecycle 5
Figure 2-2 Interaction scheme in a Web-services infrastructure 5
Figure 2-3 Web-services scenario with relations between WSDL, SOAP and UDDI [21] 7
Figure 2-4 Relationship between OGSA, GT4, WS-RF, and Web Services 9
Figure 2-5 GT4 Architecture 10
Figure 3-1 A fragment of the Vehicle ontology........ 11
Figure 3-2 Top level of the service ontology........ 13
Figure 4-1 Algorithm for output matching 19
Figure 4-2 Rules for the degree of match assignment 19
Figure 4-3 Algorithm matching outputs of either a split or a sequence node........ 21
Figure 4-4 Collaboration diagram of the service providing part of the system........ 23
Figure 4-5 Collaboration diagram of the service selecting part of the system 23
Figure 4-6 Architecture of the DAML-S/UDDI matchmaker in [10]........ 24
Figure 5-1 Illustration of the scenario of system use........ 27
Figure 5-2 Use Cases of the system (from [1])........ 29
Figure 5-3 adding a service to an SPA........ 30
Figure 5-4 Searching services ("regular" mode) 31
Figure 5-5 Searching services ("secure" mode)........ 32
Figure 5-6 Searching services ("Filter SPAs" mode) 33
Figure 5-7 Service matching algorithm 34
Figure 5-8 Sample OWL-S input parameter........ 34
Figure 5-9 Outputs matching algorithm........ 35
Figure 5-10 Algorithm assigning a matching score for two parameters........ 36
Figure 5-11 Model Matchmaker algorithm........ 37
Figure 5-12 Function selecting the appropriate algorithm for the corresponding node ... 38
Figure 5-13 Algorithm used to match either Split or Sequence Nodes 38
Figure 5-14 Category matcher algorithm........ 40
Figure 6-1 Method initializing a service search........ 43
Figure 6-2 Behaviors flow of the service search 44
Figure 6-5 Excerpt of the advertisement's profile........ 51
Figure 6-6 Excerpt of the request's profile........ 52
Figure 6-7 Service Selection Agent GUI 53
Figure 6-8 Service Provision Agent GUI........ 54
Figure 7-1 Services parsing time chart 56
Figure 7-2 Class getting time chart........ 56
Figure 7-3 Relationships inferring time chart........ 57
Figure 7-4 Time repartition in a matchmaking process........ 58

List of Tables

Table 3-1 Composition of the OWL Service Profile 14
Table 3-2 OWL Reasoner comparison 17
Table 4-1 Rankings for the matching of two parameters........ 20
Table 4-2 OWL services matcher comparison........ 22
Table 5-1 Glossary of recurrent terms 26
Table 6-1 Behaviors of the Service Provision Agent........ 42
Table 6-2 Behaviors of the Service Selection Agent 43
Table 6-3 Behaviors of the Extended Directory Facilitator........ 45
Table 6-4 Reasoner interface 46
Table 6-5 Matching interface........ 46
Table 7-1 List of services used for testing purposes........ 55

Chapter 1. Introduction

This work is based on a previous project, *an agent-based system for grid services provision and selection* in [1], and is included in a more important research framework: web and grid services automatic composition, for which it brings an essential basis. The project will focus on adding semantics to the selection and provision system described in [1], which will lead us to bring some changes to the architecture in order to host new semantic reasoning and matchmaking engines. As the matchmaking process is an important part of the services discovery problem, it will be one of our main focuses. This work will then include the development of a prototype platform supporting web services and grid services provision and selection, thanks to a matchmaking engine reasoning over the semantics of the services.

We are now about to show an example motivating the web services automatic selection, and motivating the automatic composition and we will then expose the goal of this project, before presenting the outline of this work.

1.1. Preliminary definitions & concepts

Before getting in details in the various technologies concerned, we need to clarify some of the basic concepts, such as web services, semantics, and the general context of providing and requesting web services.

1.1.1. Web services

A Web service is a modular, self-describing, self-contained unit of application logic fulfilling atomic tasks and is accessible by Internet through standard XML protocols and format. The use of standard XML protocols makes Web Services platform, language, and vendor independent. They eliminate the interoperability issues of existing distributed technology, such as CORBA and DCOM, by leveraging open Internet standards - Web Services Description Language (WSDL - to describe), Universal Description, Discovery and Integration (UDDI - to advertise and syndicate), Simple Object Access Protocol (SOAP - to communicate). All these standards will be briefly discussed in chapter 2. The advantages of web services are their simplicity, their use of open standards, their flexibility (loose coupling between application publishing and applications using the services), and their efficiency. However, standard web-services appear to be limited to human use, hence the necessity to make them usable by machines to automate various tasks.

1.1.2. Semantics

Information on the web has been primarily designed for human interpretation and use, making this information totally unusable by computers. Expressing information on the web in computer understandable form is an interesting challenge since it could allow different kind of interactions between machines; any human intervention would not be needed and machines could be able to autonomously cooperate through the Internet.

Hence, knowledge needs to be formalized and information on the web needs to be annotated with semantic information so that human and computers share the meaning of the information exchanged along with the information exchanged. Globally, syntactically different terms can be used by service requesters and service providers as long as they both agree on their semantic meaning.

All semantic reasoning is based on the important concept of ontology, hierarchies linking concepts altogether and expressing relationships between them. For instance, the ontology of vehicles could express that *Ford* is a sub-concept of *car,* and so that *ford* could somehow be replaced by *car*, in an attempt to find any particular service. Semantics could then allow more complex services discovery and could replace limited keyword look-ups. We will explore and present these possibilities in details in Section 1.1.3, showing the motivation for our project. We will see in Chapter 3 the languages used to express ontology and the languages used to describe services with semantic information, key element to the automatic service discovery issue we are now about to look closer.

1.1.3. Requesting & Providing services : a motivating example

As we saw in the previous Section, automatic services discovery seems to be an interesting issue, as with the increasing growth in popularity of Web services, discovery of relevant Web services becomes a significant challenge. Before pursuing, let us clarify the global organization of a discovery framework. These frameworks are divided in three parts:

- *Service provider.* A service provider owns services and holds their implementation, and generally stands for a company willing to advertise services.
- *Service requester*. Entity, human or not, having a need potentially achievable by advertised services. It will look for services and eventually evoke them.
- *Service broker.* A service broker stores the description of the services brought by service providers and will direct a service requester to appropriate providers holding services fulfilling the request. The service broker and service provider can somehow be the same entity, i.e. the service provider matches a request with its stored services, as in the architecture in [1]. This part is the main focus of our work, since it is in charge of matching relevant advertised services with user's need.

A service discovery framework finds all its interest in the fact that it can provide users – or machines – with an elaborate way of finding relevant services, enhancing existing standard registries. Let us say you want to invoke a service allowing you to rent cars online. You first need to locate services fulfilling your wish. Within a regular UDDI framework, the only way to look for these services is to browse the registry, filtering it with keywords such as 'car', arental' … A UDDI registry will probably have entries corresponding to the desired service but will also contain entries – apparently – matching your need, e.g. a travel agency proposing attractive car rentals within the condition that you book a flight or a hotel with them, or professional vehicle rentals services, for example. As a result, the filter will return relevant services, drowned in an important number of non relevant services. Using a service discovery framework like the one of our work, instead of fetching services by keywords, one could fetch a service by specifying the expected input parameters (type of car, location, maximum price, date …) and the expected output parameters (type of car, date, price …). This way, services having incoherent or additional input parameters (in our example, flight or hotel information) would not match the request and would be pruned, leaving the user with only relevant services. Synthetically, our task will be to augment the possibilities of UDDI, presently unable to store semantic information thus unable to allow powerful searches (in Section 2.1.2 we will see in details UDDI registries).

However, single web-services may not satisfy users' requests, thus mechanisms should be developed in order to build composite web-services, satisfying more complex requests. Due to the high number of available web-services and to the complexity of the task, composition of web-services is, for a human-being, almost infeasible. Therefore, there should be mechanisms to automatically combine existing services together in order to fulfill users' requests. These mechanisms should take into account that web-services are updated "on-the-fly", and should auto-update their data at runtime so to base the decisions upon up-to-date information. Automatic composition would be of great interest for the case where, for instance, you expect the output price of your car rental service to be in euros whereas all the services return prices in Swedish crowns. In such a case, the automatic composition system would search by itself "value-added" web-services converting euros into Swedish crowns, thus removing any needed human interaction. Automatic composition is out of the scope of this work, but our project is the base for any composition system, as it would allow services composer to automatically fetch appropriate services in the composition process. We are now about to see the goal and the scope of our work.

1.2. Project goals

The ambition of this project is to solve the sub-problem of providing a system for advertising and requesting semantic web services and grid services. An important issue of this system is the matchmaking between a requested service and a service previously advertised. By adding semantic to services in the framework, discovery of services is to be used not only by humans, but also – and essentially – by computers, enabling more ambitious purposes, such as automatic composition of services. The implication of this work in this latter purpose then becomes obvious; an intelligent program could use the platform to perform searches for the value-added services needed in the dynamically built composite service. Besides, the problem of automatic web composition has already been addressed in various systems, mainly using AI Planning ([2], [5] and [6]).

This thesis will also provide a short survey on different services discovery frameworks and matchmaking techniques. It will also provide a prototype of the system described below.

1.3. Outline

Chapter 2 provides a deeper overview of the web services and grid services technology. Themes of description, publication, discovery, and evocation of services are approached, along with the correspondent technologies, WSDL, UDDI, and SOAP.

Chapter 3 provides an approach to technologies adding semantics to the web and more specifically to web services, with a presentation of languages such as OWL, and tools needed to work with these languages.

Chapter 4 provides a view of the existing services discovery frameworks, and show different matchmaking techniques.

Chapter 5 covers the design part of our proposed architecture for provisioning and matching services.

Chapter 6 demonstrates the implementation issues of our system and shows an application example with snapshots

Chapter 7 evaluates the developed prototype.

Chapter 8 summarizes and discusses the method and future directions for this work.

Chapter 2. Web services & Grid Services

We are now about to give a more precise overview of the web services technology, covering their four basic activities: description, publication, discovery, and evocation. We will then have a look on a particular form of web services: grid services, and see what differentiates them from basic services.

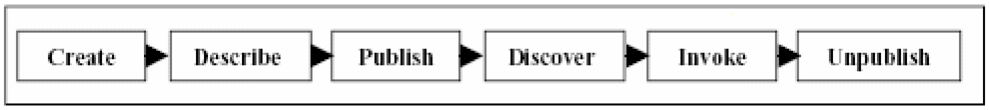


Figure 2-1 Web-services lifecycle

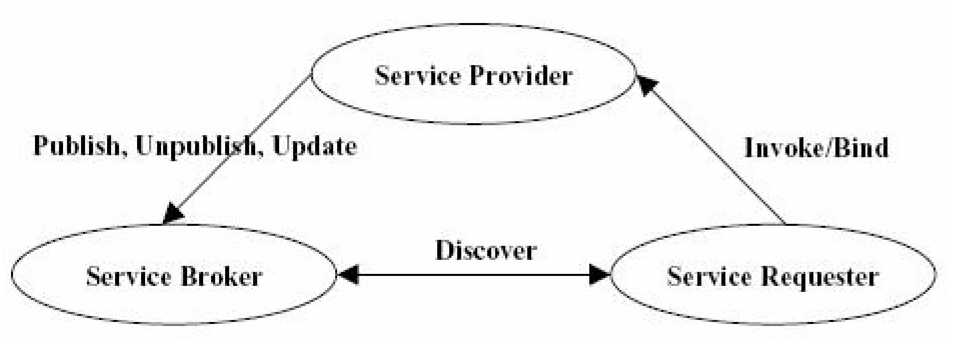


Figure 2-2 Interaction scheme in a Web-services infrastructure

2.1. Web-services basic activities

2.1.1. Description (WSDL)

In order to be classified, stored, discovered and used, services need to be described, including functional descriptions (operations provided, messages exchanged, binding information) as well as non-functional descriptions (documentation, security issues …). The standard used to describe is actually the XML based WSDL (Web Services Description Language) and can be compared to the distributed programming IDL, i.e. it serves as a programmatic interface to web services, and specifies their properties: what the service does, where it is located and how it is invoked.

2.1.2. Publishing & Discovery (UDDI)

Publishing is a Mandatory activity to make services available and ready for use. Publishers can provide description of their services to public registries available on the Internet, such as UDDI. Information about services and businesses can be published through UDDI, which can be browsed on the Internet by any user looking for particular services – or for services provided by a specific company. A UDDI response will provide all technical details required to interact with a service. A UDDI registry is nothing more than an XML schema defining four key data structures:

- *Business entities* describe information about businesses (name, description, services offered and contact information).
- *Business services* provide more details on each service being offered. Each service can have multiple binding templates.
- *Binding templates* describe a technical entry point for the service (e.g., mailto, http, ftp, etc.).
- *tModels* describe what particular specifications or standards a service uses.

The *tModel* key structure of UDDI partially addresses the problem of classification, as it allows abstractions on specifications. One could use the *categoryBag* element of the *tModel* to categorize services in a Yahoo!-like classification scheme. For instance, to find skateboards manufacturers, one would look into the directory:
"Business_and_Economy/Shopping_and_Services/Sports/Skateboarding/Deck_and_Truck_Manufacturers/"
A skate selling service would then only have to "register" itself as part of that directory in a potential Yahoo! Business Taxonomy *tModel*, so users browsing in the previous directory could find their services.

The *discovery* activity is the counterpart of the discovery, thus is highly dependant on the technique used to publish services. The basic way to discover services remains the usage of UDDI, which only allows discovering through keywords corresponding to the key structures of the registry (search by business entities, by services keywords or description, templates …). The subject of our work is precisely aimed at improving this discovery process, lacking the semantics needed for powerful searches.

2.1.3. Binding & Evocation (SOAP)

Binding and evoking services occurs once a requester has found which service to call. Binding services refers to the question "*how* to call the service?" which is selecting an entry-point for the service (http, ftp …). Once an entry-point is set, the client and the service provider communicate by exchanging SOAP messages, XML-based standard for making remote procedure calls. Its header can contain various information which can be needed by some of the entities relaying a SOAP messages. For instance, the WS-Addressing uses the SOAP header to address web-services and messages, particularly WS-Resources (see Section 2.2.3). Figure 2-3 below details Figure 2-2, and represents a more detailed scenario of interactions within a web-service framework.

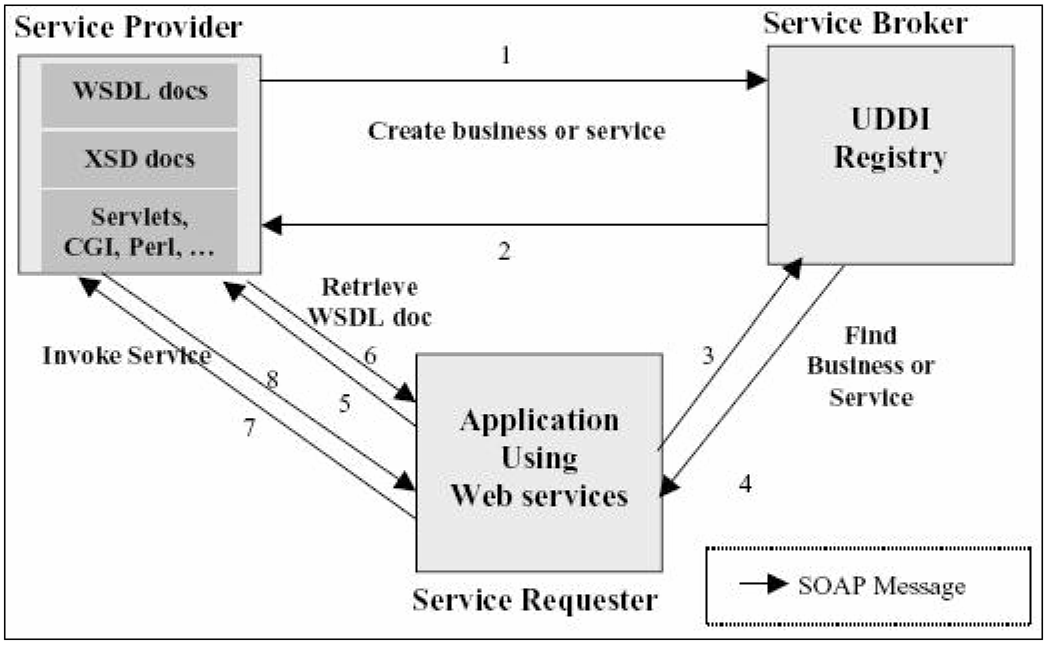


Figure 2-3 Web-services scenario with relations between WSDL, SOAP and UDDI [21]

2.2. Grid services & Stateful web-services

2.2.1. Stateful Web-services vs. Stateless Web-services

Web-services we have discussed earlier were implicitly stateless, as are commonly web-services. Stateless means that the service simply gets its task done and then disconnects from the client or from whosoever, thus keeping no state information from one invocation to another. This behavior is common and keeping information through different invocations is usually pointless (e.g. a weather service only provides temperature information when asked and does not need to remember any information between two invocations). Nonetheless, some applications, especially grid applications, require statefulness. Let us consider an example of stateful web-service. Suppose we want a service acting as an integer accumulator, which means that each call to the service would increase the accumulator by the value passed in parameter to that service. For instance, the first invocation of the service with 5 as parameter will return 5, the second call with 2 as parameter will return 7, the third call with 4 returns 11 and etc. In that case, the service remembers a state (i.e. the value of the accumulator), and is then called *stateful web-service*. The state is kept, not in the web-service itself, but in a separate entity, called *resource*. A resource can be a single integer as below or can have a more complicated structure. One web-service can have several resources associated, and each of them can be accessed thanks to their unique identifier, which has to be specified using WS-Addressing in the SOAP header.

2.2.2. Grid Services by OGSI

OGSI (Open Grid Services Infrastructure, [11]) defined mechanisms for creating and managing so called *grid-services*. A fuller presentation is available in [1], we will limit ourselves here to a short introduction as this infrastructure is becoming obsolete and is being replaced by the WS-Resource Framework (WS-RF, [3]). Briefly, OGSI is an infrastructure focused on WSDL interfaces for grid-services and sets a new transient standard to extend WSDL: GWSDL (Grid-WSDL). GWSDL allows services to be described by Service Data Elements (SDE), which brings additional description information such as classification or taxonomy. OGSI also defined a set of predefined portTypes (GridService, HandleResolver, NotificationSink …), of which at least one had to be defined by the grid-service. Additionally, OGSI provided mechanisms for managing services' lifetime, as WS-RF does, as we will now see.

2.2.3. From OGSI to WS-RF

The WS-Resource Framework (WS-RF, [3]) is an effort to merge the two concepts of web-services and grid-services, by introducing a new concept, the *WS-Resource*. A so called *WS-Resource* is an association of a (stateless) web-service and a resource (as described earlier in Section 2.2.1). As a consequence, the OGSI SDEs are now replaced by the *resource properties* (attributes of the resource) associated with a service and the transient standard GWSDL has been discontinued for WSDL 1.1 (while waiting for WSDL 2 release). Moreover, the OGSI two-level naming scheme (GSH, GSR) is now reduced to an endpoint reference optionally including a reference to a particular resource, and in addition, new specifications have been provided for the lifetime management of resources. So basically, we can assume that the names of the concepts have changed, but the concepts themselves remain the same. We can now consider the WS-RF specification, gathering five different specifications, all related to the management of WS-Resources:

- WS-ResourceProperties. A resource is composed of zero or more resource properties (resource properties are the "attributes" of the resource(s) associated with the service, e.g. a filename, file size …). WS-ResourceProperties is the specification defining how resource properties are defined and accessed.
- WS-ResourceLifetime. Resources can be created and destroyed at any time. The WS-ResourceLifetime supplies some basic mechanisms to manage the lifecycle of resources.
- WS-ServiceGroup. This specification provides functionalities to manage groups of WS-Resources, and provide an entry-point to groups of resources.
- WS-BaseFaults. This specification provides a standard way of reporting faults.

We will now explore the possibilities of the Globus Toolkit, which implements the WS-RF specifications.

2.2.4. Globus Toolkit 4

The Globus Toolkit is a software toolkit, developed by The Globus Alliance, which is used to program grid-based applications. It also essentially provides high-level services, such as resource monitoring and discovery, security infrastructure, or data management services. GT4, the latest release, includes a complete implementation of the WS-RF specification, and all material provided by the toolkit is built on top of that implementation.

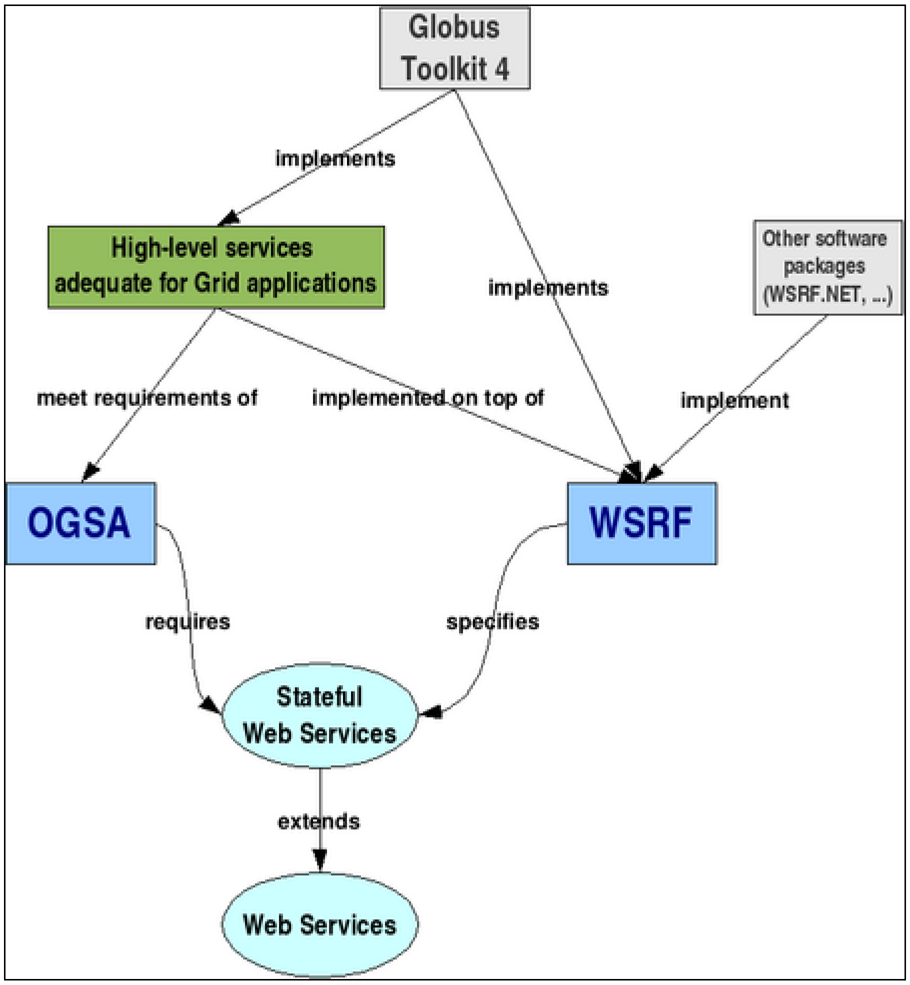


Figure 2-4 Relationship between OGSA, GT4, WS-RF, and Web Services

GT4 provided software components are divided into five categories:

- *Common Runtime*. The Common Runtime components provide a set of fundamental libraries and tools which are needed to build both WS and non-WS services.
- *Security*. Security components ensure secure communications.
- *Data management*. These components allow to manage large sets of data in virtual organizations.
- *Information services*. The Information Services, more commonly referred to as the Monitoring and Discovery Services (MDS), includes a set of components to discover and monitor resources in a virtual organization.
- *Execution management*. Execution Management components deal with the initiation, monitoring, management, scheduling and coordination of executable programs, usually called jobs, in a Grid.

The following figure (Figure 2-5) all the various components contained in each of these five categories:

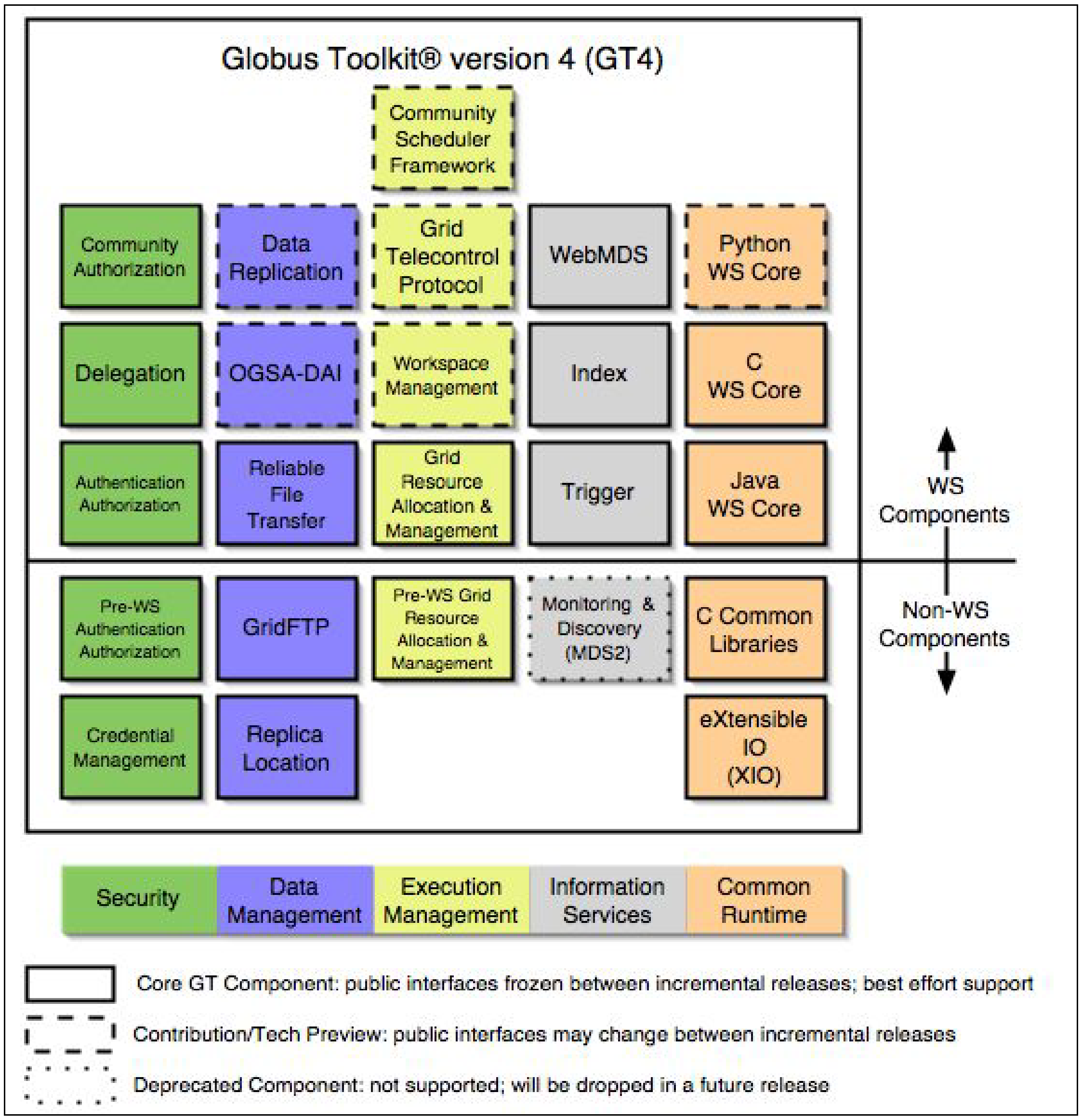

Globus Toolkit® version 4 (GT4)
Community Scheduler Framework
Community Authorization
Data Replication
Grid Telecontrol Protocol
WebMDS
Python WS Core
Delegation
OGSA-DAI
Workspace Management
Index
C WS Core
Authentication Authorization
Reliable File Transfer
Grid Resource Allocation & Management
Trigger
Java WS Core
WS Components
Non-WS Components
Pre-WS Authentication Authorization
GridFTP
Pre-WS Grid Resource Allocation & Management
Monitoring & Discovery (MDS2)
C Common Libraries
Credential Management
Replica Location
eXtensible IO (XIO)
Security
Data Management
Execution Management
Information Services
Common Runtime
Core GT Component: public interfaces frozen between incremental releases; best effort support
Contribution/Tech Preview: public interfaces may change between incremental releases
Deprecated Component: not supported; will be dropped in a future release

Figure 2-5 GT4 Architecture

It appears that the Monitoring & Discovery Service framework could be of some use in a discovery process. It includes WS-RF implementations of the Index Service and a Trigger Service. An MDS server can be used to keep resources status information, and to query resource contents. Basically, this is a point of entry for groups of resources which allows querying over these resources for discovery purposes for instance. Note that the discovery of resources is rather elementary and does not allow querying on any meta-information. Note also that the purpose of MDS is managing information of a computational grid (knowing which resource is available, knowing the state of the grid …) and not discovering WS-Resources, i.e. discovering web-services.

MDS has been widely used in the Grid community for resource discovery as UDDI has been used in the web community for business service discovery. However, both MDS and UDDI only support simple query languages and do not offer expressive description facilities, nor provide sophisticated matchmaking capabilities. This is the reason why we have to come up with a system enabling advanced matchmaking.

Chapter 3. Semantics

As seen in the first chapter, semantic web is a way to give explicit meaning to information in order to make it easier for machines to process data available on the web. In the first part, we will introduce the needed concepts and languages needed to work with ontologies, and the second part will focus on reasoning approaches and algorithms.

3.1. Introduction

3.1.1. Background

We have already stated the needs for a semantic web in Section 1.1.2, which can be summed up by saying there is a want for making data available for non-human use. Web services would profit from semantics as it would fulfill an early requirement for UDDI: a way to perform intelligent searches. Until the semantic problem was addressed in a satisfactory manner, the best mechanism to facilitate such searches was through taxonomic categorization and classification potentially allowed by the *tModel* key structure (see Section 2.1.2). This metadata information stored in UDDI is a first step toward more efficient look-ups, yet insufficient to complete precise services searches. The concept of ontology we are about to develop is close to these concepts of categorization found in the *tModels*, and we will see in the next Section languages and concepts used to solve these problems of classification and meaning agreements.

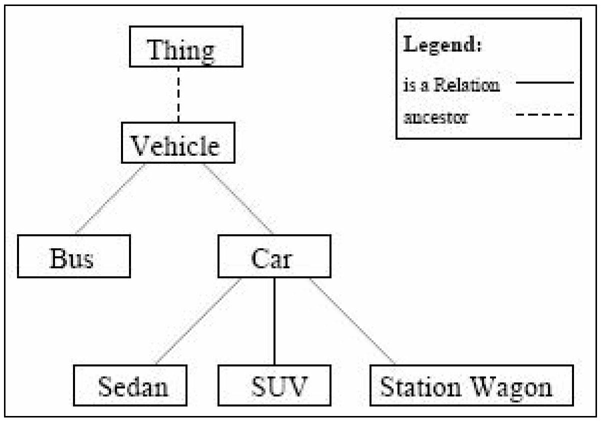


Figure 3-1 A fragment of the Vehicle ontology

An ontology defines the terms used to describe and represent an area of knowledge, and is meant to translate an explicit general agreement. We can see in the fragment of the vehicle ontology in Figure 3-1 (taken from [10]) some of the necessary DL (Description Logics, see [23]) knowledge:

- Class, or *concepts*, definitions (e.g. *Car* is a subclass of *vehicle*; *SUV* is a subclass of *Car* …), also known as *ABox* definitions.

- Not shown here, classes can be described by *properties*, or in DL terminology, *RBox* containing axioms about *properties*, *e.g.*, that *property* P is a *subProperty* of R.
- Not shown here, *classes* can be "instantiated", we call *individual* an instance of a *concept*, and in DL terminology an *ABox* contains assertions about *individuals*, either that an *individual* is a member of some *class*, or related by a *property* to some other *individual*.

3.1.2. OWL & OWL-S

To model ontologies, the W3C is proposing a standard known as OWL (Web Ontology Language, [4]), to be used in situations where the content need more than just being showed to humans. OWL is based on the language DAML+OIL, a combination of the languages DAML (Darpa Agent Markup Language) and OIL (Ontology Inference Layer), itself built on top of RDFS (Resource Description Framework for Services). OWL brings an extension to RDFS by providing richer modeling primitives and brings the equivalent of a Description Logics with XML syntax.

Ontology languages give us the possibility to reason on classes, properties and individuals (instances of classes). Individuals are described by properties, which in turn are described by properties (the properties of properties or the characteristics of properties). In addition, restrictions (cardinality, range of values) can be placed on how properties can be used by instances of a class. Next, OWL allows more complex class construction by defining union, interSection, and various operations on classes. Let us consider now the three characteristics elements of OWL.

- *Class*. A class stands for a *concept* (vehicle, car …) and is characterized by axioms and can be defined by a combinations of other *classes*:
  - Axioms: *one Of* (enumerated class), *disjoint with*, *equivalent*, *subclass*
  - Boolean combinations of other classes: *union*, *complement*, *interSection*
- *Individuals*. An individual is an "instance" of a *class*, and they can be linked to *properties*.
- *Properties*. A *property* can be applied to *individuals*, and can have characteristics (i.e. "the properties of properties"), or can be applied restrictions:
  - characteristics: *inverse, transitive*, functional
  - restriction: *cardinality*, *range of values*

Several ontology languages are available (OWL, DAML+OIL, RDFS, WSMO …) and no standards have been set yet, however OWL seems to be well positioned to become the standard in the future, for its rich expressive power and his layered architecture perfectly fit for scalability. This is the reason why we will focus in this project on OWL and its upper-ontology for web services, OWL-S.

OWL-S is an OWL based web services ontology providing a mark-up language to describe properties and capabilities of services in a computer-understandable form, and is currently at the version 1.1. OWL-S has been developed purposely to enable automatic web discovery, automatic web invocation and automatic Web service composition and interoperation, which fits the aim of the project. We can notice that OWL-S is the successor of the outdated DAML-S. OWL-S Services are described by profiles, models, and groundings. Each service (instance of the general Service class) can present several (or no) profiles and these one or more profiles can optionally be described by at most 1 service model (subclass of ServiceModel) which has to be supported by 1 service grounding (instance of a subclass of ServiceGrounding). To summarize, the service profile is used to advertise services, to support requests and is to be used in the context of service discovery, while the ServiceModel and the ServiceGrounding give information on how to use the selected service, as shown in the figure below from the owl-s 1.1 white paper ([3]).

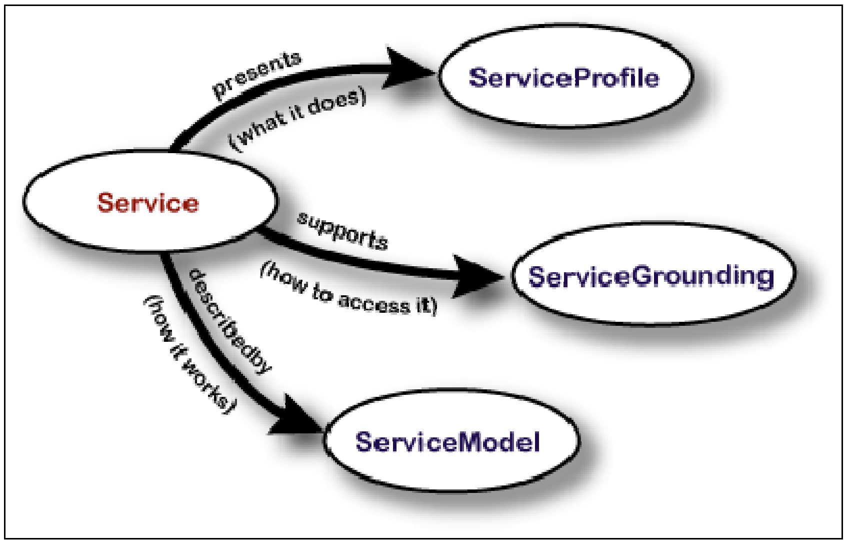


Figure 3-2 Top level of the service ontology

We will now detail the ServiceProfile, ServiceModel and ServiceGrounding.

- ServiceProfile:

The class ServiceProfile is meant to "advertise" the service, i.e. it presents "what the service does" with all necessary functional information: required input and generated output parameters of the service, its preconditions (e.g. being logged on the system), and its results (i.e. the transformation produced by the execution of the service, e.g. a booking ticket service has for result to effectively book the ticket). It is important to note that the ServiceProfile class can be used not only to describe and advertise services, but also to request a service, i.e. a requester can create a service profile to describe its need, so that this requested profile can be matched against advertised profiles. The profile is meant to present all the information needed to determine whether a service meets one's need or not. All the elements of the service profile, organized in *description*, *service functionalities* or *functional attributes*, are shown in Table 3-1.

Description	***Service Functionalities***	***Functional Attributes***
serviceName	*parameter*	*geographicRadius*
intendedPurpose	*input*	*degreeOfQuality*
textDescription	*output*	*serviceType*
role	*conditionaloutput*	*serviceCategory*
requestedBy	*precondition*	*serviceParameter*
providedBy	*accesscondition*	*communicationThru*
	effect	*qualityRating*
	domainResource	*qualityGuarantees*

Table 3-1 Composition of the OWL Service Profile

- ServiceModel

The service model tells "how the service works", that is, it describes in details all the processes chained while executing the service, how these potential processes are executed, and under which conditions they are executed. This can be compared to a BPEL composite service (see [12]). This service model can be useful in the case of selecting relevant services matching specific needs, as it can be used to perform a more in-depth analysis of whether the service meets a particular need. For instance, the matchmaking engine in [9] uses the ServiceModel to match a request with an advertisement.

- ServiceGrounding

A service grounding specifies the details of how to access a service. Typically a grounding may specify some well know communications protocol (e.g., RPC, HTTP-FORM, SOAP, Java remote calls …), and service-specific details such as port numbers used in contacting the service. This will not be of great interest for our work.

3.1.3. Reasoning

Given OWL documents, it should be possible to deduce additional information which could be used while matching services. The reasoning process essentially deduces new information from subsumption (subconcept and superconcept relationships between concepts of a given terminology, e.g. in our ontology in Figure 3-1, Vehicle *subsumes* Car, Vehicle *subsumes* SUV) but should also provide reasoning on individuals, that is, mainly checking the *consistency* of the knowledge base (e.g. determining, given an appropriate TBox corresponding to the Figure 3-1, that an individual A, if declared to be both a Car and a Bus, is a consistency error).

Reasoning engines usually work with standard algorithms for DL reasoning, called *tableau algorithms,* evolutions of tableau calculus for first order logic*.* Yet *s*ome other inference engines, as Jess (Section 3.2.1) may use other algorithms, such as The Rete algorithm. The Rete algorithm reasons over a graph, *the Rete*, where the nodes, with the exception of the root, represent patterns and paths from the root to the leaves represent left-hand sides of rules (conditional expression).

3.2. Reasoning

3.2.1. JESS approach

Jess (Java Expert System Shell) is a rule engine and scripting language which supports the development of rule-based systems. The data from RDF or RDF based document is transformed and stored as *facts* and the logic is defined as a collection of *rules*. Jess uses the Rete algorithm (see [20]) to process rules and infer new information, an efficient mechanism for solving the difficult many-to-many matching problem. These facts and rules are represented using the KIF (Knowledge Interchange Format) axiomatisation, shown in the following examples. One Jess rule would have the form "*(PropertyValue <predicate> <subject> <object>)*", which is enough to assert any OWL information since it is based on RDF, using triples for any constructs. For instance one Jess fact may be:

```
(MAIN::triple
    (predicate "http://www.w3.org/2000/01/rdf-schema#domain")
    (subject http://www.mindswap.org/2003/owl/geo/onto.owl#hasCoordinateSystem")
    (object ”http://www.mindswap.org/2003/owl/geo/onto2.owl#SpatialThing"))
```

Indicating that, in the owl document, the property *hasCoordinateSystem* is limited to the *domain* (as defined in the rdf-schema from the W3C specification) *SpatialThing.* And one Jess rule could look like:

```
(defrule subclassInstances
  (PropertyValue http://www.daml.org/2001/03/daml+oil#subClassOf ?child ?parent)
  (PropertyValue http://www.w3.org/1999/02/22-rdf-syntax-ns#type ?instance ?child)
=>
  (assert (PropertyValue http://www.w3.org/1999/02/22-rdf-syntax-
ns#type ?instance ?parent))
)
```

Stating that an instance of a subclass is an instance of the parent class.

The Jess underlying Rete Algorithm is a powerful mechanism improving the speed of forward-chained rule systems by limiting the effort required to re-compute the conflict set after a rule is fired. Its drawback is its high memory space requirements. It takes advantage of two empirical observations:

- *Temporal Redundancy*: The firing of a rule usually changes only a few facts, and only a few rules are affected by each of those changes.
- *Structural Similarity*: The same pattern often appears in the left-hand side of more than one rule.

However, it is important to note that Jess is not freely distributed and is submitted to license restrictions.

3.2.2. JENA approach

Jena is a Java framework for building Semantic Web applications. It provides a programmatic environment for RDF, RDFS and OWL, including a rule-based inference engine. It includes:

- A RDF API
- Reading and writing RDF in RDF/XML, N3 and N-Triples
- An OWL API
- In-memory and persistent storage
- RDQL – a query language for RDF

Apart from these features, widely used by an important number of applications, Jena (starting from Jena2) also provides a support for inference engines. The Jena2 inference subsystem is designed to allow a range of inference engines or reasoners to be plugged into Jena. Such engines are used to derive additional RDF assertions. The primary use of this mechanism is to support the use of languages such as RDFS and OWL which allow additional facts to be inferred from instance data and class descriptions. The default OWL reasoner included in Jena is rather limited and incomplete hence the need for a fuller reasoner to be plugged on Jena.

3.2.3. Comparing approaches

A bunch of OWL reasoners are available, all different by the underlying inference engine, the language they are based on, or by their completeness. These reasoners are, for the most known, FaCT, Racer, F-OWL, Pellet, OWLJessKB and many others. As seen in Section 3.1.3, our requirement for a reasoner is the ability to reason over OWL-DL documents, supporting subsumption inferencing and reasoning on individuals. This is basically done by almost all available reasoners, however, for our work is implemented in a Java environment, we should limit our choice to a reasoner efficiently pluggable in a Java environment. Basing our choice on this criterion may not seem straightforward, however our reasoning tasks required being not excessively complex and the available reasoners providing quite the same functionalities, it is yet sound to discriminate reasoners upon Java compatibility. We recorded three potential adapted reasoners for our work: the simple Jena OWL reasoner, Pellet and OWLJessKB. The first is an elementary but yet sufficient reasoner, the second is an elaborated OWL-DL reasoner plugged on Jena, and the latter is based on Jess. Our choice will thus essentially be made upon a comparison between Jess and Jena.

- Jena OWL Reasoner is a still undergoing development rule-based reasoner for OWL-Lite. Reasoning about classes is done indirectly by creating "temporary" instances, and if a "temporary" instance of a class A can be deduced as being member of another class B, then the reasoner deduces that class A is a subclass of class B. This approach is in contrast to more sophisticated Description Logic reasoners which work with class expressions and can be less efficient when handling instance data. The reasoner is thus most suited to applications involving primarily instance reasoning with relatively simple ontologies and least suited to applications involving large rich ontologies. Moreover, the reasoner is supposed to be sound but not complete.

- Pellet is an open-source Java based OWL DL reasoner which can be used in conjunction with either Jena or OWL API libraries. It is based on the standard tableaux algorithms developed for expressive Description Logics and supports all the OWL DL constructs. Pellet presents several useful features such as ontology analysis and repairing, data type reasoning or entailment.

- OWLJessKB is a description logic reasoner for OWL and is a successor to DAMLJessKB. The semantics of the language is implemented using Jess (see Section 3.2.1), and both Jena (see Section 3.2.2) and Jess are needed to run OWLJessKB. This reasoner can parse OWL documents, but is limited to OWL 1.0.

	FaCT	Racer	Jena	Pellet	OWLJessKb
Java	DIG interface	DIG interface	Implement ed in Java	Implemented in Java	Implemented in Java
OWL support	OWL-DL	OWL-DL	OWL-Lite	OWL-DL	OWL-DL
License	Free/Ope n-Source	License/ Commercial (*http://www.racer-systems.com/products/racerpro/index.phtml)*	Free/ Open-Source	Free/Open-Source	License *(http://herzberg.ca.sandia.gov/jess/)*
Limitations	No ABox support		Maladapted to large ontologies/ only supports RDF tuples	Performance slightly under FaCT or Racer	Limited to OWL 1.0
Advantages		Optimized/ Better performances		Suited for light weight applications (performance between Jena and FaCT/Racer)	Handle arbitrary tuples/ supports closed-world assumption

Table 3-2 OWL Reasoner comparison

We will retain only two reasoners for our work: Jena and Pellet, suited for our basic reasoning needs, and offering good enough performances.

Chapter 4. Services discovery approaches

In this chapter, we will focus on solutions addressing similar concerns as the one of our work. This includes the global framework, where providers register their advertised services, and where requesters send descriptions of desired services to a matchmaker. One of the most important features, not to say the crucial one, is the engine responsible of the matchmaking, i.e. finding advertised services corresponding to a request. In the first part of the chapter we will concentrate only on existing OWL services matchmaking methods, and we will then see services discovery frameworks and how matchmaking engines are included in those frameworks. Section 4.2.2 provides a description of the framework this project is extending.

4.1. Matchmaking

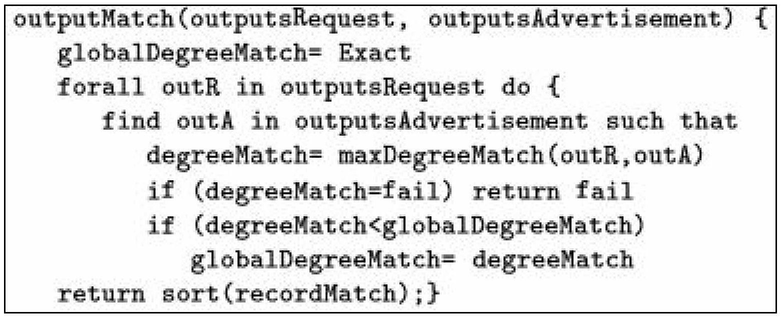
The matchmaking is the core part of any discovery system as it is the component in charge of proposing relevant services to a user requesting a specific service. As mentioned in Chapter 3, the most relevant language used to describe web or grid services appears to be the potential standard OWL-S. An OWL-S description comprises three parts: the profile, the model and the grounding, where only the first two parts describe the service in itself. Thus, only the profile and the model can be exploited to calculate to what degree a pair of services matches. In the following two Sections, we will see how the profile and the model can be used for matchmaking.

4.1.1. Matching service profiles

The idea beneath the matching methods (in [10] and [7]), is that two services do not necessarily need to be exactly equal to match; they only need to be "sufficiently" similar. It is obvious that a service provider and a service requester do not have any prior agreement and can have very different objectives. For instance, a provider can advertise vehicle selling services, whereas a requester can be looking for a service selling car, both services are aimed at different objectives, but they are still similar enough to be considered as matching. In order to allow sufficiently similar matches, the matching process has to be flexible, i.e. it should recognize a degree of similarity, and it should be to the user to decide the minimum degree of similarity required in a match. Yet, the inherent problem with flexible matches is the risk that providers would advertise voluntarily generic services, so the matching engine systematically returns its services (problem which can occur on the requester side as well), thus leading to a great number of "false positive" (services wrongfully returned as matching). Consequently, an important task is to encourage providers and requesters to describe services honestly, so to reduce false positives and false negatives. It is also noteworthy to remark that the more (resp. the less) flexible a match is, the more (resp. the less) false positives and the less (resp. the more) false negatives will be returned. The key to sufficiently similar matching is then to recognize semantic matches despite syntactic differences.

- A first approach:

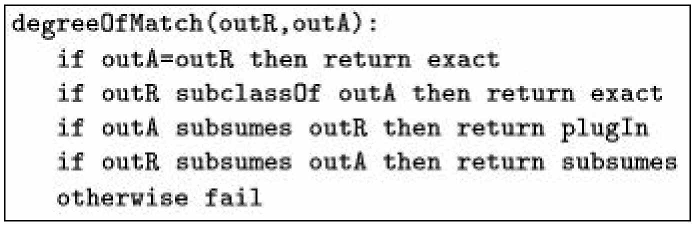
The algorithm taken from [10] and shown in Figure 4-1 consists of matching all the outputs of a request against the outputs of an advertisement, and all the inputs of the advertisements against the inputs of the request.



```
outputMatch(outputsRequest, outputsAdvertisement) {
   globalDegreeMatch= Exact
   forall outR in outputsRequest do {
      find outA in outputsAdvertisement such that
         degreeMatch= maxDegreeMatch(outR,outA)
         if (degreeMatch=fail) return fail
         if (degreeMatch<globalDegreeMatch)
            globalDegreeMatch= degreeMatch
   return sort(recordMatch);}
```

Figure 4-1 Algorithm for output matching

Then, according to subsumption relationships between inputs or outputs, a degree of match is determined (*exact*, *plug in*, *subsumes* or *fail*). Figure 4-2 shows how the degree of a match is determined:



```
degreeOfMatch(outR,outA):
   if outA=outR then return exact
   if outR subclassOf outA then return exact
   if outA subsumes outR then return plugIn
   if outR subsumes outA then return subsumes
   otherwise fail
```

Figure 4-2 Rules for the degree of match assignment

- An extension of the algorithm

Another extended profile matching has been presented in [7] and [8] which uses the classification of elements available since the DAML language. The idea is similar to the previous matching engine described above, but deepens the concept of flexible matches and degree assignment. Indeed, the algorithm distinguishes up to 9 different degrees for the matching of parameters. These 9 different degrees are justified by the classification of parameters and service profiles, taken into consideration. The signification of each degree is shown in Table 4-1. DAML-S service profiles are defined as subclass of the *Profile* class, but can also be indirect subclasses of *Profile*, this way it is possible to build a service hierarchy (see the explanatory remarks about profile-base class hierarchies in [13]) and relationships between two profiles can be found with reasoning on subsumption. This feature, even if not really used in practice, would provide an interesting "yellow-page" style service categorization. Moreover, IOPEs (Inputs-Outputs-Preconditions-Effects) can also be classified the same way, by defining an IOPE parameter as a subproperty of another IOPE parameter. Thanks to these classifications, a distance between two profiles or two parameters (parameter here stands for the meta-information and not the value of the parameter itself) can be computed.

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

Table 4-1 Rankings for the matching of two parameters

The property-match result is the result obtained from the "category match" of the properties of the IOPEs. Then, for each possible property-match result, 3 type-match results are possible (the type-matching is the basic matching evoked in the first approach). If a property-match (resp. a type-match fails), we do not need to consider the type-match (resp. property-match) result.

- the ATLAS matchmaker

In [14], a different matchmaker, based on the DAML-S services profiles, is presented. The novelty brought by this method is simply the consideration of the functional attributes (*geographicRadius*, *degreeOfQuality* …) during the matching process, the service functionalities matching remains quite similar as [10]. The formula used to match inputs is below:

$$match(R_I, A_I) \Leftarrow (\forall j, \exists i: (i \in R_I) \wedge (j \in A_I) \wedge subs(i, j)) \wedge R_I = \varnothing$$

And the matching of outputs:

$$match(R_O, A_O) \Leftarrow \forall i, \exists j: (i \in R_O) \wedge (j \in A_O) \wedge subs(j, i)$$

Where subs(i, j) is true when i subsumes j.

4.1.2. Matching service models

The authors of the method in [9] present their algorithm as an extension to profile matchmaking as they take into account the detailed process description of services, the service model. It is their belief that this algorithm, based on richer descriptions, should lead to more accurate matches. To understand how the algorithm works, let us first note that the service model describe the process executed when calling the service. This process can be decomposed into other processes, which can in turn be decomposed themselves, and so on … A process which cannot be decomposed is called atomic and a process composed of other processes composite. A composite process can be of several types: *Split*, *Sequence*, *Unordered*, *Split+Join*, *Choice*, *If-then-else*, *Iterate* and *Repeat-Until*. The service model can eventually be considered as a tree representing the global process, having the atomic constituent processes as leafs and each sub-tree being a composite process. Thus, the proposed algorithm works recursively over the tree of the service model, and each node is matched (with a particular algorithm depending on the type of the composite process at the node: split, sequence …). The algorithm is started by simply matching the root of the tree, which will call recursively the matching algorithm over its nodes and sub-nodes. Below is shown the algorithm used to match outputs of either a split or a sequence composite process (Figure 4-3). *I* is the list of inputs to be matched, *O* is the list of outputs to be matched, *N* is the node being matched.

```
matchOutputs(List I, List O, split-seq-Node N)
if O is empty then
  return true
end if
$o_1 \leftarrow$ head(O)
for all $k \in$ N.children do
  k.matchSet $\leftarrow$ k.matchSet $\cup \{o_1\}$
  if matchOutputs(I, k.matchSet, k) then
    if matchOutputs(I, tail (O), N) then
      return true
    end if
  end if
  k.matchSet $\leftarrow$ k.matchSet - $o_1$
end for
return false
```

Figure 4-3 Algorithm matching outputs of either a split or a sequence node

A split or a sequence node denotes a list of processes to be done concurrently (split) or in order (sequence), and which finishes when all the children processes are terminated. Hence the algorithm above which calls the match a success if all the desired outputs can be satisfied by all the children collectively. By opposition, matching a choice node is done by finding at least one node satisfying the desired outputs.

We believe it is true that this kind of match can provide a more precise and more accurate match between two services, yet we should not forget that the service model is not primarily provided to express requirements for finding matches with advertisements, thus this does not seem to be the most relevant way to solve the matching problem.

4.1.3. Comparison

In the following table, n stands for the height of process model tree.

	Profile Matching		Model Matching	Others (Atlas …)
	4 degrees	9 degrees		
Complexity	O(1)	O(1)	$O(2^n)$	O(1)
Accuracy	Minimal sufficiency	Better than 4 degrees matching	High	

Speed	Fast	Fast	Time consuming $O(2^n)$ algorithm	Similar as profile matchers
Space	Knowledge storage (low)	Knowledge storage (low)	Knowledge storage + algorithm stack	Similar as profile matchers

Table 4-2 OWL services matcher comparison

From the two profile matchers, we should prefer the 9 degrees profile matcher which offers a better accuracy with the same performance in time or space, the only difference observed being the slightly higher complexity due to the higher number of reasoning tasks to perform. For a higher accuracy, the service model should be picked, which would imply neglecting speed and memory considerations. As a matter of fact, this matcher does the same reasoning tasks as the profile matchers, but performs them a greater number of times, and furthermore adds a non-negligible algorithm complexity. Profile matchers offer a tradeoff between speed and accuracy, model matcher offer a high accuracy at any costs, and other matchers (ATLAS or others) allow unsubstantial additional features easily pluggable into one of the previous matchers.

4.2. Frameworks

We saw above the subpart of matching a pair of services, we are now about to review some frameworks which integrate these matchmaking techniques, along with other ways of matching services. But before that, let us introduce important concepts of agents' structures.

4.2.1. Introduction to Agents & Multi-Agents Systems

Agents systems provide the scalability and the flexibility needed in distributed environments and distributed agents offer an interesting alternative to centralized repositories as the number of web services is dramatically increasing each day. Agents systems can be useful to leverage the problem of location and implementation of services changing frequently, by removing the centralized server necessary to check regularly services changes. Using MAS (Multi-Agent Systems), we make service advertisers autonomous, so that themselves can inform other entities of any change on the service(s) they are providing. Thus, providers and requesters can interoperate asynchronously.

Let us present shortly the principle of agents. Agent oriented programming is a paradigm on top of the object oriented paradigm, that is, agents are an evolution of objects. Synthetically, agents are entities acting (taking decisions independently) on behalf of a user, in order to achieve specific goals. Their characteristics are:

- Ability to act proactively, i.e. ability to take initiatives
- Reactivity (to events occurring in their environment, to contacts with other agents …)
- Social abilities.

We call Multi-Agents Systems (MAS) systems in which several (heterogeneous) agents are connected altogether. MAS are commonly used to solve problems beyond single agents capabilities or to model market places where each agent act in a selfinterested manner toward their own and divergent motivations. Yet our setting is a situation of mutually beneficial cooperation where agents have different goals, but where no opposition or conflicts of interest can arise among them (providers' sole motivation is advertising services, requesters' motivation is to find suitable services and each one need to cooperate so that they both can accomplish their task). Our MAS infrastructure will be used to make our system more flexible and more scalable, with greater conception clarity. We will now see the MAS system this work is extending.

4.2.2. Agent-based system for Grid services provision and selection

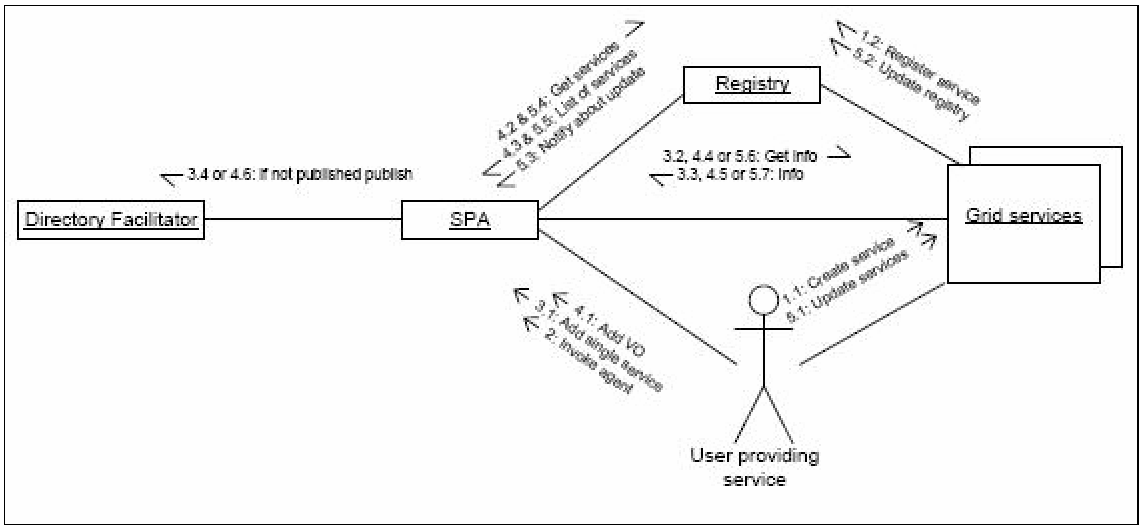


Figure 4-4 Collaboration diagram of the service providing part of the system

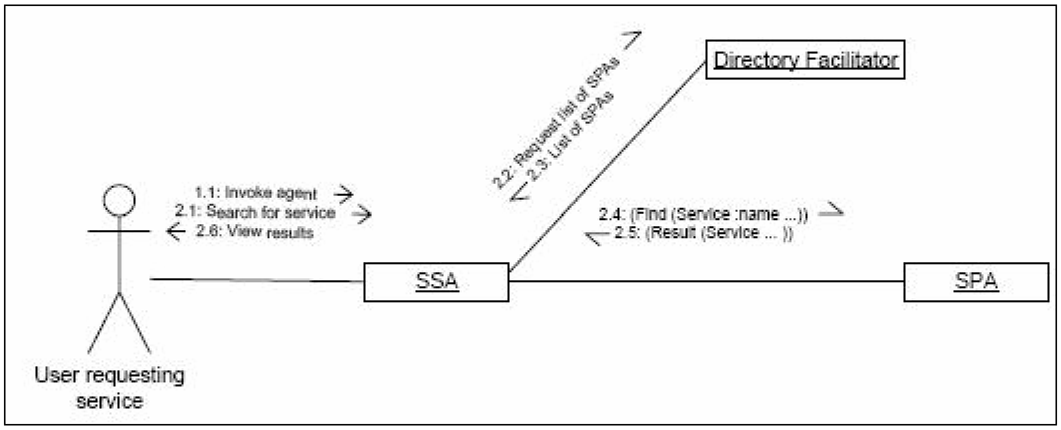


Figure 4-5 Collaboration diagram of the service selecting part of the system

This system is the base system our work is intended to extend. It provides a framework service providers and requesters can directly use to publish or find services. This platform has been designed to be used exclusively for grid services, for it uses OGSI's Service Data Elements (see Section 2.2.2). In this system, no semantic information is used to describe services, and services can only be retrieved thanks to the OWL-S *ServiceCategory* element, used for classification. Services (requested or advertised) are described in WSDL, with the OWL-S *ServiceCategory* attributes (CategoryName, Code, Taxonomy and Value) enclosed in the SDEs. The interest of this platform is not limited to the matchmaking of services, as it also gives users an ad-hoc method to publish services, either by adding single services or by adding a *Globus* Virtual Organization Registry ([15]). Basically, the system works as follow. A user has first to instantiate a Service Selection Agent (SSA) to perform its search. The SSA then fetches all the Agents providing services in the system, and sends the service request to all of these services. The Agents interfacing the users providing services (Service Providing Agents, or SPA) matches the request with the one or more services they store, and in case of success, returns the list of matching services to the SSA originating the request. Once an SSA has emitted a request to several SPAs, it only waits until a certain time for results to come. Below are the collaboration diagrams showing the global setting and interactions of the system for the provision and selection part.

4.2.3. UDDI-based discovery systems

Some methods have been developed, consequently to the observations that UDDI does not allow browsing according to capabilities and that it only provides limited search possibilities (see Section 2.1.2), in order to bring new functionalities to it. [16] and [10] propose a framework coupled with UDDI. This framework remains compliant with the current UDDI registries, so that searches can be done through regular registries (keyword searches) or through the new augmented registry, storing semantics information. Below is the sample architecture used by [10] in addition to the DAML-S matchmaker presented above in Section 4.1.1.

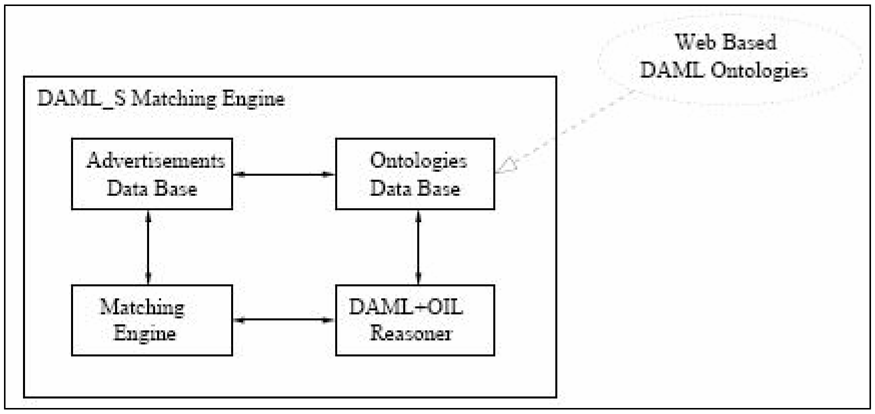


Figure 4-6 Architecture of the DAML-S/UDDI matchmaker in [10]

The system works as follow. Upon receiving a request, the Matching Engine component selects the advertisements from the *AdvertisementDB* that are relevant for the current request. Then it uses the *DAML+OIL Reasoner* to compute the level of match. In turn the *DAML+OIL Reasoner* uses the *OntologyDB* as data to use to compute the matching process. The *AdvertisementDB* also takes advantage of the *OntologiesDB* to index advertisements for fast retrieval at matching time. The *AdvertisementDB* can be likened to the *Directory Facilitator* (DF) in the previously described work ([1]), although this latter *DF* simply always returns *all* the services (more accurately, all the agents providing services). In the same way, the *Matching Engine* can be likened to the *SPAs,* as these latter match requests with their services even though they do not include any semantic reasoner or any ontology data bases. We can see that our ambition to extend the system in [1] will imply endowing *SPAs* of semantic reasoning (along with an appropriate ontology storage), and improving the *DF* to make it work similarly as the *AdvertisementDB* of the system just described and thus return only relevant *SPAs*.

4.2.4. Others

- RETSINA/LARKS

RETSINA is a multiagent infrastructure that performs goal-directed information retrieval, information integration and planning tasks. The motivation for this platform is a bit different from ours but still proposes an interesting matchmaking engine, composed of several filters:

- *Context matching*: for each pair of word of the slot context in the associated language LARKS used to describe the services, a word distance is computed.
- *Profile comparison*: term-frequency inverse document frequency weighting (TF-IDF), technique from the information retrieval area, is used to calculate the similarity of two profiles (based on frequency and relevance of words in the document).
- *Similarity matching*: computation of distance values for input/output pairs and for input/output constraint. Used to refine the previous filter (e.g. recognizes that {computer, book} has a closer distance than {computer, notebook}, which is not seen by the previous filter).
- *Signature matching*: matching pretty similar to other techniques evoked in the previous Sections. Checks if the input/output matches, based on semantics.
- *Constraint matching*: similar as signature matching, but applied to input and output constraints.

The filters provided are quite powerful but not portable in OWL-S, it would yet be interesting to find equivalent filters for an OWL-S matchmaker, as it may help filtering the set of services to match, i.e. to improve the Directory Facilitator as evoked in Section 4.2.3 (selecting candidate services for a match instead of selecting *all* services for a match).

- COINS, EcoCBL and others:

Other approaches for matchmaking exist but are mainly based on their own capability description language, due to their concern being only to describe capabilities of agents, instead of specifically describing services. However, these methods have the same concern as ours: matchmaking capabilities. We can quote COINS in [17], matching capabilities by calculating the distance of the words in the capability description thanks to TF-IDF (equivalent to the RETSINA context matching), ecoCBL in [18] or JAT-CDL in [19]. Like we said in Chapter 3, we will focus on OWL-S for the reasons already evoked, and this will lead us to discard such methods and use methods described in the first paragraph of this chapter instead.

Chapter 5. Proposed Solution – Design

In this chapter we will approach the overall design of our proposed solution. We will start by discovering the needs of the system along with how it could be used, and then we will model the application with UML diagrams, finally the non-trivial algorithms used will be discussed. This chapter will tackle functional specifications and not technical specifications, which will be discussed in the next chapter.

5.1. Preliminary Considerations

5.1.1. Terminology

Here are presented the most recurring terms used in this thesis, along with their signification.

Term	Signification
Service Provision Agent (SPA)	An agent handling the service provider's part in a negotiation of services.
Service Selection Agent (SSA)	An agent handling the service requestor's part in a negotiation of services.
Directory Facilitator (DF)	A predefined agent holding a directory where other agents can publish themselves and search for others.
Extended Directory Facilitator (EDF)	An agent meant to replace a Directory Facilitator Agent (DF), by providing some additional functionalities.

Table 5-1 Glossary of recurrent terms

5.1.2. Scenario of system use

There will be two different types of users in the system; those who provide services and the ones requesting them.

The scenario we are presenting involves music and media libraries on the internet. Suppose a music-selling company wants to publish its services on the web to allow users to buy music online. Providing an artist name and a track name, the service would return the MP3 corresponding to the search criteria. This would constitute an interesting low-cost way of selling music, all automatized. The company could then register to the platform along with their others media-accessing services, so that anyone can find and access the services. The company could also register its services to UDDI registries but its limitations does not make it the best way to make the services known to the open public. Moreover, some other limitations evoked a little bit later will confirm the choice of not using UDDI. Suppose now a user wishes to use the platform to locate services of that type. If he knows exactly his desired songs with the performer, he will not have any trouble to get his music. However, it could happen that the user does not know exactly the name of his wanted track. In that case, the first reaction to adopt is to locate another service on the platform which has for output the same concept the music-selling service had in input ("track name"). Such a search should provide results containing media-library services giving track names provided the name of an artist and an album name (plus optionally a track number). Those other media-library services could provide lots of useful services to be combined with the music-selling services, and could substantially facilitate the music search. For instance they could help finding artists given a genre, or could help to inform about tracks (see Figure 5-1 for examples). These media-library/music-selling services combined together would form sorts of composite services with the same result: finding mp3s from the music-selling service, but with different inputs (genre, or artist and track, or year and artist and track number …). We can see that the company selling music makes a good choice when using the platform instead of UDDI, indeed, it naturally enhances users' ergonomics, who do not have to restrict their music search on a track name and an artist name. Such things are not possible with UDDI, as media-library services are searched based on input or output parameters, according to what the music-selling service expects in input and to what information the user wishes to provide in input.

Below is illustrated the scenario, the music-selling service provided by the company is the service on the right, and all other media-library services are other costless services, provided by other entities to the platform. The dotted line represents some of the possible combination between services.

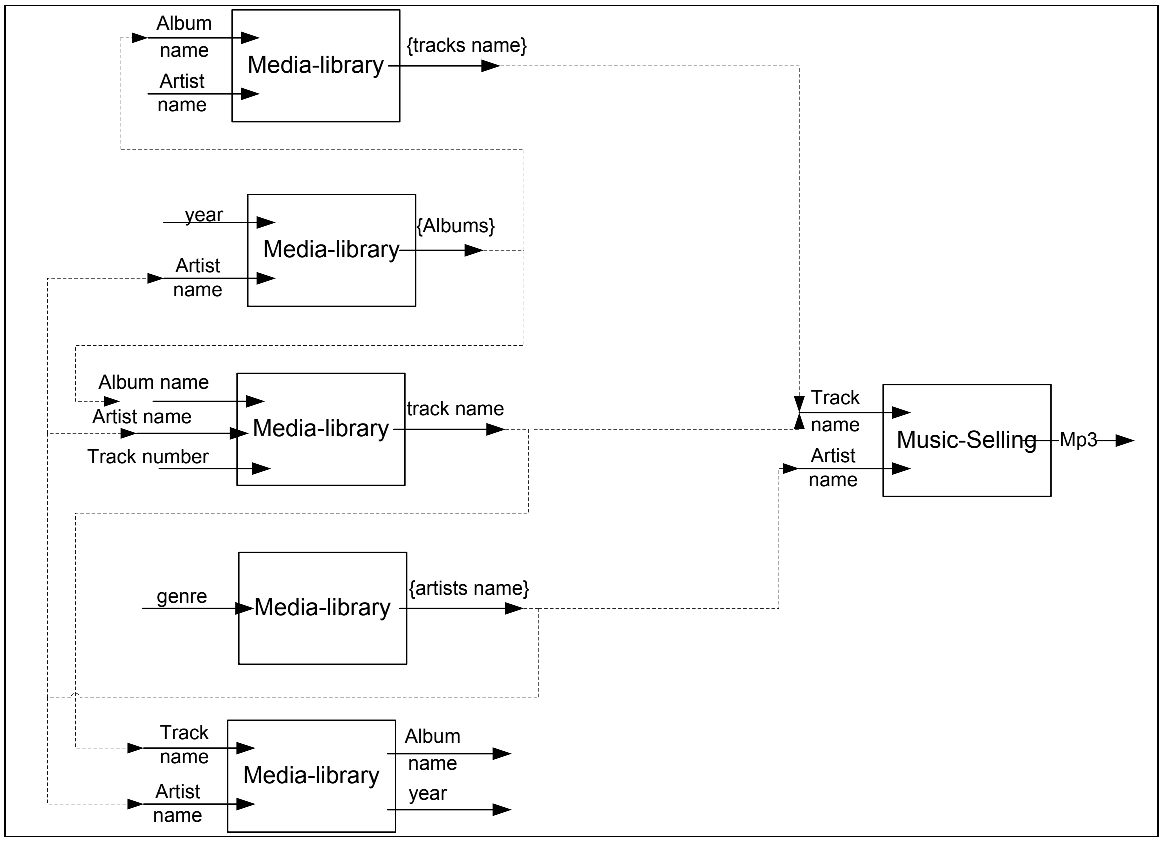


Figure 5-1 Illustration of the scenario of system use

5.1.3. System needs

The system will be articulated around two main objectives: it should first provide a way for services providers to register their services, so that users on the other side can have access to them, and it should then provide a way for users to find relevant services, according to their specific needs. We should also bear in mind that users can be humans as well as computers. In order to realize these two objectives, a third important feature has to be set: the matchmaking engine (coupled with a reasoning engine), which will be in charge of finding correspondences between requests and advertisements. Finally, as this system is to be in a distributed environment by nature, there should be a mechanism enabling actors of the systems to locate each others. Apart from upgrading the previous version of the system by adding semantics, several other issues should have to be tackled.

We should first consider some security issues of the system, for instance, there should be possibilities for a user to emit requests not giving away too much information to SPAs (privacy issue), and the requester should then be able to select its services from a set of services given by the different SPAs. This way, the role of the SPAs would not be to *match* its services with a request, but to filter its potentially matching services from a request carrying limited information. This mode will be called the "secure mode", by opposition to the regular mode, where the SPAs match a request containing all necessary information.

Then, efficiency issues could be considered, by evolving the DF agent. In the previous version, this agent's task is limited to returning *all* available SPAs, our system could provide an extended DF (EDF) which would return only a restricted list of SPAs, depending on the request of the user (in that case, the user would provide limited information, as for the secured mode, to the EDF first). We will call that mode the "filter SPAs" mode.

To sum up, we can notice the following major upgrades from the previous system:

- Semantics to be added
- Matchmaking to be enhanced
- Several working modes, which can be combined or both deactivated:
    - Secure mode
    - Filter SPAs mode

5.2. Model of the application

5.2.1. Use Cases

The Use Case for the application have not changed since last version, the system is still used the same way. Here is the global Use Case as a reminder (Figure 5-2):

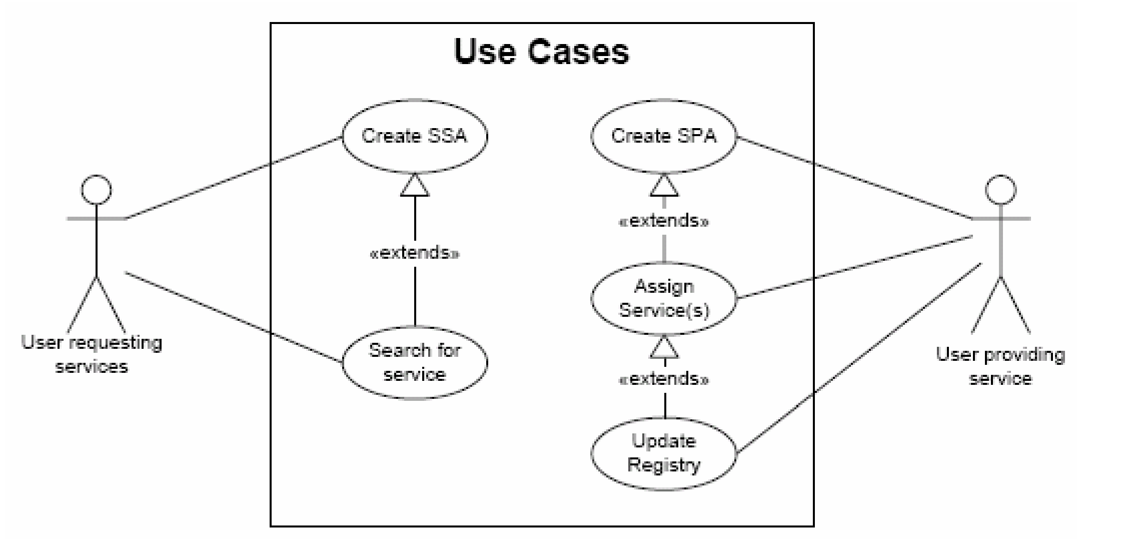


Figure 5-2 Use Cases of the system (from [1])

As the use of the system does not deeply change from last version, we will only focus here on important changes, for deeper understanding we invite the reader to refer to Section 3.3 in [1].

- *Type of Services concerned by the system*. The previous system was resolutely using Grid Services, whereas this version should be more flexible. Besides, the concept of Grid service, as said in Section 2.2.2, is becoming obsolete, and both web services and grid services are now to be replaced by the WS-RF. However, in order to select a service, we only need to consider the description part of an OWL Service (mostly the profile and secondarily the model, refer to Section 3.1.2). The fact that a service uses a wsdl, gwsdl or endpoint references in SOAP messages only affects the grounding part of an OWL Service, part which is note relevant in the selection process. As a consequence, regular Web-Services, Grid-Services, or WS-RF services can be used by the system. Yet, The Virtual Organizations provided by the Globus Toolkit 3 ([15]) used in the previous system cannot be used anymore, but have to be replaced by the Indexing Services, provided by the Globus Toolkit 4, implementing the WS-RF.
- *Description of services*. OWL Services were used by the previous system, but without exploiting the capabilities of the language. Services were described by GWSDL documents along with SDEs (see Section 2.2.2), which were wrapped and translated into an OWL Service (more precisely, the SDEs were transferred to the service category tag of the OWL Service). This system should dispense with WSDL or SDEs descriptions, and use OWL descriptions directly to express the requests and to describe advertised services.

5.2.2. Interaction between parts (sequence diagrams)

This Section will deal with the internal running processes of the application, i.e. with the interaction between agents. Collaboration diagrams from the previous system, presented in Section 4.2.2, are still up to date.

- SPA

The SPAs should provide the following functionalities:
- Adding & Removing service(s)
- Registering & Deregistering to a DF or to an EDF

Below (Figure 5-3) is presented the UML sequence diagram for the service adding action (which includes the registering functionality).

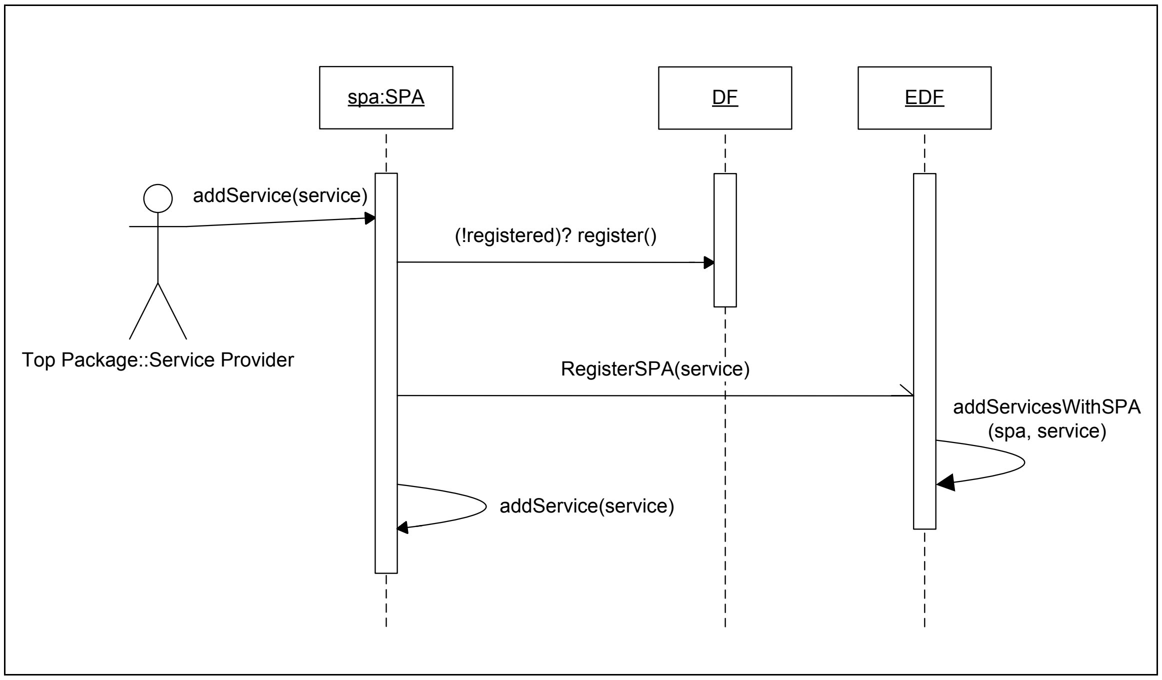

spa:SPA
DF
EDF
addService(service)
(!registered)? register()
Top Package::Service Provider
RegisterSPA(service)
addServicesWithSPA (spa, service)
addService(service)

Figure 5-3 adding a service to an SPA

As the EDF agent is only optionally started, the action "RegisterSPA(service)" is not done systematically. The EDF, by opposition to the regular DF, also stores, apart from agent's references, references (i.e. URLs) of the services shared by each SPAs.
The diagram for the service removing action is not presented but is the trivial symmetric of the previous diagram.

- SSA

The SSA should provide the following functionalities:
- Finding SPAs
- Finding "relevant" SPAs ("Filter SPAs" mode)
- Finding services matching a request
- Matching services with a request ("Secure" mode)

A user wishing to search services has to first instantiate an SSA, let ssa be that instance, and let spa_1 … spa_n be instances of SPAs sharing services. The regular search for services works as in the sequence diagram below:

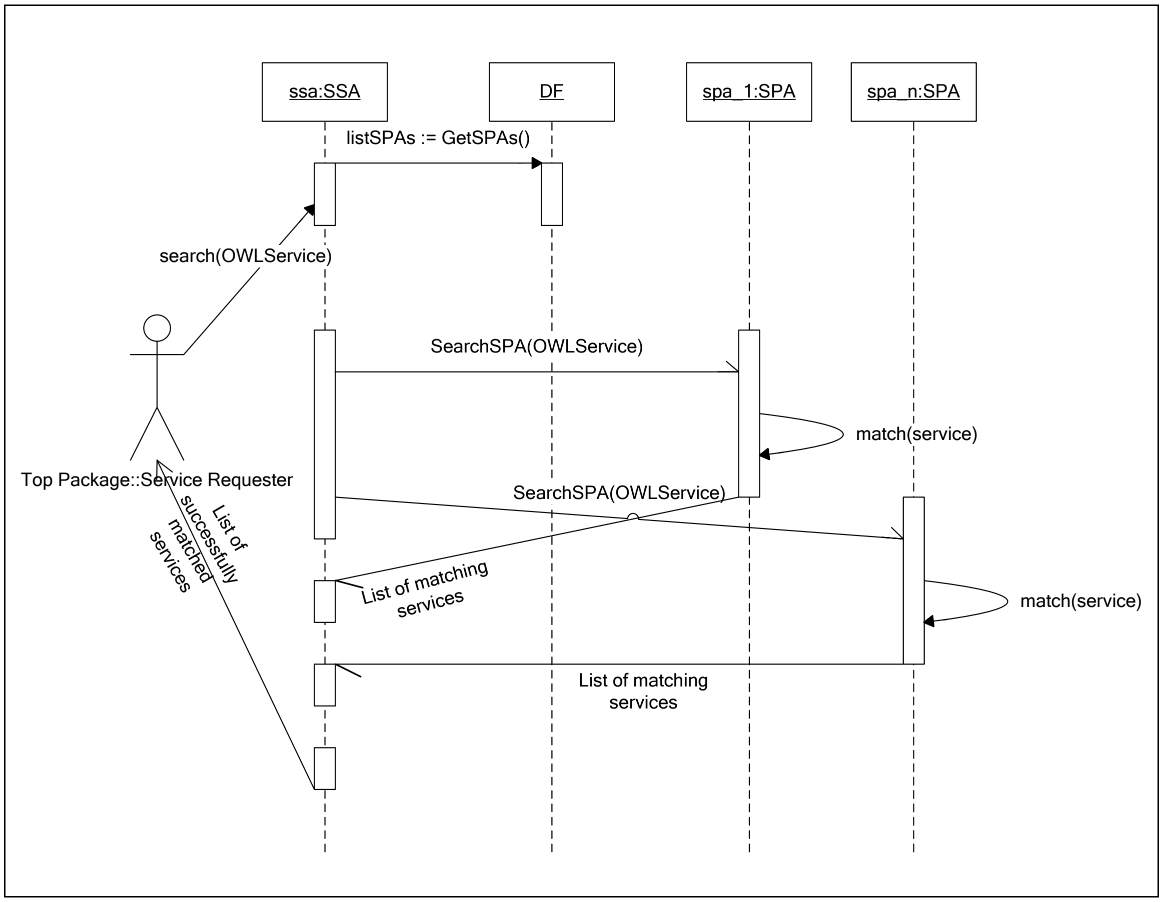


Figure 5-4 Searching services ("regular" mode)

The first call to the DF (*GetSPAs()*) allows the SSA to obtain a list of all available SPAs, so that each of them can be probed (*SearchSPA(…)*) for services matching the request (OWLService). The SSA eventually asynchronously receives replies from probed SPAs and gathers all returned services for further treatments (limited to simple display in our project).

Small differences can be observed when using the "secure" mode, as the following sequence diagram shows:

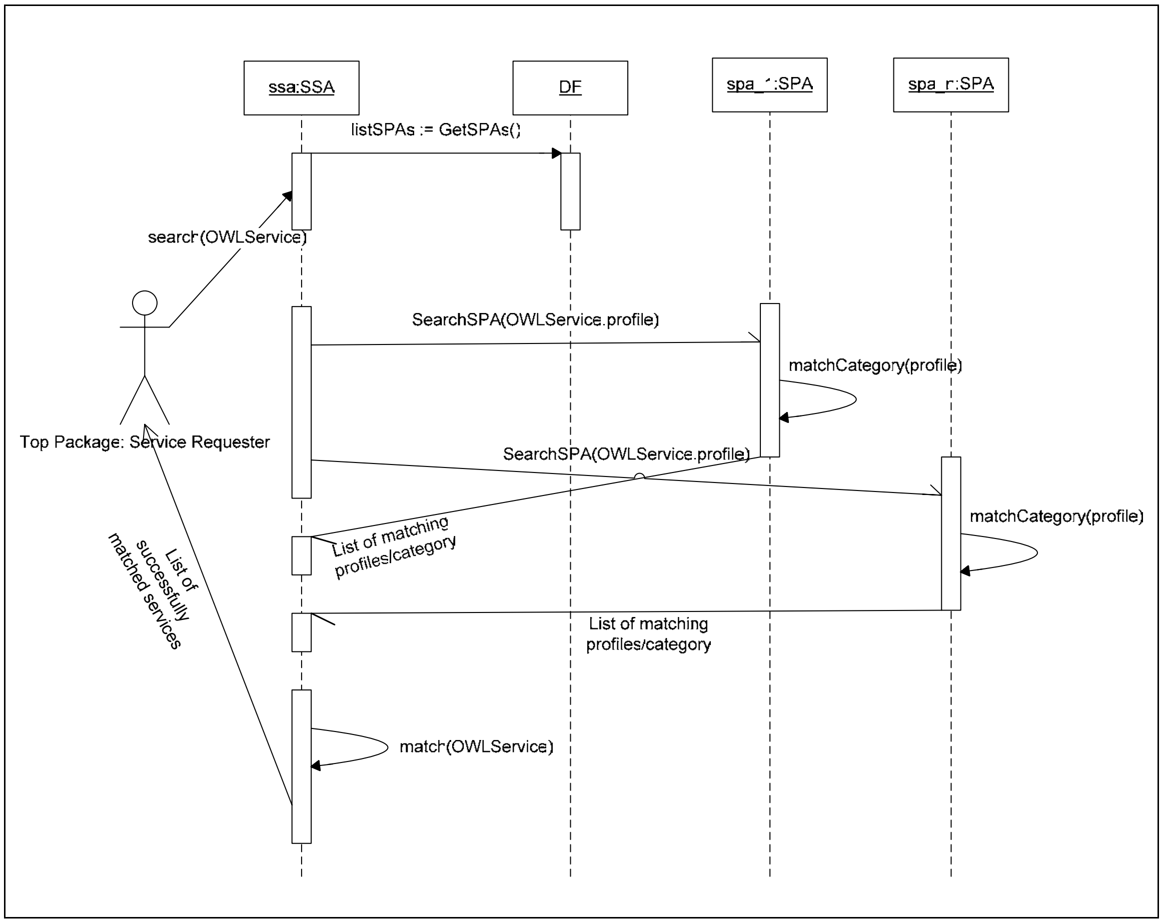

ssa:SSA
DF
spa_1:SPA
spa_n:SPA
listSPAs := GetSPAs()
search(OWLService)
Top Package: Service Requester
SearchSPA(OWLService.profile)
matchCategory(profile)
SearchSPA(OWLService.profile)
matchCategory(profile)
List of matching profiles/category
List of matching profiles/category
List of successfully matched services
match(OWLService)

Figure 5-5 Searching services ("secure" mode)

Instead of sending the full OWL Service to the SPAs, only the profile of the request is sent. The class of the profile and its category are matched (*matchCategory(…)*) against the same entities in the services advertised by each SPA. After sending all the requests to the SPAs, the SSAs then wait to collect all the replies. Each reply from each SPA ("List of matching profiles/category") contains a set of services *potentially* matching the user's request. To determine if those services effectively match the request, the SSA has to match each of them against its request (*match(OWLService)*) (the job done by the SPAs on regular mode is here done by the SSA).

The "Filter SPAs" mode varies only for the first part of the negotiation, as the list of SPAs to contact is obtained from the EDF instead of the DF:

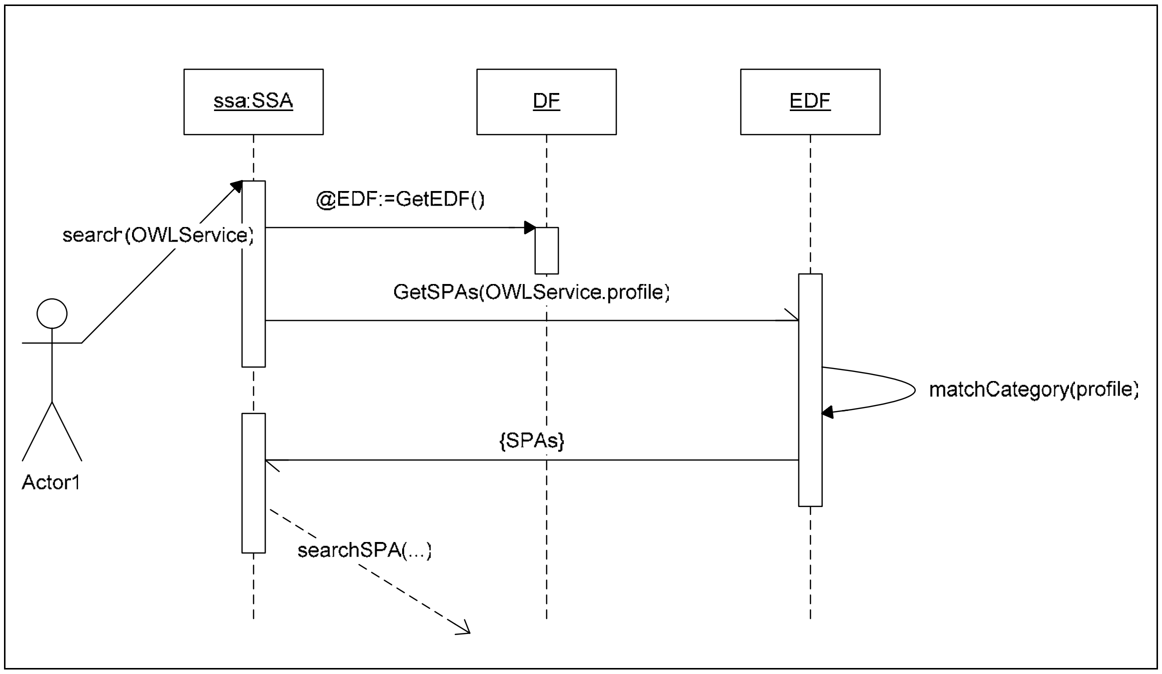


Figure 5-6 Searching services (“Filter SPAs” mode)

The DF still has to be used to locate the EDF (*GetEDF()*). Then, in order to find relevant SPAs (i.e. SPAs sharing services potentially matching the user’s request), the profile of the request is sent to the EDF (*GetSPAs(…)*), which in turn matches the class of the profile and its category against all the advertised services (*matchCategory(…)*, rigorously identical to the action performed by SPAs on “secure” mode). A list of SPAs to contact is then returned to the SSA and the services search can then begin as usual, in either “regular” (see Figure 5-4) or “secure” mode (Figure 5-5)

5.3. Algorithms

5.3.1. Profile matching

The algorithm used is the one described in Section 4.1.1 and is used with its extension (9 degrees profile matcher). The global algorithm, as described in Figure 5-7, does not differ from the one of the previous system, except the fact that it can return more than one service. It successively matches the inputs, the outputs and the class of the profiles.

```
List match(requestedService, providedServices) {
        List matching_Services := {}

        for each providedService in providedServices do
                int currScore := 0

                currScore :+= weightInput *
                                matchInput(requestedService, providedService)
                currScore :+= weightOutput *
                                matchOutput(requestedService, providedService)
                currScore :+= weightProfile *
                                matchProfile(requestedService, providedService)

                if currScore > MIN_SCORE then
                        matching_Services := matching_Services . providedService
                end if
        end for

        return matching_Services
}
```

Figure 5-7 Service matching algorithm

As mentioned in Section 4.1.1, the *matchProfile*(…) function only performs a concept match on the type of the profile (most of the time, profiles are instances (are hence of the type) of the default base class Profile, contained in the ontology **http://www.daml.org/services/owl-s/1.1/Profile.owl** and are thus considered as "unclassified").

To understand the following algorithms and the concept match or property match, let us focus on the following considerations. The concept match will match the *type* of the parameters, by opposition to the property match which matches the *class* of the parameters. Let us have a look at the following example:

```
- <process:Input rdf:ID="InputLanguage">
    <process:parameterType>http://www.mindswap.org/2004/owl-
      s/1.1/BabelFishTranslator.owl#SupportedLanguage</process:parameterType>
    <rdfs:label>Input Language</rdfs:label>
  </process:Input>
```

Figure 5-8 Sample OWL-S input parameter

In that example, the *type* of the parameter (i.e. that will be used in the concept match) is "*SupportedLanguage*", present in the ontology http://www.mindswap.org/2004/owl-s/1.1/BabelFishTranslator.owl. The class, or property, of the input (i.e. that will be used in the property match) is "Input" (the input is thus said to be "unclassified"). The concept match, as seen in the first part of the algorithm in Figure 5-10, is hence a process which will try to find relationships between two concepts (equivalence, subsumption, invert subsumption, or disjoint-ness). The same way, the property match, as seen in the second part of the algorithm in Figure 5-10, is a process which will try to find relationships between two properties (unclassified, subproperty, equivalence, or disjoint-ness).

The final score for two parameters, depending on both the concept match of the parameters' types and the property match of the classes of the parameters (function *assignDegree(…)* in Figure 5-10), is an integer comprised between 0 and 9; the signification of each rank is given in Table 4-1.

The rather trivial algorithm used to match inputs and outputs has been described in [10], and the outputs matching algorithm is shown in Figure 5-9.

```
int matchOutput(Profile request, Profile advertisement) {
        List outputsReq := request.getOutputs()
        List outputsAdv := advertisement.getOutputs()
        int finalScore, scoreMax, score

        finalScore := 9
        for each outputReq in outputsReq do
                scoreMax := 0
                for each outputAdv in outputsAdv do
                        score := scoreMatch(outputAdv, outputReq);
                        if score > scoreMax then
                                scoreMax := score
                        end if
                end for

                if scoreMax < finalScore then
                        finalScore := scoreMax
                end if
        end for

        return finalScore
}
```

Figure 5-9 Outputs matching algorithm

The above algorithm computes what can be stated as "the worst of the best scores" obtained. Indeed, for each output of the request, it finds the best matching output in the advertisement (finds "the best score"), and the final result of the function is the smallest score obtained in all the best matching outputs ("worst of the best score"). It can be summed up by the formula:

$$\min(\max(scoreMatch(outputAdv, outputreq) \;|\; outputAdv \in outputsAdv.getOutputs) \;|\; outputreq \in outputsreq.getOutputs)$$

The algorithm used to match inputs is quite similar to the above algorithm, but instead of finding a matching output in the advertisement for each output of the request, an input in the request is found for each input of the advertisement (the outer loop iterates over the parameters of the advertisement and the inner one over the parameters of the request), and the call to *scoreMatch* is done by *scoreMatch(inputReq, inputAdv)*.

The match degree assignment (*scoreMatch(…)*) as described in [8] for the 9-degree profile matcher, is done by the algorithm in Figure 5-10.

```
int scoreMatch(request_Parameter, adv_Parameter) {
        Req_Param_Type = request_Parameter.getParamType()
        Adv_Param_Type = adv_Parameter.getParamType()

        degreeType degree_of_conceptMatch
        if Req_Param_Type = Adv_Param_Type then
                degree_of_conceptMatch = EQUIVALENT
        if Req_Param_Type subsumes Adv_Param_Type then
                degree_of_conceptMatch = SUBSUMES
        if Req_Param_Type subclassOf Adv_Param_Type then
                degree_of_conceptMatch = SUBCLASS
        otherwise degree_of_conceptMatch = FAIL


        Req_Type = request_Parameter.getType()
        Adv_Type = adv_Parameter.getType()

        degreeType degree_of_propertyMatch
        if isUnclassified(Req_Type) or isUnclassified(Adv_Type)  then
                degree_of_propertyMatch= UNCLASSIFIED
        if Req_Type = Adv_Type then
                degree_of_propertyMatch= EQUIVALENT
        if Req_Type isSubPropertyOf Adv_Type then
                degree_of_propertyMatch= SUBPROPERTY
        otherwise degree_of_propertyMatch= FAIL

        return assign_degree(degree_of_conceptMatch,
                                      degree_of_propertyMatch)
}
```

Figure 5-10 Algorithm assigning a matching score for two parameters

The function getParamType() gets the type of the parameter in order to compute the concept match. The function getType() gets the class (property) of the parameter in order to compute the property match.

5.3.2. Model matching

The system allows to choose between the previous profile matchmaker, and the model matchmaker we are about to describe. The model matchmaker has the advantage of being rather sound, but pays the price of its efficiency by being quite time consuming. See Section 4.1.2, for a reminder of the idea of using the service model to match services. The first algorithm we are about to present is copied straight from [9], whereas we've taken some slight liberties with a second version exposed later on this Section.

```
int match(Model_request, Model_advertisement) {

        boolean inputs = parameterMatch(Model_request.getInputs(),
                                        Model_advertisement)

        boolean outputs = parameterMatch(Model_request.getOutputs(),
                                        Model_advertisement)

        int inputs_score, outputs_score
        if inputs then input_score = 1 else inputs_score = 0
        if outputs then output_score = 1 else outputs_score = 0

        int score = weightInput * inputs_score + weightOutput * outputs_score

      return score
}
```

Figure 5-11 Model Matchmaker algorithm

As we can see from the algorithm of Figure 5-11, the matching relation between inputs or outputs of two services is binary: two Inputs or Outputs match or fail; by opposition to the profile matchmaker where several degrees could be assigned to express the level of match. We will see later that these degrees intervene at another point of the algorithm, and also allow flexible matches as the profile matchmaker does.

The call to *parameterMatch* starts the recursive algorithm (launched two times, one for the inputs, one for the outputs) at the root node of the process model of the service. The function, shown in Figure 5-12, simply redirects the execution to the right algorithm corresponding to the type of the node being matched (Split, Sequence, IfThenElse …). An example of those algorithms is given in Figure 5-13, and is nothing more than the algorithm already presented in Figure 4-3 in Section 4.1.2, slightly adapted for our application.

```
boolean parameterMatch(List parameters, Process N) {
        if (N instanceof AtomicProcess) {
                return AtomicMatch(parameters, N)
        }

        if (N instanceof Choice) {
                return ChoiceNode(parameters, N)
        }

        if (N instanceof Split || N instanceof Sequence) {
                return SplitSequenceNode(parameters, N)
        }

        if (N instanceof IfThenElse) {
                return IfThenElse(parameters, N)
        }
(...)

        //Unknown Process
        return false
}
```

Figure 5-12 Function selecting the appropriate algorithm for the corresponding node

In the shown functions, N is a node which can be of any type (*Choice*, *Split*, *Sequence* …), but we also have to assume that it contains a list of its child nodes (*children* element) and a *matchSet* element (set type). The parameter *O* is a list containing the parameters to be matched (either inputs or outputs, depending on the first call to parameterMatch). The operation *head(O)* extracts the first element of the list and *removes* it.

```
boolean SplitSequenceNode(List O, split-seq-Node-Process N) {
        if O is empty then
                return true
        end if

        o1 := head(O)
        for each k in N.children do
                k.matchSet := k.matchSet . {o1}

                if parameterMatch(k.matchSet, k) then
                        if parameterMatch(tail(O), N) then
                                return true
                        end if
                end if

                k.matchSet := k.matchSet - o1
        end for
        return false
}
```

Figure 5-13 Algorithm used to match either Split or Sequence Nodes

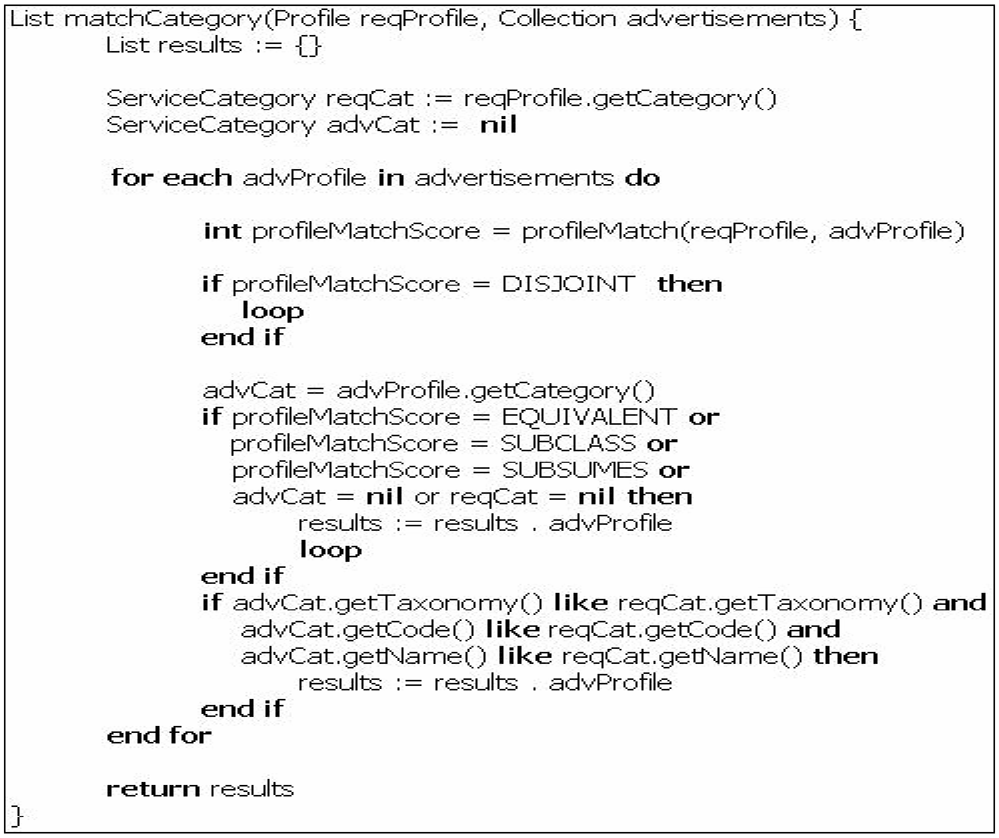
The recursion ends when an atomic process (leaf in the process model tree) is found, and the algorithm used to match this type of node is similar to the algorithm used to match two services in the profile matchmaker: the requested parameters are matched against the parameters of the atomic process and a score is assigned the same fashion as in the function in Figure 5-10 (score assignment in the profile matchmaker). Besides, it is precisely at this moment that the 9 degrees evoked earlier intervene, as a score is assigned to represent the matching degree of two parameters. However, as the general model matchmaker algorithm requires that the result of each node matching functions be a Boolean, we thus have to declare the match of an atomic process a success if the score obtained is greater or equal than a predefined, user-modifiable, threshold. It is in consequence at the atomic process level that the flexibility of the matchmaker is justified. Nevertheless, we should bear in mind that the algorithms of some nodes (e.g. *Split* or *Sequence* nodes) work by distributing the requested parameters to the child nodes, trying to find the appropriate attribution of parameters to each child. Minding this observation, we can easily see that if the threshold is set too low, the distribution of the parameters can be erroneously done (one parameter can be "assigned" to an atomic process whereas it should have been assigned to another one, where it would have had a greater score). As a result, the threshold has to be extremely well chosen not to induce any errors in the matchmaker. The idea of our second version of the model matchmaker comes from that remark; instead of letting the user choose the threshold, the algorithm systematically tries all possible threshold (from 1 to 9) when matching each node. This version should provide optimal results, but with a slightly worse complexity.

5.3.3. Filtering SPAs & Secure mode filter

The platform also lets the user choose two optional modes: "secure" and "Filter SPAs". The modes can be used together or none can be used.

The idea beneath the matchmaking algorithm used in the "secure" mode is to filter a set of services based on limited information, for privacy keeping purposes. Using the information of the profile such as inputs or outputs is considered as too much information giveaway and should be avoided. Other information contained in the profile has thereby to be used, and the service category element seems to be a good candidate as a replacement. Indeed, comparing the elements of the service category seems to be a good way to eliminate incompatible services (if the service category elements does not "match", there is no more need to look at any other elements of the profile, the profiles are advertising divergent services). We can also remark that looking at the class of the profile, i.e. seeing if two profiles belong to the same hierarchy, is also a good indication, for the same reasons.

The "Filter SPAs" mode uses exactly the same algorithm as the "secure mode", but not for the same reason: the mode requires a quick matching to prune a list of services. Profile hierarchy matching and service category comparisons responds well to that demand as it works by doing quick elementary comparisons and eliminates services which have no chances of matching if passed into a profile or model matchmaker.



```
List matchCategory(Profile reqProfile, Collection advertisements) {
        List results := {}

        ServiceCategory reqCat := reqProfile.getCategory()
        ServiceCategory advCat :=  nil

        for each advProfile in advertisements do

                int profileMatchScore = profileMatch(reqProfile, advProfile)

                if profileMatchScore = DISJOINT  then
                   loop
                end if

                advCat = advProfile.getCategory()
                if profileMatchScore = EQUIVALENT or
                  profileMatchScore = SUBCLASS or
                  profileMatchScore = SUBSUMES or
                  advCat = nil or reqCat = nil then
                       results := results . advProfile
                       loop
                end if
                if advCat.getTaxonomy() like reqCat.getTaxonomy() and
                   advCat.getCode() like reqCat.getCode() and
                   advCat.getName() like reqCat.getName() then
                       results := results . advProfile
                end if
        end for

        return results
}
```

Figure 5-14 Category matcher algorithm

The algorithm is shown in Figure 5-14. We can see that the service category is obtained by the function *getCategory()*. Let us notice that two services category do not need to be – exactly – identical to be considered equal, this has the virtue to allow users to use jokers ('*') or even full regular expression when expressing requests. Furthermore the profile matching, as evoked in Section 5.3.1 or 4.1.1, is also used to complete the matching process. Indeed, if the two matched profiles are classified (i.e. have their profile class different from the "Profile" class defined in the daml.org ontologies), the service category comparison is not even used.

Chapter 6. Implementation

As in the previous work, the prototype has been implemented in Java using the JADE multi-agent platform. Our goal in this chapter is to explain some details of the implementation and to understand the structure of the prototype. In order to do so, we will organize our study around the various Java packages:

- Agents package: SSA, SPA and EDF
- Other packages used by the agents:
    - matcher: used for matchmaking services
    - reasoner: used to infer new relationships between elements (used by the matchmaker)
    - storage: used by agents or other package for various storing purposes
    - content: package used for managing various contents: FIPA ACL messages, OWL-S, Strings

Two versions have been implemented, one supporting only owl-s 1.0, another supporting owl-s 1.1. The owl-s 1.1 uses a beta version of the owl-s api. The upgrading to owl-s 1.1 is due to an undocumented bug in the owl-s 1.0 api which prevented the application to work optimally (the version is nevertheless working). Yet, this upgrade has had a good impact on the future scalability on the application, as the owl-s api considerably changed from version 1.0 to 1.1 and now should not include such major changes in future versions. The upgrade of the prototype to greater owl-s version should be hence greatly eased. Besides, we changed the owl-s 1.0 api to make it support multiple profiles, but this could not be done in version 1.1 (the *getProfiles()* method, meant to return all the profiles, is yet still present, but returns a list containing only one random profile).

The first version uses Java 5 features and has necessarily to be used with a Java 5 SDK (generics, "enum" and other new libraries are used). Java 5 could not be used in the second version and we had to downgrade the prototype to Java 1.4 due to the use of the axis package imposed by owl-s 1.1. Indeed, Java 5 cannot be used with Axis libraries as the developers had the dumbest idea to call a package "enum", which is now a reserved word in Java 5.

For reasoning purposes, have been used: Pellet 1.2, Jena 2.2; and for xml parsing purposes: jdom 1.0.

Notice: the javadocs of the application can be found at: http://zill.free.fr/thesis/javadocs/

6.1. Agents

6.1.1. SPA

The service provisioning agents, implemented by the class ServiceProvisionAgent (package agents.jade.SPA), are used to interface a user wishing to provide services to the platform. As seen in the sequence diagrams of Section 5.2, the agent provides the following functionalities:

- Adding a service: *addService()*
- Registering the agent to the DF/registering the services to the EDF: *storeServices(Collection services)*
- Matching a requested service against all its provided services: *match(OWLOntology service, false)*
- Matching the category of a requested service against all its provided service categories: *match(OWLOntology service, true)*

The functionality allowing managing Virtual Organizations (GT4 Index Service) has not been implemented.

All the implemented functionalities are implemented through "behaviors". The table below shows all behaviors of the agent:

Behavior	Description
addServices	Read an OWL-S service at the specified URL and stores the read OWLOntology into the local agent storage, and in its knowledge base. When done, launches the registerSPA behavior.
ListenForReq	A cyclic behavior that listens for incoming requests. If a message is received it will be parsed and the right action will be taken (typically, a search for services, secure or not).
RegisterSPA	Registers itself to the DF, and, if available, sends a reference of shared services to the EDF.
removeServices	Removes the desired stored services from the local agent storage and informs the EDF, if relevant, that these services are not available anymore.
SearchAndResponse	Searches the local storage for the requested service and sends the result back.

Table 6-1 Behaviors of the Service Provision Agent

6.1.2. SSA

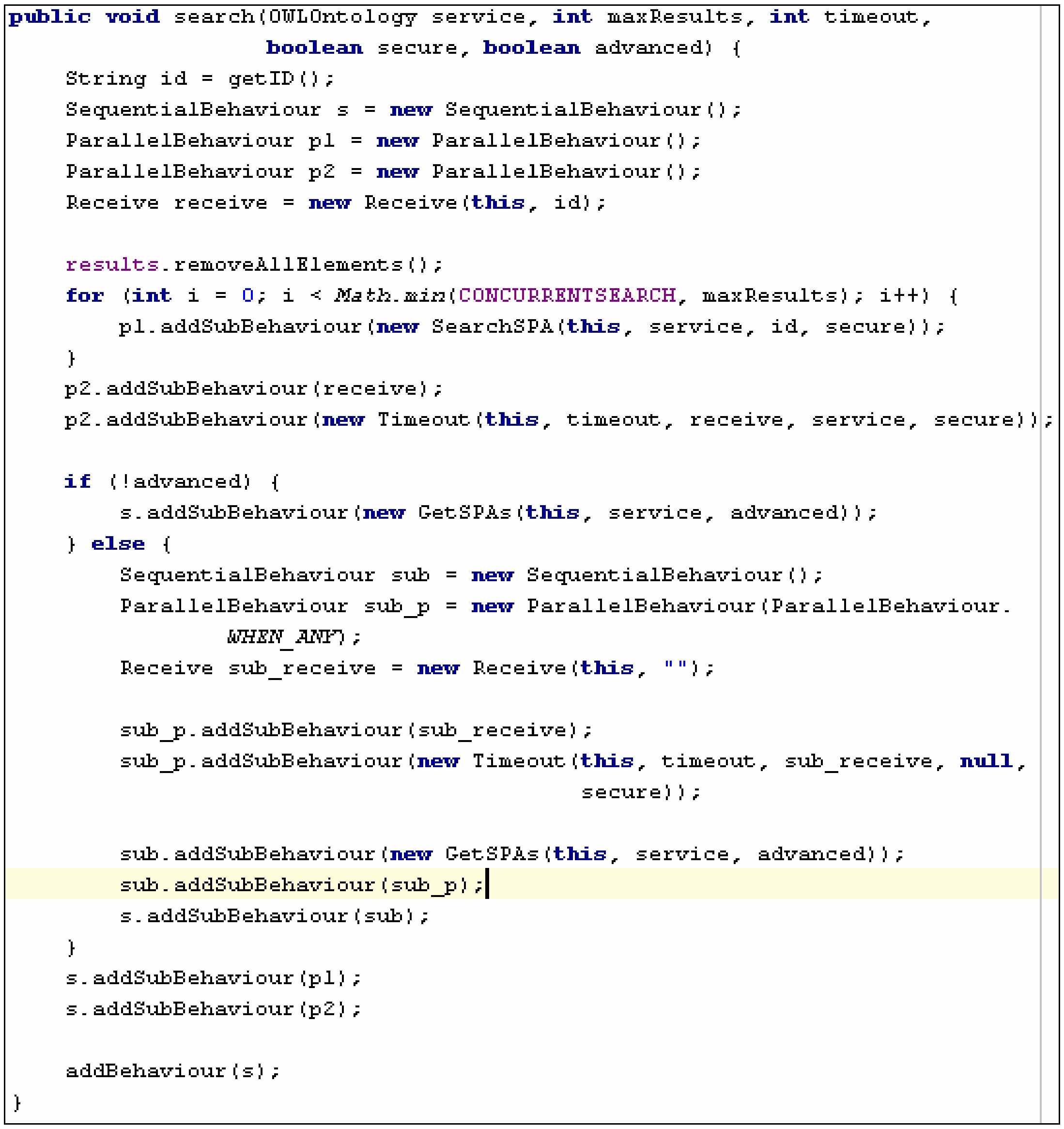
The service selection agents, implemented by the class ServiceSelectionAgent (package agents.jade.SSA), are used to interface a user wishing to find services. As seen in the sequence diagrams of Section 5.2, the agent provides the user one functionality: searching services matching the desired requirements, specified in an OWL-S file.

Behavior	Description
GetSPAs	Sends a request to the DF, or to the EDF ("Filter SPAs" mode), to get available SPAs to contact.
SearchSPA	Sends a request to one given SPA to find a desired service.

Receive	A cyclic behavior that listens for incoming messages. The behavior typically collects returned results following requests emitted by "GetSPAs" or "SearchSPA".
Timeout	A "waker" behavior, i.e. a behavior that sleeps and wakes up after a given timeout to terminate the collection of search results (shuts down the "Receive" behavior)

Table 6-2 Behaviors of the Service Selection Agent

Below is shown the function initiating a search for a service and determining the way the agent behaves. This single method is called whenever a user wants to find a service and it specifies the sequence the behaviors are executed.



```
public void search(OWLOntology service, int maxResults, int timeout,
                   boolean secure, boolean advanced) {
    String id = getID();
    SequentialBehaviour s = new SequentialBehaviour();
    ParallelBehaviour p1 = new ParallelBehaviour();
    ParallelBehaviour p2 = new ParallelBehaviour();
    Receive receive = new Receive(this, id);

    results.removeAllElements();
    for (int i = 0; i < Math.min(CONCURRENTSEARCH, maxResults); i++) {
        p1.addSubBehaviour(new SearchSPA(this, service, id, secure));
    }
    p2.addSubBehaviour(receive);
    p2.addSubBehaviour(new Timeout(this, timeout, receive, service, secure));

    if (!advanced) {
        s.addSubBehaviour(new GetSPAs(this, service, advanced));
    } else {
        SequentialBehaviour sub = new SequentialBehaviour();
        ParallelBehaviour sub_p = new ParallelBehaviour(ParallelBehaviour.
                WHEN_ANY);
        Receive sub_receive = new Receive(this, "");

        sub_p.addSubBehaviour(sub_receive);
        sub_p.addSubBehaviour(new Timeout(this, timeout, sub_receive, null,
                                          secure));

        sub.addSubBehaviour(new GetSPAs(this, service, advanced));
        sub.addSubBehaviour(sub_p);
        s.addSubBehaviour(sub);
    }
    s.addSubBehaviour(p1);
    s.addSubBehaviour(p2);

    addBehaviour(s);
}
```

Figure 6-1 Method initializing a service search

Figure 6-2 clarifies the function by showing a diagram representing the succession of the behaviors. The ovals of the diagram represent the composite behaviors (the names from the function above have been kept), the squares are the behaviors described in Table 6-2. Grossly, the scheme redundantly used is:

- One (or more) request(s) is emitted (GetSPAs or SearchSPA) to one or more agent(s) (EDF or SPA(s)), then
- A couple Receive-Timeout comes and collects the results:
  - Receive and Timeout works in parallel: Receive collects all incoming results indefinitely until Timeout wakes up to shut it down.

The diagram below corresponds to the "Filter SPAs" mode. If working without the mode, the composite behavior "sub" is simply replaced by the behavior GetSPAs (which do not need to use a Receive-Timeout couple as it can get its result immediately). Note that working in "secure" mode or not does not affect the flow of behaviors.

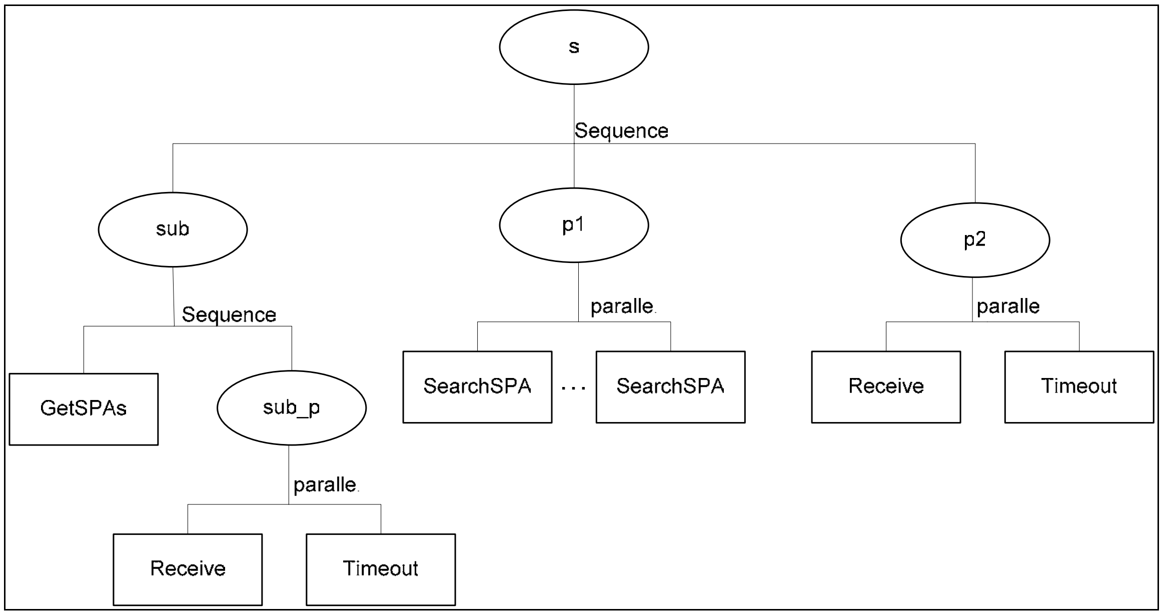


Figure 6-2 Behaviors flow of the service search

6.1.3. EDF

The Extended Directory Facilitator is a novelty brought up to enhance the Directory Facilitator included in the JADE platform. Ideally, the DF should be changed in order to fit our needs; unfortunately, this agent is a predefined JADE agent and hence cannot be modified, it is also already exploited at its full possibilities. The EDF It is implemented in the class DFWrapperAgent in the package agents.jade.DF. The EDF is only optionally launched, but if present, the SPAs systematically remain in contact with it to make it hold an up-to-date list of services provisioned by each of them. It is furthermore used only when SSAs specifically ask to work in "Filter SPAs" mode, in which case the EDF will be used to select in its storage, SPAs providing services that could match the request an SSA sent.

Behavior	Description
Receive	A cyclic behavior that listens for incoming messages. Messages received can be: ▪ An SPA registering services ▪ An SPA deregistering services ▪ An SSA requesting a list of SPAs
Reply	Sends the desired list of SPAs to the SSA originating the request received by the behavior "Receive".
RegisterDF	Register itself to JADE's DF so it can be found by SPAs and SSAs.

Table 6-3 Behaviors of the Extended Directory Facilitator

6.2. Other packages

6.2.1. matcher

The matcher package holds the modules in charge of matchmaking services. The matching methods used (Profile matcher and the two versions of the model matcher) implement the OWLServiceMatcher interface, and hence provide the following functions:

- public void addServices(Collection services): Adds a collection of services to the knowledge base
- public void clearKb(): Clears the knowledge base
- public Vector match(OWLOntology request, Collection advertisements, int minScore): returns a collection of services contained in *advertisements*, which match *request* with a score greater or equal than *minScore*.
- public Vector match(OWLOntology request, Collection advertisements): returns a collection of services contained in *advertisements*, which match *request* with a score greater or equal than a default score specified in the configuration file (see Section 6.3.2).
- public int match(OWLIndividual request, OWLIndividual advertisement): matches request against advertisement, and returns the score obtained.
- public Vector matchCategory(OWLOntology request, Collection advertisements): returns a collection of services contained in *advertisements*, which category match *request*'s category.

The implementation of the profile matcher and of the model matchers follows carefully the algorithms specified in Section 5.3.

Note: the model matcher could not be fully implemented, the authors of the algorithms could not be contacted and we had to settle for two node algorithms (Split, Sequence/Choice). Nonetheless, the few missing algorithms only need a trivial transcription in Java and can then be easily inserted in the model matcher class.

6.2.2. reasoner

The reasoner package includes two interfaces defined to provide reasoning mechanisms, to be used by the service matchers. The interface Reasoner proposes methods necessary to infer new information from an ontology, whereas the interface Matching proposes functions meant to use the Reasoner interface, in order to perform concept match, property match, and score assignment.

Reasoner
boolean subsumes(URI conceptA, URI conceptB)
boolean isSubClass(URI conceptA, URI conceptB)
boolean isEquivalentClass(URI conceptA, URI conceptB)
boolean isSuperProperty(URI propertyA, URI propertyB)
boolean isSubProperty(URI propertyA, URI propertyB
boolean isEquivalentProperty(URI propertyA, URI propertyB)
boolean isUnclassified(URI property)

Table 6-4 Reasoner interface

Matching
int conceptMatch(URI conceptA, URI conceptB)
int propertyMatch(URI propertyA, URI propertyB)
int scoreMatch(Parameter req, Parameter adv)

Table 6-5 Matching interface

6.2.3. storage

The storage interfaces define methods to manage services (OWLOntology type) storage. We can encounter two types of storage: collection or map storage. The classes implementing collection storage needs to present the following functions:

- add one service
- add a collection of services
- remove a service
- remove several services
- get all the services

The class VectorStorage implements this interface with vectors. Databases are presurmised to present a good version of collection storage, but were not implemented.

The class implementing the map storage needs to present the following functions:

- *public void add(Object key, Object value):* adds a couple <Key, Value>
- *public void add(Collection keys, Object value):* adds couples <Key, Value> with several keys having the same value
- *public Object get(Object key):* gets the value associated to the key
- *public Collection getFiltered(OWLOntology filterService):* Values are interpreted as Services: gets Services whose category match the filter service *filterService* category.
- *public Object remove(Object key):* removes the *key* and its *value*

- *public void removeValue(Object key, Object value):* removes the couple <*key*, *value*>
- *public void removeValues(Object value):* removes all keys associated to the value *value*
- *public Map getAll():* gets the underlying map
- *public Collection getValues():* gets the collections of values
- *public Collection getKeys():* gets the collection of keys

The class MultiHashMapStorage implements this interface with multimaps, i.e. maps having keys associated with a list of values.

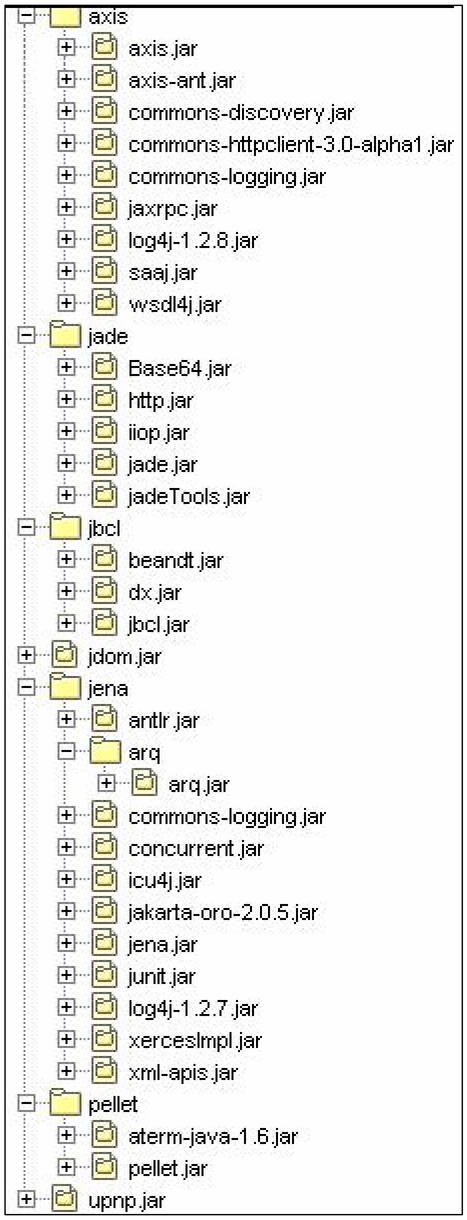
6.2.4. content

The content package includes classes for managing the content carried in the ACL messages. The package has been divided into three sub packages:

- lang: contains the structure of ACL messages (see Section 4.3.1 in [1])
- owls: contains static functions used to read and write OWL-S files. The reading functions return OWL Services (of type OWLOntology) (translated to OWL-S 1.1 if necessary) from an input URI where a service is located, or from a String containing an OWL Service. The writing functions write OWLOntology objects (OWL Services in the OWL-S API) into Strings (basically so that the service can be sent on the wire).
- Strings: contains functions used to translate vector to strings and vice-versa, so that vectors can be passed as arguments in ACL messages. Note: the serialization of Java Vectors cannot be used in an ACL message as it contains forbidden characters for XML's CDATA elements. Vectors are hence translated into strings the following way: Vector: <elem1, elem2, elem3> ➔ "elem1^elem2^elem3", where '^' is considered as a delimiter between elements.

6.3. Manual

6.3.1. Required software and libraries



```
axis
    axis.jar
    axis-ant.jar
    commons-discovery.jar
    commons-httpclient-3.0-alpha1.jar
    commons-logging.jar
    jaxrpc.jar
    log4j-1.2.8.jar
    saaj.jar
    wsdl4j.jar
jade
    Base64.jar
    http.jar
    iiop.jar
    jade.jar
    jadeTools.jar
jbcl
    beandt.jar
    dx.jar
    jbcl.jar
jdom.jar
jena
    antlr.jar
    arq
        arq.jar
    commons-logging.jar
    concurrent.jar
    icu4j.jar
    jakarta-oro-2.0.5.jar
    jena.jar
    junit.jar
    log4j-1.2.7.jar
    xercesImpl.jar
    xml-apis.jar
pellet
    aterm-java-1.6.jar
    pellet.jar
upnp.jar
```

Figure 6-3 Required libraries

As explained in the beginning of the chapter, the prototype (OWLS-1.1) is meant to work under a Java 1.4 JVM (the prototype version OWL-S 1.0 has to work under a Java 1.5 JVM). The prototype runs on top of the JADE platform.

So, in order to run the agents a JADE container must be initialized and all the libraries dependencies have to be included in the classpath. All the required libraries are shown in Figure 6-3, and are all contained in the "lib/" folder of the project. Beware of not creating libraries version conflicts, when upgrading one of the dependencies, as those libraries interlace each other and need precise versions to work properly at execution. The typical command used to launch an SPA, with its GUI, would be:

```
> javaw –classpath <classpath> jade.Boot -container
agent_name:agents.jade.SPA.ServiceProvisionAgent
```

A windows batch file is provided to simply launch any agents by the syntax:

>launchAgent Agent_Type Agent_Name

Where Agent_Type is amongst: *SPA, SSA* or *EDF.*

6.3.2. Configuration

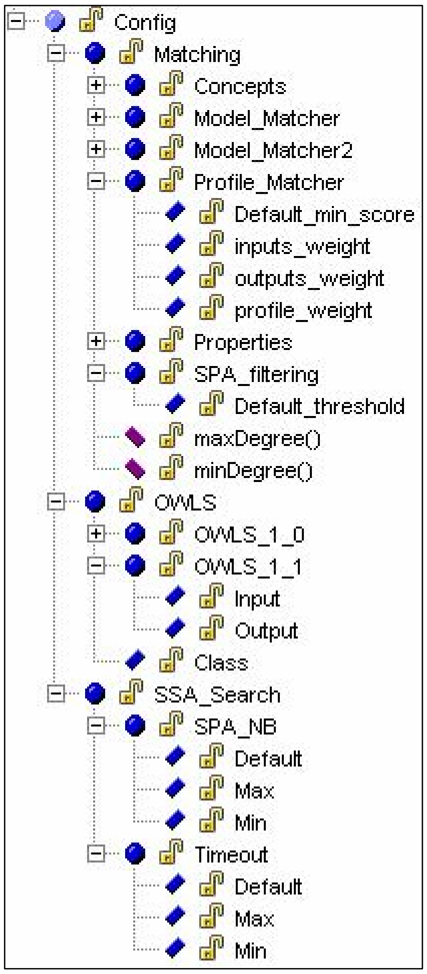


Figure 6-4 Configuration class

A configuration class is provided which is used to hold all the constants or default parameters, so that everything can easily be modified. For instance, the weight of the outputs in the profile matcher (Section 5.3.1) is in:

"Config.Matching.Profile_Matcher.outputs_weight".

To get the default Timeout (Section 6.1.2) for services search in the SSA:

"Config.SSA_Search.Timeout.Default"

6.3.3. GUIs & Application example

The prototype includes two user interfaces, one for the providers, and one for the requesters. The interface for requesters is more for a testing purpose as this side of the system is more to be used by automats (e.g. in a composite service building process), and the user interface can hence be shortcut to directly start a search by the function:

*public void search(OWLOntology service, int maxResults, int timeout,
boolean secure, boolean advanced);*

located in the SSA class. Let us explain the parameters:

- service: the requested service (OWLOntology class with owl-s 1.1 api or Service class with owl-s 1.0 api).
- maxResults: indicates the maximum numbers of SPAs to probe.
- Timeout: the time, in milliseconds, to spend collecting requests (after that timeout, the agent will not receive any replies sent by SPAs).
- Secure: set to true if the search has to be done in "secure" mode, false otherwise.
- Advanced: set to true if the search has to be done in "Filter SPAs" mode, false otherwise.

On the other hand, we encourage user to use the GUI for the Service Provision Agent, but if not wished, one could simply start the agent and add services with the function

public void addService(URI EPR);

located in the SPA class. *EPR* is the URI of the service to add. Service can be removed by:

public void removeServices(Collection EPRs);

EPRs is a collection of services to remove. Adding and removing are the sole actions a user can undertake with an SPA, all others (matching …) are automatically triggered when a request is received.

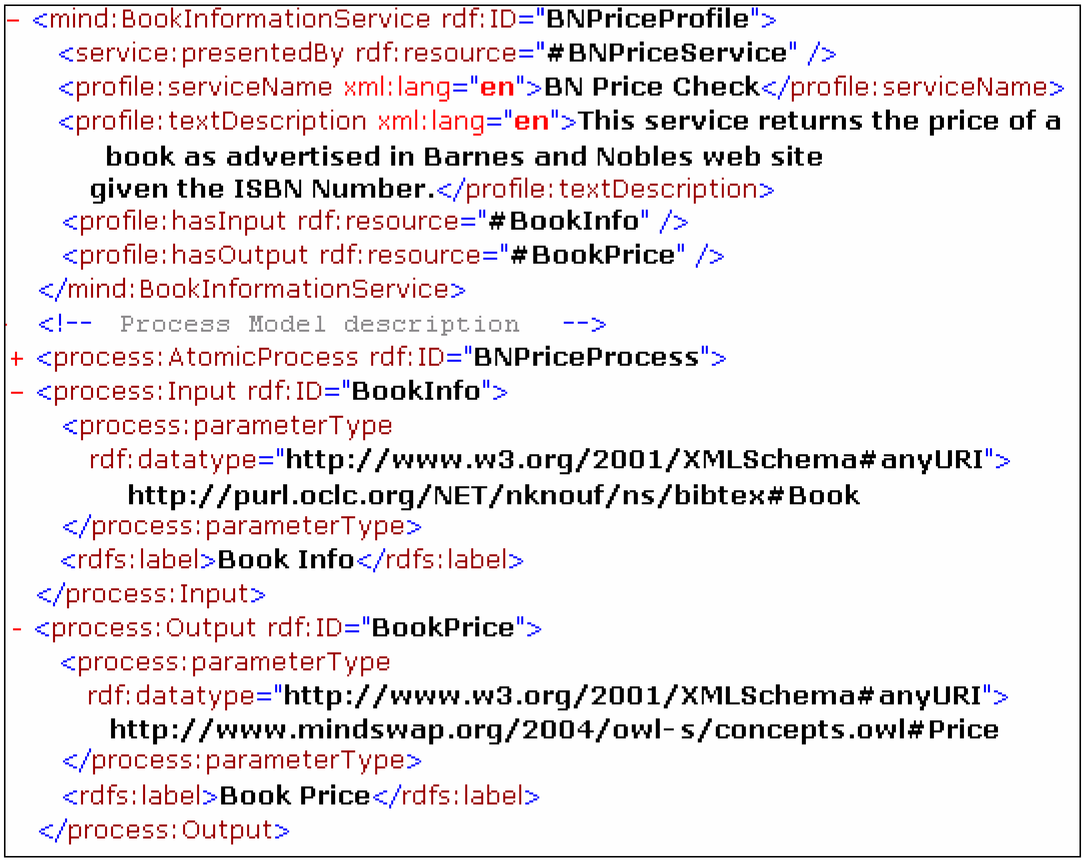
Next is shown an application example, where the request is the file "request.owl" and where the only SPA running is advertising the service "BNPPrice.owl" existing at the address: http://www.mindswap.org/2004/owl-s/1.1/BNPrice.owl (The SPA is also provisioning two other services that will not match, for brevity sake, we will only show the significative part of their profile in Appendix B). The listings of both OWL-Services are shown in Appendix, and Figure 6-7 and Figure 6-8 show and explain how to handle the user interfaces, using the application example.

Below are also shown the parts of the profiles relevant in the profile matchmaker (Figure 6-5and Figure 6-6), and are used to explain the obtained results. Let us compute the result "by hand" before seeing what the application will give us:

- Outputs score: the advertisement (Figure 6-5) presents one output named "BookPrice", which can be found in the ontology http://www.mindswap.org/2004/owl-s/concepts.owl. The request (Figure 6-6) also presents one output, which is exactly the same. The concept match score should hence be 3 (perfect match). Their properties are both "Output", meaning they are both unclassified and obtain 0 at the property match score. The final Outputs score is hence 3 (= 3+0).

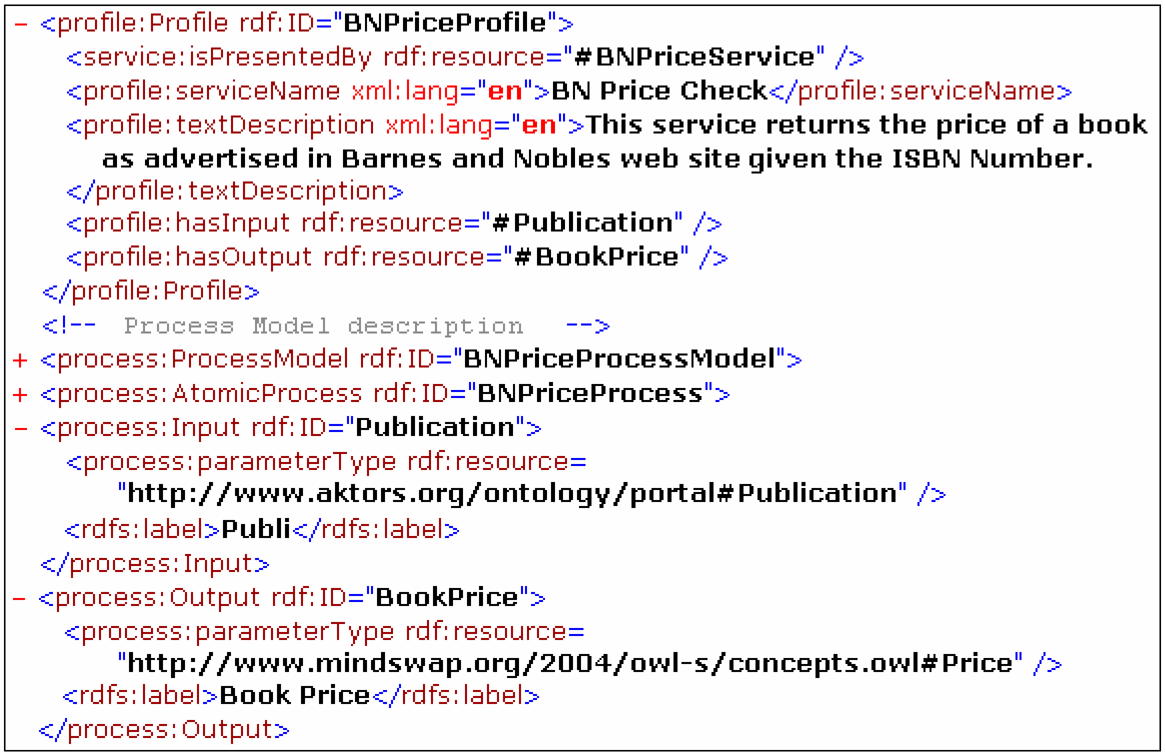
- Inputs score: the advertisement presents one input named BookInfo, of type "Book", as found in the ontology http://purl.oclc.org/NET/nknouf/ns/bibtex. The request presents one input named "Publication", as found in the ontology http://purl.oclc.org/NET/nknouf/ns/bibtex**.** In this ontology (shown in Appendix), we can see that "Publication" is a super-class of "Book". The final score for the outputs should hence be 2 (= 2 + 0, as the inputs are also unclassified).
- Profile score: The advertisement has classified its profile (seldom), and its class is "BookInformationService" and can be found at the ontology: http://www.mindswap.org/2004/owl-s/1.1/MindswapProfileHierarchy.owl (cannot be seen in Figure 6-5, see Appendix instead). However, the request has not classified its profile ("Profile" class), the matchmaker cannot adjudicate a match or mismatch, and should declare the profile match as "unclassified" (score 1)

With default parameters (weight of the outputs score equals to 100, inputs 10, and profile 1), the global score of these two matched services should hence be 321.



```
- <mind:BookInformationService rdf:ID="BNPriceProfile">
    <service:presentedBy rdf:resource="#BNPriceService" />
    <profile:serviceName xml:lang="en">BN Price Check</profile:serviceName>
    <profile:textDescription xml:lang="en">This service returns the price of a
        book as advertised in Barnes and Nobles web site
      given the ISBN Number.</profile:textDescription>
    <profile:hasInput rdf:resource="#BookInfo" />
    <profile:hasOutput rdf:resource="#BookPrice" />
  </mind:BookInformationService>
  <!--  Process Model description   -->
+ <process:AtomicProcess rdf:ID="BNPriceProcess">
- <process:Input rdf:ID="BookInfo">
    <process:parameterType
      rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
        http://purl.oclc.org/NET/nknouf/ns/bibtex#Book
    </process:parameterType>
    <rdfs:label>Book Info</rdfs:label>
  </process:Input>
- <process:Output rdf:ID="BookPrice">
    <process:parameterType
      rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
      http://www.mindswap.org/2004/owl-s/concepts.owl#Price
    </process:parameterType>
    <rdfs:label>Book Price</rdfs:label>
  </process:Output>
```

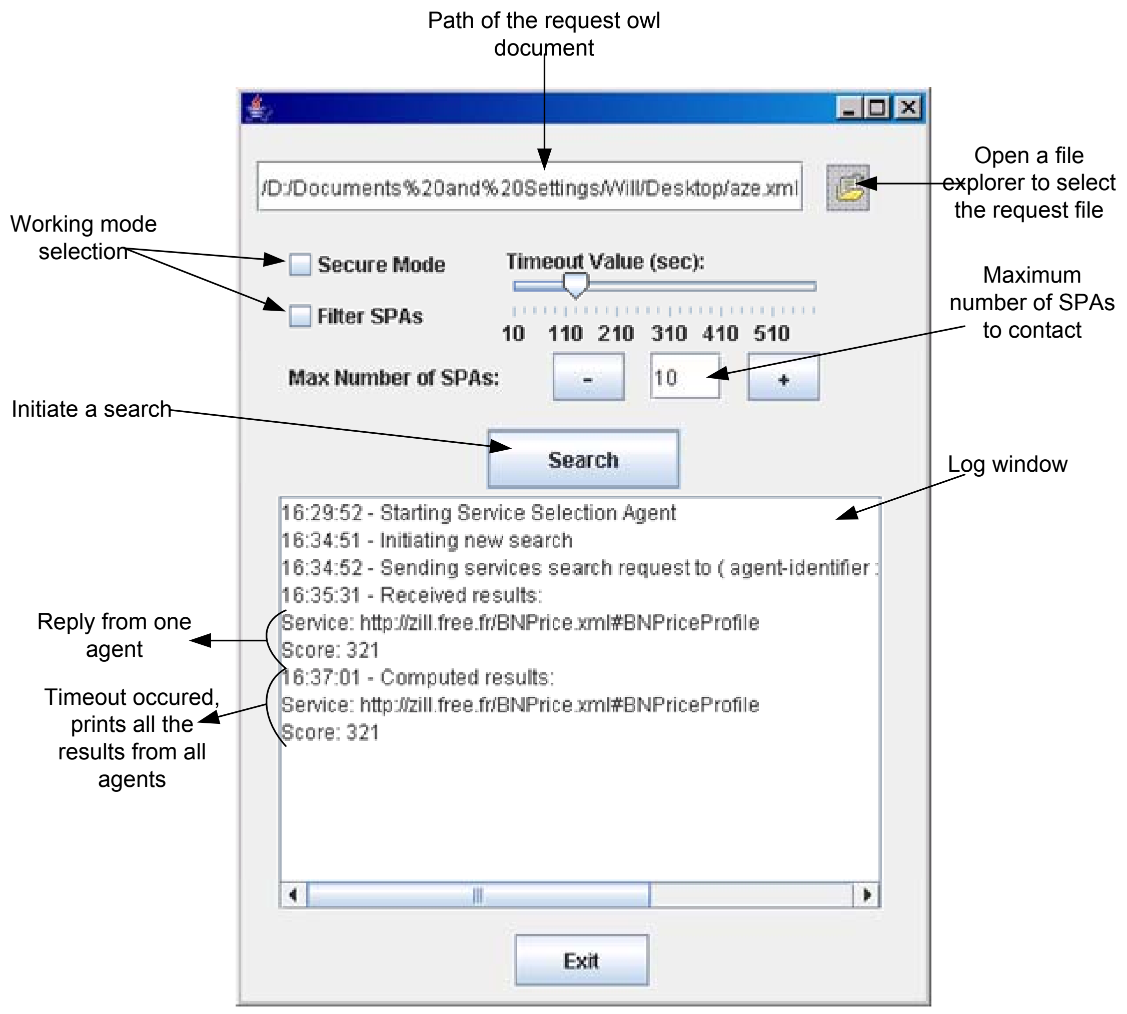
Figure 6-5 Excerpt of the advertisement's profile



```
- <profile:Profile rdf:ID="BNPriceProfile">
    <service:isPresentedBy rdf:resource="#BNPriceService" />
    <profile:serviceName xml:lang="en">BN Price Check</profile:serviceName>
    <profile:textDescription xml:lang="en">This service returns the price of a book
       as advertised in Barnes and Nobles web site given the ISBN Number.
    </profile:textDescription>
    <profile:hasInput rdf:resource="#Publication" />
    <profile:hasOutput rdf:resource="#BookPrice" />
  </profile:Profile>
  <!--  Process Model description   -->
+ <process:ProcessModel rdf:ID="BNPriceProcessModel">
+ <process:AtomicProcess rdf:ID="BNPriceProcess">
- <process:Input rdf:ID="Publication">
    <process:parameterType rdf:resource=
        "http://www.aktors.org/ontology/portal#Publication" />
    <rdfs:label>Publi</rdfs:label>
  </process:Input>
- <process:Output rdf:ID="BookPrice">
    <process:parameterType rdf:resource=
        "http://www.mindswap.org/2004/owl-s/concepts.owl#Price" />
    <rdfs:label>Book Price</rdfs:label>
  </process:Output>
```

Figure 6-6 Excerpt of the request's profile

Explained snapshot of the SSA when running the example:


Path of the request owl document
Open a file explorer to select the request file
Working mode selection
Maximum number of SPAs to contact
Initiate a search
Log window
Reply from one agent
Timeout occured, prints all the results from all agents
Secure Mode
Filter SPAs
Timeout Value (sec):
10 110 210 310 410 510
Max Number of SPAs:
-
+
Search
Exit

/D:/Documents%20and%20Settings/Will/Desktop/aze.xml

Max Number of SPAs: 10

```
16:29:52 - Starting Service Selection Agent
16:34:51 - Initiating new search
16:34:52 - Sending services search request to ( agent-identifier :
16:35:31 - Received results:
Service: http://zill.free.fr/BNPrice.xml#BNPriceProfile
Score: 321
16:37:01 - Computed results:
Service: http://zill.free.fr/BNPrice.xml#BNPriceProfile
Score: 321
```

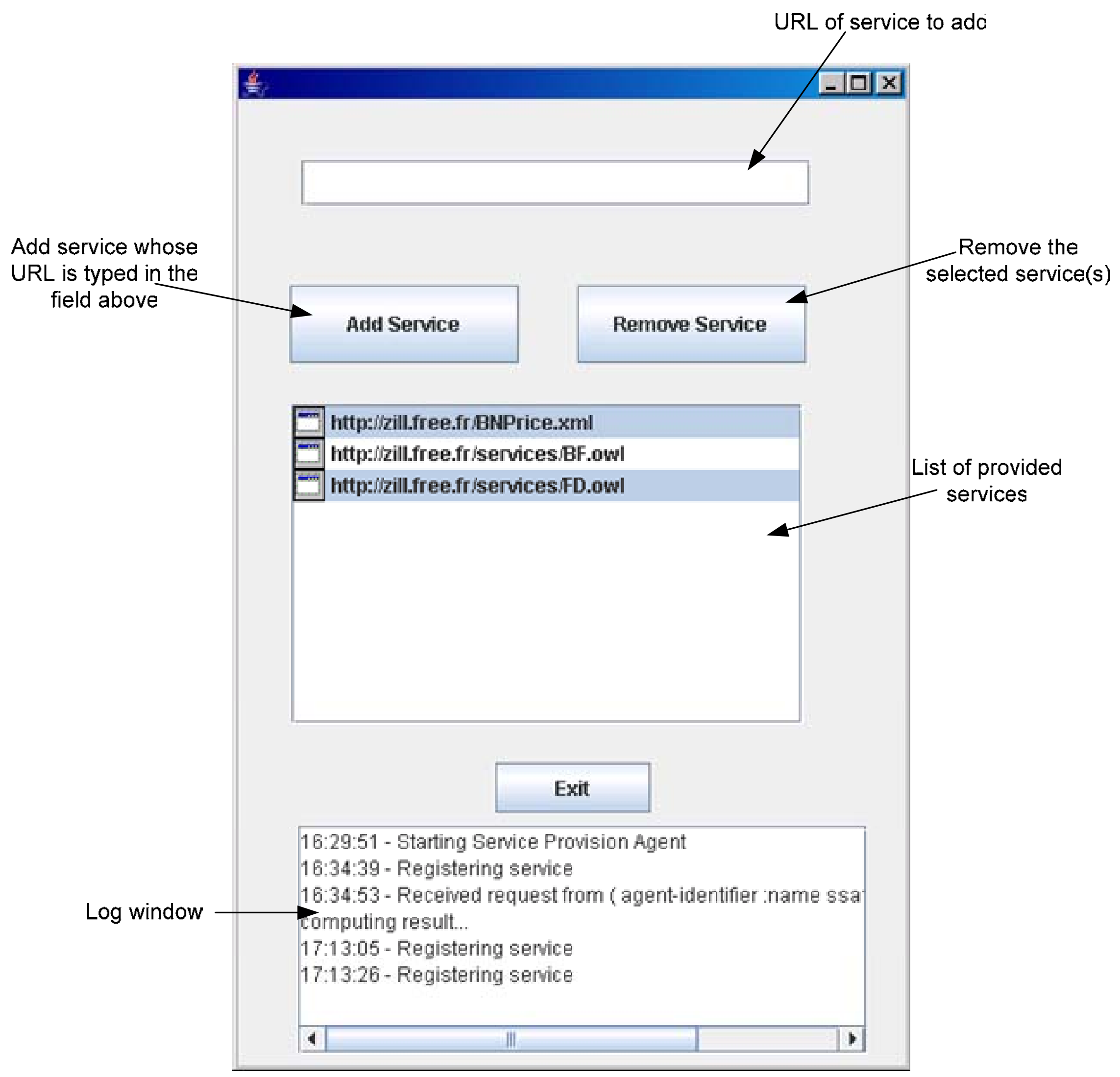
Figure 6-7 Service Selection Agent GUI

With the default settings of the Configuration class (see Section 6.3.2) the score obtained from the only result (321) can be interpreted as follows:

- 3 is the score obtained from the output match (output weight is set to 100 by default). '3' means that all the outputs of both services perfectly match, but that the property of at least one of the outputs is unspecified (3 = 3 + 0)
- 2 is the score obtained from the input match (input weight is set to 10 by default). '2' means that at least one input of the request subsumes its corresponding input in the advertisement, and that the other inputs (if any) match perfectly. Here again, the property of at least one input is unspecified (2 = 2 + 0)
- 1 is the score obtained from the profile match (profile weight is set to 1 by default). '1' means that at least one profile was unclassified.

We can see that the two other services the SPA advertised (http://zill.free.fr/FD.owl and http://zill.free.fr/BF.owl, as seen in next page) were not selected as they did not match the request sufficiently.

Explained snapshot of the SPA when running the example:


URL of service to add
Add service whose URL is typed in the field above
Add Service
Remove Service
Remove the selected service(s)
http://zill.free.fr/BNPrice.xml
http://zill.free.fr/services/BF.owl
http://zill.free.fr/services/FD.owl
List of provided services
Exit
Log window

```
16:29:51 - Starting Service Provision Agent
16:34:39 - Registering service
16:34:53 - Received request from ( agent-identifier :name ssa
computing result...
17:13:05 - Registering service
17:13:26 - Registering service
```

Figure 6-8 Service Provision Agent GUI

We can notice the time spent at different step of the application:

The SPA sent its request at 16:34:52, which is received one second later at 16:34:53 by the SPA, which immediately starts computing the results. The results are then received by the SSA at 16:35:31, 39 seconds later. At 16:37:01, the SSA computes its results, meaning that its timeout for receiving results has expired (129 seconds after the search was initiated, which corresponds to the timeout value indicated by the slide bar of the SSA's GUI).

Chapter 7. Proposed Solution – Evaluation

In this chapter we will evaluate the performances of the application, essentially to locate bottlenecks and pinpoint sensitive points using most of the application time. This should be useful for future improvements of the system. All these tests were made with a Pentium IV (mobile), 1.2Ghz and 512Mb DDR-Ram. We can notice that these evaluations are relevant for both OWL-S 1.0 and OWL-S 1.1 prototypes (the reasoning, independent from the OWL-S API, is essentially evaluated).

7.1. Time performances

7.1.1. Matchmaking

We are first going to study the matchmaking case. To evaluate the time needed, we decomposed the process into its elementary steps:

- Reading (parsing) services at a given URI (i.e. *Reader.read(uri)*):

We read ten ontologies of different sizes to assess the time needed in function of their size, the times obtained are presented in Table 7-1 below:

Time (ms)	URIs of services	Size (Kb)
511	http://www.mindswap.org/2004/owl-s/1.1/MindswapProfileHierarchy.owl	4.9
550	http://www.mindswap.org/2004/owl-s/1.1/BNPrice.owl	5.7
561	http://www.mindswap.org/2004/owl-s/1.1/GoogleSpelling.owl	5.0
1042	http://www.mindswap.org/2004/owl-s/1.1/ZipCodeDistance.owl	7.2
1072	http://www.mindswap.org/2004/owl-s/1.1/GoogleSearch.owl	5.3
4056	http://www.mindswap.org/2004/owl-s/1.1/BabelFishTranslator.owl	12
4366	http://www.mindswap.org/2004/owl-s/1.1/FindLatLong.owl	6.8
5077	http://www.mindswap.org/2004/owl-s/1.1/CurrencyConverter.owl	7.6
8292	http://www.mindswap.org/2004/owl-s/1.1/FrenchDictionary.owl	8.7
20449	http://www.mindswap.org/2004/owl-s/1.1/FindCheaperBook.owl	16

Table 7-1 List of services used for testing purposes

The times obtained are represented in the chart in Figure 7-1 below

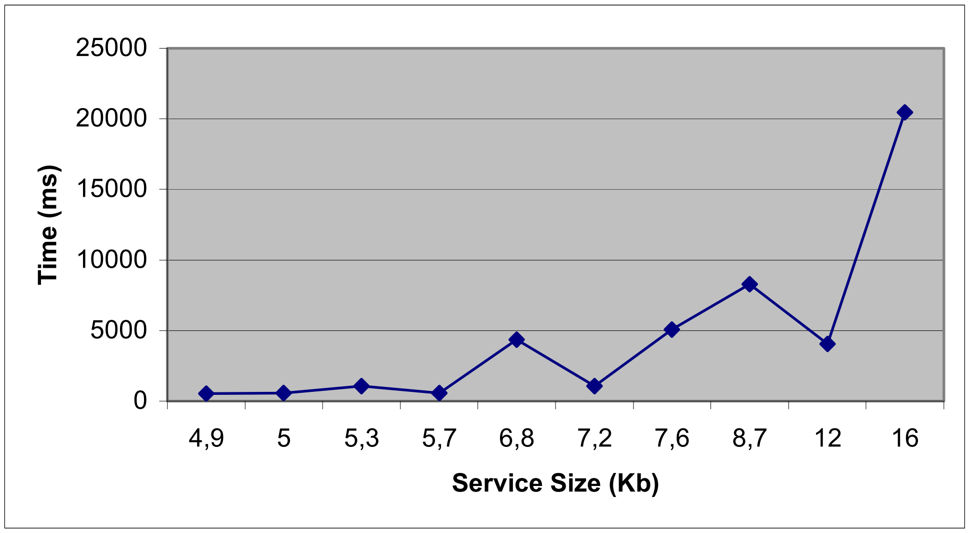


Figure 7-1 Services parsing time chart

As we can see, the time needed to parse services increases exponentially with the ontologies size.

- Adding ontologies to the reasoner

Once the ontology has been parsed to an OWLOntology Object, the time needed to add it to the knowledge base is constantly equal to 0ms.

- Getting a class in the ontology (*kb.getOntClass(uri)*)

When the reasoner has to compute the relation between two concepts, it has to first get the concepts, given by their URIs, from the ontology. This time is similar to getting a property in the ontology (*kb.getOntProperty(uri)*). From our experiences, this time does not vary much depending on the size of the ontology. Yet, much bigger ontologies can make getting the class take a bit longer. To stress that point, instead of inserting each ontology one by one, we cumulated the ontologies in the knowledge base until nine ontologies were inserted. We obtained the times shown in Figure 7-2 below:

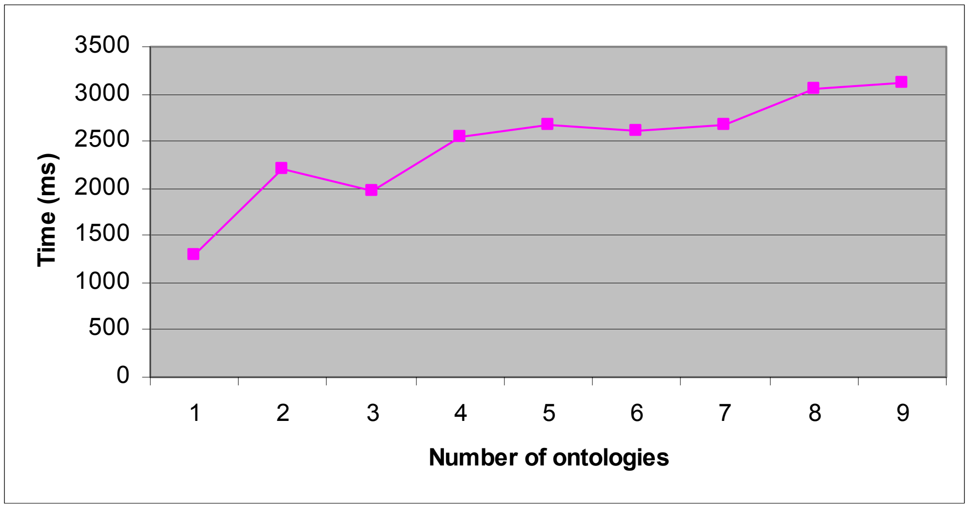


Figure 7-2 Class getting time chart

As we can see, even with bigger ontologies, the time needed to get the class still does not vary much, but is still quite long (~3 sec) for an operation that will be done a certain amount of times.

- Determining relationships between elements (e.g. *subsumes(uriA, uriB)*)

Once two classes have been obtained in the ontology, the reasoner can infer new relationships between them two. The time needed depend on the size of the ontology probed. To get the times from Figure 7-3 we inserted the ontology http://www.aktors.org/ontology/portal, and then added one to nine more ontologies in the knowledge base. The times given are the times needed to evaluate always the same relationship: *subsumes("Book", "Publication"*). The concepts "Book" and "Publication" are found in the mentioned portal ontology and the result of the request is voluntarily false ("Publication" subsumes "Book"), so that the worst time is assessed. The times obtained are shown in Figure 7-3 below

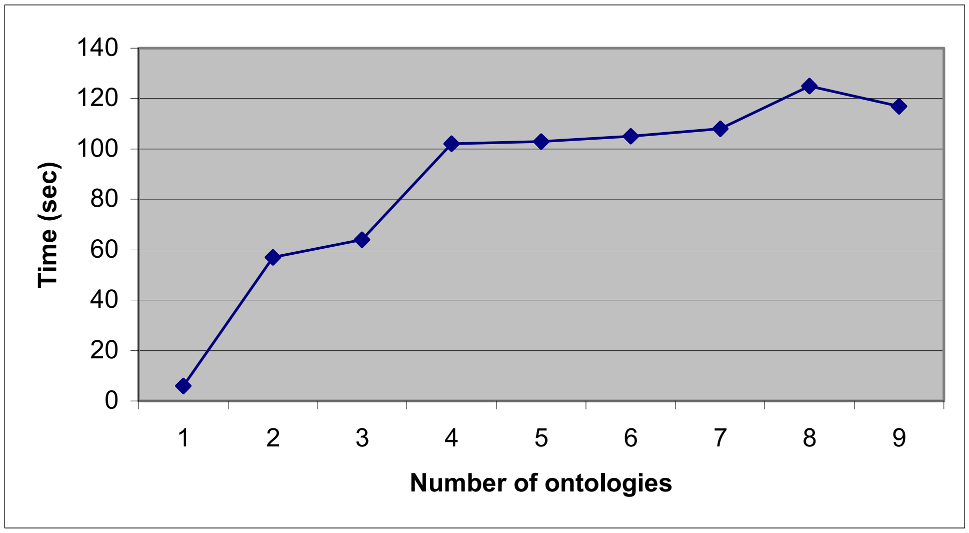


Figure 7-3 Relationships inferring time chart

NB: The order of insertion of the ontologies is the order of their ascending size
NB2: When one ontology is inserted, all its imports are also inserted.

Obviously, the time is way too long as soon as the knowledge base gets bigger (more than one minute as soon as more than two ontologies are inserted), but progresses linearly/logarithmically with the size of the knowledge base.

The following chart of Figure 7-4 shows how a matchmaking process shares its time between the different activities. We took for each activity the average time obtained and multiplied it by the number of times it was done in a matchmaking process. The function *scoreMatch(…)*, assigning a score for two parameters (see 5.3.1) includes five calls to "Determining relationships" at worst, ten to "getting a class" (2 times for each relationships inferring) at worst and "parsing services" is done 2 times in total. *scoreMatch(…)* is then called (*nb_Outputs_adv* x *nbOutputs_req* + *nb_Inputs_Req* x *nb_Inputs_Adv*) times. Regular services have in general 2 inputs and 1 output, we can then estimate the call to *scoreMatch(…)* will be done 5 times. Moreover we have 3 more calls to "Determining relationships" and 6 more for "getting a class" (profile match). Thus, we finally have to count:

- 5*5 + 3 = 28 calls to "Determining relationships"
- 10*5 + 6 = 56 calls to "getting a class"
- 2 calls to "Parsing service"

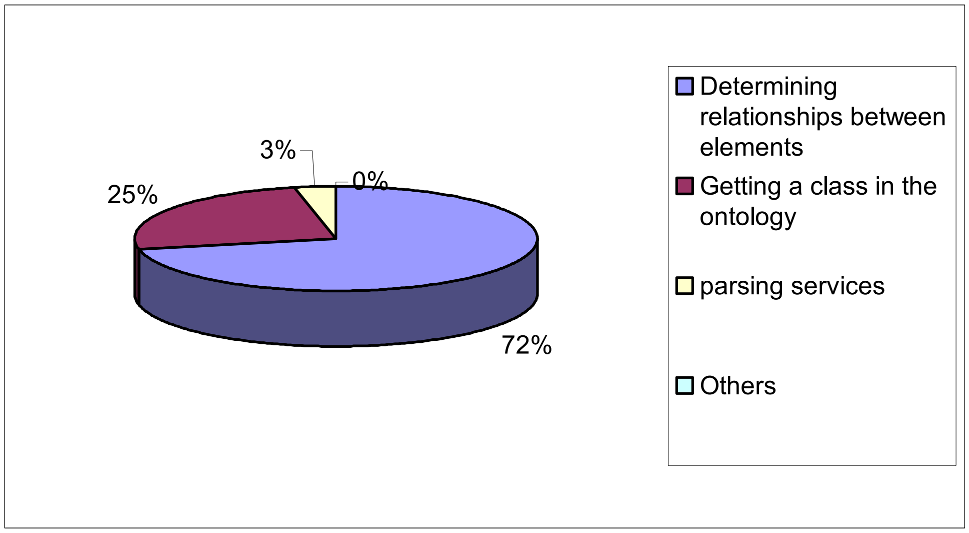


Figure 7-4 Time repartition in a matchmaking process

We can see that the reasoner takes 97% of the time spent by the matchmaker. Among that 97%, 25% is spent by the class getting operation which effectuates a huge number of redundant calls. Indeed, in our example, 56 calls are made to *getOntClass(…)*, whereas we are getting only 3 different classes in total (two inputs parameters and one output parameter, as chosen in the example). Such an optimization is easily doable, but would not make big profits anyway, "getting a class in the ontology" would shrink down to 3%, but "determining relationships" would still represent 93% (1% of time won). One big effort is thus to choose a better reasoner capable of inferring much faster.

7.1.2. Application

To evaluate the performances of the application as a whole we only need to assess the time needed for communicating between agents and some other minor algorithmic processes (Vector-String translations …). However, we cannot evaluate the time needed to communicate as it is not dependant of our application but of the quality of the connection and of the physical distance between agents. Moreover, the other notable operations, apart from matchmaking: building/parsing ACL messages and translating Vectors to String and vice-versa (see Section 6.2.4) are operations needing negligible time (less than 100ms). We can hence consider that the performances of the application are the performances of the matchmaker (which includes OWL-Services parsing). In a case where the held knowledge base is quite large (more than 10 different 10Kb ontologies), matching two services can take up to five minutes, and should be imperatively improved.

7.2. Evaluating accuracy

There are no practical ways of quantifying the accuracy of the matchmakers, yet we estimate that the model matchmakers specified in Section 5.3.2 should be used as a reference in terms of accuracy. Indeed, they respond perfectly to our needs and cannot be faulted, by opposition to the profile matchmaker. Here is a case where the profile matchmaker can be wrong (taken from [9]). Let us envision a simple choice process which produces two outputs, say o1 and o2. If a request for a service providing both these outputs were to be matched by simply comparing the Service Profile outputs, the result would be a positive match. In reality however, the process is not capable of providing both these outputs. As a result, we can only quantify the accuracy by saying that the model matchmaker is our reference, and that the profile can – in certain cases – be wrong.

Chapter 8. Conclusion & Future Works

In this report we have described our approach to upgrade the system "An agent-based system for Grid-services provision and selection". The system is based on communicating agents which negotiate services on behalf of providers and requestors of services.

In order to provision a service, i.e. to assign it to a Service Provision Agent (SPA), the provider only has to indicate the URL where the OWL-S description is, using either the GUI or by calling the appropriate Java function of the agent. When a service requestor wants to initiate a search for services, he has to create a synthetic service description representing the requested service in OWL-S and to specify the location of that created service to its Service Selection Agent (SSA), once again via the GUI or via the appropriate function of the agent. Using the synthetic service the SSA sends messages, requesting search to be carried out at some SPAs. When a search request is received at an SPA, the SPA extracts all its advertised services (i.e. the provisioned ones) from its local storage and matches them against the requested one, using a matching algorithm. The best matching services (i.e obtaining a score greater or equal than the score specified by the requester) are returned to the SSA. If a requester wishes to preserve its privacy, he can also choose to send only a fragment of his desiderata to the SPAs and match himself the list of services replied.

This system is implemented as a prototype in Java under the JADE multi-agent platform. It was validated, in regular mode, with services found on the website http://www.mindswap.org, in a full utilization case. This prototype was evaluated in order to assess the time needed by the different parts of the system, and to assess which part uses most of the application time. It showed that when working with regular small ontologies, not numerous in the SPA's knowledge base, the time needed to match two services is about one minute. This appears to be quite slow as in regular cases a matching process involves the matchmaking of more than two services. Moreover, the time taken increases linearly with the size of the knowledge base (i.e. with the number of provided services), and it can take up to more than five minutes for a single match. When running tests, we saw that 72% of this time is taken by the reasoner to infer relationships, and that 25% was also taken by the reasoner to get concepts in the ontologies. We also made the observation that the code could be factorized to reduce this 25% down to a few percents, but that it would not solve the problem of sluggishness. In conclusion, it becomes obvious that the reasoner chosen is not adequate and not efficient enough and needs to be changed imperatively. We also came across the idea that a reasoner specifically designed for our application could be built up, as we do not require lots of advanced functionalities. A personal reasoner should be faster as it would be stripped off from all useless features.

Apart from changing the Pellet reasoner, several features still need to be changed or implemented. For instance, the model matchmaker still has to be implemented fully, easy task once the exact algorithms are known. Then, we did not implement the functionality allowing users to add registries of services directly to the system. The previous system used Globus Virtual Organizations (VO Registry, [15]), but since the system is to be upgraded to WS-RF, the VOs are to be considered obsolete, as are Grid-services. Nevertheless, the latest version of Globus Toolkit (Version 4) still proposes index services which constitute a good equivalent to VOs. We also omitted to let the requester specify a minimal score, and as a consequence a default score is systematically used. Changing this detail is rather trivial (the only need is to pass a score along with the requested service), but involves a few changes in the application. Next, it would be a good idea to investigate other information to provide to SPAs in the case of the "secure" mode. Indeed, the profile classification is not widely used yet and thus does not appear as the most relevant information to use when pre-selecting services. Moreover, despite the security mode provided, it is still crucial to investigate other security aspects of the system before taking it into use. If the future architecture is implemented on top of Jade one can with rather small means enforce user-to-agent authentication as well as message integrity and confidentiality. The permissions granted for each users of the Jade platform can also be specified in a policy file. Except for the agent platform it might be necessary to have some security regarding interacting with the UDDI registry and the invocation of the provisioned Web services.

We hope that the system presented, still very effective in favorable conditions, will be useful for advanced services selection purposes, and will offer great guidance for the future overall project working towards a novel solution for *Agent-Enabled Logic-Based Web Services Selection and Composition* [22].

A. Bibliography

[1] G. Nimar, *"An Agent based system for Grid Services provision and selection",* 2004

[2] J. Rao. *"Semantic Web Service Composition via Logic-based Program Synthesis"*, Department of Computer and Information Science, Norwegian University of Science and Technology, December 10, 2004.

[3] Czajkowski, Ferguson, Foster, Frey, Graham, Maguire, Snelling, Tuecke*, "From Open Grid Services Infrastructure to WSResource Framework: Refactoring & Evolution"*, 2004

[4] Martin, *"OWL-S: Semantic Markup for Web Services"*, 2005

[5] Mithun Sheshagiri and Marie desJardins and Tim Finin, *"A Planner for Composing Services Described in DAML-S"* In Proceedings of the AAMAS Workshop on Web Services and Agent-based Engineering, 2003

[6] Sheila A. McIlraith, Ronald Fadel: *"Planning with complex actions"*, NMR 2002: 356-364

[7] Jaeger, Rojec-Goldmann, Liebetruth, Kurt Geihs *"Ranked Matching for Service Descriptions Using OWL-S"*, KiVS 2005: 91-102

[8] Stefan Tang, *"Matching of Web Service Specifications Using DAML-S Descriptions",* Thesis, 18Mar-2004

[9] Sharad Bansal, José M. Vidal, *"Matchmaking of Web Services Based on the DAMLS Service Model"*, AAMAS'03, July 14–18, 2003, Melbourne, Australia

[10] Paolucci, Kawamura, Paine, Sycara*, "Semantic matching of Web-Services capabilities"*, Int. Semantic Web Conference 2001

[11] Czajkowski, Ferguson, Foster, Frey, Graham, Maguire, Snelling, Tuecke, *"Open Grid Services Infrastructure (OGSI)"*, 2003

[12] Frank Leymann, Dieter Roller, and Satish Thatte, *"Goals of the BPEL4WS Specification"*, Working document submitted to the OASIS Web, August 25, 2003

[13] *http://www.daml.org/services/owl-s/1.1/ProfileHierarchy.html*. Profile-based Class Hierarchies

[14] Terry R. Payne, Massimo Paolucci, and Katia Sycara*, "Advertising and matching daml-s service descriptions"*, In Position Papers for SWWS' 01, pages 76–78, Stanford, USA, July 2001. Stanford University

[15] The Globus Alliance, *http://www.globus.org*.

[16] S. Miles , et al., *"Personalized Grid Service Discovery",* IEE Proc., vol. 150, no. 4, Aug. 2003

[17] D. Kuokka and L.Harrada. *"On using kqml for matchmaking",* In CIKM-95 3rd Conf. on Information and Knowledge Management. AAAI/MIT Press, 1995

[18] R.J. Glushko, J.M. Tenenbaum, and B. Meltzer, *"An XML framework for agent-based e-commerce",* Communications of the ACM, 42(3), March 1999

[19] CDL. Capability Description Language. *http://www.aiai.ed.ac.uk/oplan/cdl*.

[20] The RETE Algorithm. *http://www.cis.temple.edu/~ingargio/cis587/readings/rete.html*

[21] Clement, Hately, von Riegen, Rogers, *"UDDI Spec Technical Committee Draft",* 2004

[22] M Matskin and V Vlassov, *"Agent-Enabled Logic-Based Web Services Selection and Composition",* Research Project Proposal 2004.

[23] F. Baader, W. Nutt. *"Basic Description Logics"* In the Description Logic Handbook

B. Abbreviations

API	Application Programming Interface
BPEL	Business Process Execution Language
CORBA	Common Object Request Broker Architecture
DAML	DARPA Agent Mark Up Language
DAML-S	DARPA Agent Mark Up Language for Services
DCOM	Distributed Component Object Model
DF	Directory Facilitator
DIG	Description logics Interface
DL	Description Logics
EDF	Extended Directory Facilitator
GSH	Grid Service Handle
GSR	Grid Service Reference
GT4	Globus Toolkit 4
GUI	Graphical User Interface
GWSDL	Web Services Description Language for Grid
IDL	Interface Definition Language
IOPE	Inputs-Outputs-Preconditions-Effects
KIF	Knowledge Interchange Format
MAS	Multi-Agent Systems
MDS	Monitoring and Discovery System
OGSA	Open Grid Services Architecture
OGSI	Open Grid Services Infrastructure
OIL	Ontology Inference Layer
OWL	Web Ontology Language
OWL-S	Ontology Web Language for Services
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
RDQL	RDF Data Query Language
RPC	Remote Procedure Call
SDE	Service Data Element
SOAP	Simple Object Access Protocol
SPA	Service Provision Agent
SSA	Service Selection Agent
TF-IDF	Term Frequency-Inverse Document Frequency
tModel	Technical Model
UDDI	Universal Discovery Description and Integration
VO	Virtual Organization
W3C	World Wide Web Consortium
WSDL	Web Services Description Language
WSMO	Web Service Modeling Ontology
WS-RF	WS-Resource Framework
XML	eXtensible Markup Language

C. OWL services used in the application example of Section 6.3.3

- **Request.owl: application example's request file:**

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
  xmlns:owl=        "http://www.w3.org/2002/07/owl#"
  xmlns:rdfs=       "http://www.w3.org/2000/01/rdf-schema#"
  xmlns:rdf=        "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:service=    "http://www.daml.org/services/owl-s/1.0/Service.owl#"
  xmlns:process=    "http://www.daml.org/services/owl-s/1.0/Process.owl#"
  xmlns:profile=    "http://www.daml.org/services/owl-s/1.0/Profile.owl#"
  xmlns:grounding=  "http://www.daml.org/services/owl-s/1.0/Grounding.owl#"
  xml:base=         "file:/D:/Documents%20and%20Settings/Will/Desktop/aze.xml"
>

<owl:Ontology rdf:about="">
        <owl:imports rdf:resource="http://www.aktors.org/ontology/portal"/>
        <owl:imports rdf:resource="http://www.mindswap.org/2004/owl-
s/concepts.owl"/>
</owl:Ontology>

<!-- Service description -->
<service:Service rdf:ID="BNPriceService">
        <service:presents rdf:resource="#BNPriceProfile"/>

        <service:describedBy rdf:resource="#BNPriceProcessModel"/>

        <service:supports rdf:resource="#BNPriceGrounding"/>
</service:Service>

<!-- Profile description -->
<profile:Profile rdf:ID="BNPriceProfile">
        <service:isPresentedBy rdf:resource="#BNPriceService"/>

        <profile:serviceName xml:lang="en">BN Price Check</profile:serviceName>
        <profile:textDescription xml:lang="en">This service returns the price of a book
as advertised in Barnes and Nobles web site given the ISBN
Number.</profile:textDescription>

        <profile:hasInput rdf:resource="#Publication"/>

        <profile:hasOutput rdf:resource="#BookPrice"/>
</profile:Profile>
```

```
<!-- Process Model description -->
<process:ProcessModel rdf:ID="BNPriceProcessModel">
        <service:describes rdf:resource="#BNPriceService"/>
        <process:hasProcess rdf:resource="#BNPriceProcess"/>
</process:ProcessModel>

<process:AtomicProcess rdf:ID="BNPriceProcess">
        <process:hasInput rdf:resource="#Publication"/>
        <process:hasOutput rdf:resource="#BookPrice"/>
</process:AtomicProcess>

<process:Input rdf:ID="Publication">
        <process:parameterType
rdf:resource="http://www.aktors.org/ontology/portal#Publication"/>
        <rdfs:label>Publi</rdfs:label>
</process:Input>

<process:Output rdf:ID="BookPrice">
        <process:parameterType rdf:resource="http://www.mindswap.org/2004/owl-
s/concepts.owl#Price"/>
        <rdfs:label>Book Price</rdfs:label>
</process:Output>

<!-- Grounding description -->
<grounding:WsdlGrounding rdf:ID="BNPriceGrounding">
        <service:supportedBy rdf:resource="#BNPriceService"/>
        <grounding:hasAtomicProcessGrounding
rdf:resource="#BNPriceProcessGrounding"/>
</grounding:WsdlGrounding>

<grounding:WsdlAtomicProcessGrounding rdf:ID="BNPriceProcessGrounding">
        <grounding:owlsProcess rdf:resource="#BNPriceProcess"/>
        <grounding:wsdlDocument>http://www.xmethods.net/sd/2001/BNQuoteService.
wsdl</grounding:wsdlDocument>
        <grounding:wsdlOperation>
                <grounding:WsdlOperationRef>

        <grounding:portType>http://www.xmethods.net/sd/2001/BNQuoteService.wsdl#
BNQuotePortType</grounding:portType>

        <grounding:operation>http://www.xmethods.net/sd/2001/BNQuoteService.wsdl#
getPrice</grounding:operation>
                </grounding:WsdlOperationRef>
        </grounding:wsdlOperation>
        <grounding:wsdlInputMessage>http://www.xmethods.net/sd/2001/BNQuoteServi
ce.wsdl#getPriceRequest</grounding:wsdlInputMessage>
        <grounding:wsdlInputMessageParts rdf:parseType="Collection">
                <grounding:WsdlMessageMap>
                        <grounding:owlsParameter rdf:resource="#Publication"/>

        <grounding:wsdlMessagePart>http://www.xmethods.net/sd/2001/BNQuoteServic
e.wsdl#isbn</grounding:wsdlMessagePart>
                        <grounding:xsltTransformation>
                         <![CDATA[
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:portal="http://www.aktors.org/ontology/portal#">
        <xsl:output method="xml" version="1.0" encoding="UTF-8" indent="yes"/>
        <xsl:template match="/ ">
                <xsl:value-of select="rdf:RDF/portal:Book/portal:has-publication-
reference/portal:Book-Reference/portal:has-ISBN-number"/>
        </xsl:template>
</xsl:stylesheet>
                         ]]>
                        </grounding:xsltTransformation>
                </grounding:WsdlMessageMap>
        </grounding:wsdlInputMessageParts>

        <grounding:wsdlOutputMessage>http://www.xmethods.net/sd/2001/BNQuoteSer
vice.wsdl#getPriceResponse</grounding:wsdlOutputMessage>
        <grounding:wsdlOutputMessageParts rdf:parseType="Collection">
                <grounding:wsdlMessageMap>
                        <grounding:owlsParameter rdf:resource="#BookPrice"/>

        <grounding:wsdlMessagePart>http://www.xmethods.net/sd/2001/BNQuoteServic
e.wsdl#return</grounding:wsdlMessagePart>
                        <grounding:xsltTransformation>
                         <![CDATA[
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
        <xsl:template match="/">
                <xsl:variable name="X1" select="/"/>
                <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:concepts="http://www.mindswap.org/2004/owl-s/concepts.owl#">
                        <concepts:Price>
                                <concepts:currency
rdf:resource="http://www.daml.ecs.soton.ac.uk/ont/currency.owl#USD"/>
                                <concepts:amount>
                                        <xsl:value-of select="$X1"/>
                                </concepts:amount>
                        </concepts:Price>
```

```
                    </rdf:RDF>
          </xsl:template>
</xsl:stylesheet>
                                ]]>
                              </grounding:xsltTransformation>
                    </grounding:wsdlMessageMap>
          </grounding:wsdlOutputMessageParts>
</grounding:WsdlAtomicProcessGrounding>

</rdf:RDF>
```

- **BNPPrice.owl (matching) advertisement's file:**

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
  xmlns:owl=         "http://www.w3.org/2002/07/owl#"
  xmlns:rdfs=        "http://www.w3.org/2000/01/rdf-schema#"
  xmlns:rdf=         "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:service=      "http://www.daml.org/services/owl-s/1.0/Service.owl#"
  xmlns:process=      "http://www.daml.org/services/owl-s/1.0/Process.owl#"
  xmlns:profile=      "http://www.daml.org/services/owl-s/1.0/Profile.owl#"
  xmlns:grounding=    "http://www.daml.org/services/owl-s/1.0/Grounding.owl#"
  xml:base=          "http://zill.free.fr/BNPrice.xml"
>

<owl:Ontology rdf:about="">
          <owl:imports rdf:resource="http://www.aktors.org/ontology/portal"/>
          <owl:imports rdf:resource="http://www.mindswap.org/2004/owl-
s/concepts.owl"/>
</owl:Ontology>

<!-- Service description -->
<service:Service rdf:ID="BNPriceService">
          <service:presents rdf:resource="#BNPriceProfile"/>

          <service:describedBy rdf:resource="#BNPriceProcessModel"/>

          <service:supports rdf:resource="#BNPriceGrounding"/>
</service:Service>

<!-- Profile description -->
<profile:Profile rdf:ID="BNPriceProfile">
          <service:isPresentedBy rdf:resource="#BNPriceService"/>

          <profile:serviceName xml:lang="en">BN Price Check</profile:serviceName>
```

```
        <profile:textDescription xml:lang="en">This service returns the price of a book
as advertised in Barnes and Nobles web site given the ISBN
Number.</profile:textDescription>

        <profile:hasInput rdf:resource="#BookInfo"/>

        <profile:hasOutput rdf:resource="#BookPrice"/>
</profile:Profile>

<!-- Process Model description -->
<process:ProcessModel rdf:ID="BNPriceProcessModel">
        <service:describes rdf:resource="#BNPriceService"/>
        <process:hasProcess rdf:resource="#BNPriceProcess"/>
</process:ProcessModel>

<process:AtomicProcess rdf:ID="BNPriceProcess">
        <process:hasInput rdf:resource="#BookInfo"/>
        <process:hasOutput rdf:resource="#BookPrice"/>
</process:AtomicProcess>

<process:Input rdf:ID="BookInfo">
        <process:parameterType
rdf:resource="http://www.aktors.org/ontology/portal#Book"/>
        <rdfs:label>ISBN Number</rdfs:label>
</process:Input>

<process:Output rdf:ID="BookPrice">
        <process:parameterType rdf:resource="http://www.mindswap.org/2004/owl-
s/concepts.owl#Price"/>
        <rdfs:label>Book Price</rdfs:label>
</process:Output>

<!-- Grounding description -->
<grounding:WsdlGrounding rdf:ID="BNPriceGrounding">
        <service:supportedBy rdf:resource="#BNPriceService"/>
        <grounding:hasAtomicProcessGrounding
rdf:resource="#BNPriceProcessGrounding"/>
</grounding:WsdlGrounding>

<grounding:WsdlAtomicProcessGrounding rdf:ID="BNPriceProcessGrounding">
        <grounding:owlsProcess rdf:resource="#BNPriceProcess"/>
        <grounding:wsdlDocument>http://www.xmethods.net/sd/2001/BNQuoteService.
wsdl</grounding:wsdlDocument>
        <grounding:wsdlOperation>
                <grounding:WsdlOperationRef>
```

```
          <grounding:portType>http://www.xmethods.net/sd/2001/BNQuoteService.wsdl#
BNQuotePortType</grounding:portType>

          <grounding:operation>http://www.xmethods.net/sd/2001/BNQuoteService.wsdl#
getPrice</grounding:operation>
                    </grounding:WsdlOperationRef>
          </grounding:wsdlOperation>

          <grounding:wsdlInputMessage>http://www.xmethods.net/sd/2001/BNQuoteServi
ce.wsdl#getPriceRequest</grounding:wsdlInputMessage>
          <grounding:wsdlInputMessageParts rdf:parseType="Collection">
                    <grounding:WsdlMessageMap>
                              <grounding:owlsParameter rdf:resource="#BookInfo"/>

          <grounding:wsdlMessagePart>http://www.xmethods.net/sd/2001/BNQuoteServic
e.wsdl#isbn</grounding:wsdlMessagePart>
                              <grounding:xsltTransformation>
                               <![CDATA[
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:portal="http://www.aktors.org/ontology/portal#">
          <xsl:output method="xml" version="1.0" encoding="UTF-8" indent="yes"/>
          <xsl:template match="/ ">
                    <xsl:value-of select="rdf:RDF/portal:Book/portal:has-publication-
reference/portal:Book-Reference/portal:has-ISBN-number"/>
          </xsl:template>
</xsl:stylesheet>
                               ]]>
                              </grounding:xsltTransformation>
                    </grounding:WsdlMessageMap>
          </grounding:wsdlInputMessageParts>

          <grounding:wsdlOutputMessage>http://www.xmethods.net/sd/2001/BNQuoteSer
vice.wsdl#getPriceResponse</grounding:wsdlOutputMessage>
          <grounding:wsdlOutputMessageParts rdf:parseType="Collection">
                    <grounding:wsdlMessageMap>
                              <grounding:owlsParameter rdf:resource="#BookPrice"/>

          <grounding:wsdlMessagePart>http://www.xmethods.net/sd/2001/BNQuoteServic
e.wsdl#return</grounding:wsdlMessagePart>
                              <grounding:xsltTransformation>
                               <![CDATA[
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
          <xsl:template match="/">
                    <xsl:variable name="X1" select="/"/>
```

```
                  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:concepts="http://www.mindswap.org/2004/owl-s/concepts.owl#">
                          <concepts:Price>
                                  <concepts:currency
rdf:resource="http://www.daml.ecs.soton.ac.uk/ont/currency.owl#USD"/>
                                  <concepts:amount>
                                          <xsl:value-of select="$X1"/>
                                  </concepts:amount>
                          </concepts:Price>
                  </rdf:RDF>
          </xsl:template>
</xsl:stylesheet>
                           ]]>
                          </grounding:xsltTransformation>
                  </grounding:wsdlMessageMap>
          </grounding:wsdlOutputMessageParts>
</grounding:WsdlAtomicProcessGrounding>

</rdf:RDF>
```

- **http://www.aktors.org/ontology/portal ontology selected excerpts:**

```
<owl:Class rdf:ID="Book">
   <rdfs:subClassOf rdf:resource="#Publication"/>
   <rdfs:subClassOf>
     <owl:Restriction>
       <owl:onProperty rdf:resource="#has-publication-reference"/>
       <owl:allValuesFrom rdf:resource="#Book-Reference"/>
     </owl:Restriction>
   </rdfs:subClassOf>
   <rdfs:subClassOf>
     <owl:Restriction>
       <owl:onProperty rdf:resource="#has-publication-reference"/>
       <owl:minCardinality
rdf:datatype="&xsd;nonNegativeInteger">1</owl:minCardinality>
     </owl:Restriction>
   </rdfs:subClassOf>
   <rdfs:isDefinedBy rdf:resource="&base;"/>
  </owl:Class>

  <owl:Class rdf:ID="Publication">
     <rdfs:comment>A publication is something which has one or more publication
references. A publication can be both an article in a journal or a journal itself. The
distinction between publication and publication-reference makes it possible to distinguish
```

```
between multiple occurrences of the sam publication, for instance in different
media</rdfs:comment>
      <rdfs:subClassOf rdf:resource="#Information-Bearing-Object"/>
      <rdfs:subClassOf>
        <owl:Restriction>
          <owl:onProperty rdf:resource="#has-publication-reference"/>
          <owl:allValuesFrom rdf:resource="#Publication-Reference"/>
        </owl:Restriction>
      </rdfs:subClassOf>
      <rdfs:subClassOf>
        <owl:Restriction>
          <owl:onProperty rdf:resource="#has-publication-reference"/>
          <owl:minCardinality
rdf:datatype="&xsd;nonNegativeInteger">1</owl:minCardinality>
        </owl:Restriction>
      </rdfs:subClassOf>
      <rdfs:subClassOf>
        <owl:Restriction>
          <owl:onProperty rdf:resource="#cites-publication-reference"/>
          <owl:allValuesFrom rdf:resource="#Publication-Reference"/>
        </owl:Restriction>
      </rdfs:subClassOf>
      <rdfs:isDefinedBy rdf:resource="&base;"/>
  </owl:Class>
```