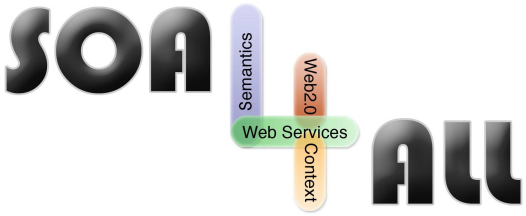

Project Number:	**215219**
Project Acronym:	**SOA4All**
Project Title:	**Service Oriented Architectures for All**
Instrument:	**Integrated Project**
Thematic Priority:	**Information and Communication Technologies**

D5.1.1 State of the Art Report On Service Description and Existing Discovery Techniques

Activity N:	Activity 2- Research and Development Activities	
Work Package:	WP 5 – Service Location	
Due Date:		M6 and M12
Submission Date:		15/08/2008 Resubmission: 11/03/2009
Start Date of Project:		01/03/2008
Duration of Project:		36 Months
Organisation Responsible of Deliverable:		UIBK
Revision:		1.0
Author(s):	Ioan Toma Nathalie Steinmetz Jean Pierre Lorre	STI Innsbruck STI Innsbruck EBM
Reviewer(s):	Maria Maleshkova Marc Richardson	UKARL BT

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)		
Dissemination Level		
PU	**Public**	**X**

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	11 June 2008	First draft and table of contents	Ioan Toma (STI Innsbruck), Nathalie Steinmetz (STI Innsbruck)
0.2	16 June 2008	Added initial version of the introduction	Ioan Toma (STI Innsbruck)
0.3	20 June 2008	Added Semantic Web services approaches	Ioan Toma, Dumitru Roman (STI Innsbruck), Holger Lausen (seekda), Jacek Kopecky (STI Innsbruck)
0.4	09 July 2008	Added initial content in Registries section	Ioan Toma (STI Innsbruck), Jean Pierre Lorre (ebm)
0.5	22 July 2008	Added section about WSDL	Ioan Toma, Jacek Kopecky (STI Innsbruck)
0.6	23 July 2008	Finished introduction and conclusions	Ioan Toma (STI Innsbruck)
0.7	24 July 2008	Added section about SAWSDL	Ioan Toma (STI Innsbruck)
0.8	26 July 2008	Added section about RESTful services	Ioan Toma (STI Innsbruck)
0.9	28 July 2008	Added section about Logic based discovery	Ioan Toma (STI Innsbruck)
0.10	1 August 2008	Added section about Portals, standard search engines, crawling and outlook	Nathalie Steinmetz (STI Innsbruck)
1.0	14 August 2008	Incorporate reviewers comments; Final version	Ioan Toma (STI Innsbruck)
Final	March 9th 2009	Overall formant and quality revision	Malena Donato (ATOS)

Table of Contents

EXECUTIVE SUMMARY 6

1 INTRODUCTION 7

2 STATE OF THE ART ON SERVICE DESCRIPTION 9

2.1 WSDL 9
2.1.1 Web Service Interface 9
2.1.2 Web Service Endpoints, Bindings 10
2.1.3 WSDL Documents 11
2.1.4 Note on the differences between WSDL 2.0 and WSDL 1.1 11
2.2 RESTFUL WEB SERVICES 12
2.2.1 REST principles 12
2.2.2 RESTful service descriptions 12
2.3 SEMANTIC SERVICES FRAMEWORKS 14
2.3.1 WSMO Approach 15
2.3.2 OWL-S 19
2.3.3 The SWSF approach 23
2.3.4 IRI-III 26
2.3.5 SAWSDL 27

3 STATE OF THE ART ON SERVICE DISCOVERY 31

3.1 REGISTRIES 31
3.1.1 The Registry/Repository concepts 31
3.1.2 The registry non competing standards 31
3.1.3 Today's registry/repository solutions 33
3.1.4 Dragon: an emerging Open Source Registry/Repository 34
3.2 PORTALS 37
3.3 STANDARD SEARCH ENGINES 39
3.4 LOGIC BASED APPROACHES 39
3.4.1 WSMO discovery 40
3.4.2 DAML-S/OWL-S discovery approaches 41

4 STATE OF THE ART ON CRAWLING 43

4.1 WEB CRAWLERS 43
4.2 BASIC CRAWL STEPS 44
4.3 CRAWLING STRATEGIES 45
4.4 CRAWLER TYPES 45
4.4.1 Broad Crawling 45
4.4.2 Focused Crawling 46
4.4.3 Continuous Crawling 47
4.5 WEB SERVICE CRAWLING TECHNIQUES 47

5 OUTLOOK OF SERVICE CRAWLING TECHNIQUES 50

5.1 CRAWLED SERVICE DATA 50
5.2 OUTLOOK OF SERVICE CRAWLING TECHNIQUES 50
5.3 OUTLOOK OF SOA4ALL SERVICE DISCOVERY 51

6 CONCLUSIONS 53

7 REFERENCES 54

List of Figures

Figure 1 WSMO top level elements 16
Figure 2 OWL-S top level elements 20
Figure 3 Layered structure of SWSL-Rules 25
Figure 4 Dragon Architecture 35
Figure 5 Dragon federated registry 36
Figure 6 seekda Web Service Search Engine 39
Figure 7 WSMO Discovery model 40
Figure 1 Example Figure ..... **Error! Bookmark not defined.**

List of Listings

Listing 1 Illustrative example of a WSDL interface *13*
Listing 2 Example of a WSDL binding and service *14*
Listing 3 WADL Example *17*
Listing 4 WSMO Ontology class *20*
Listing 5 WSMO Web service class *21*
Listing 6 WSMO Goal class *21*
Listing 7 WSMO Mediator class *22*
Listing 8 SAWSDL annotation example *35*

Glossary of Acronyms

Acronym	Definition
D	Deliverable
EC	European Commission
WP	Work Package
WSDL	Web Services Description Language
UDDI	Universal Description Discovery and Integration
WSMO	Web Service Modeling Ontology
WSML	Web Service Modeling Language
WSMX	Web Service Modeling eXecution environment
SOA	Service Oriented Architecture
SWS	Semantic Web Services
SOAP	Simple Object Access Protocol
WADL	Web Application Description Language
JSON	JavaScript Object Notion
OWL-S	Web Ontology Language for Services
SWSF	Semantic Web Services Framework
IRS-3	Internet Reasoning Service
SAWSDL	Semantic Annotations for WSDL
QoS	Quality of Service
SLA	Service Level Agreement

Executive Summary

This deliverable analyzes existing approaches for service descriptions, service discovery and service crawling. The aim is to provide an overview of existing service description approaches, second to review existing techniques for service crawling and discovery that are using the service description approaches reviewed in the first step. First an overview of the state of the art on service description is provided. Three groups of service description approaches are investigated, namely (1) WS-* standards for Web service description, focusing on WSDL services, (2) RESTful Web services that are closely following Web principles applied to service descriptions. A special interest is given to Web API documentations and also to Web2.0 sources relevant for service descriptions such as tagging, service characterizations and ratings in blogs and (3) Semantic Web services approaches that are bringing Semantic Web technologies to annotated Web services. Second, an overview of state of the art in service discovery is provided including analysis of registry-based, portal-based and logic based approaches. Third, an analysis of existing techniques for crawling of web resources, the emphasis being mostly on focus crawling techniques is provided. Finally, an outlook for crawling techniques; containing initial ideas for SOA4All crawling techniques as well as a short description of the service description data already collected using the seekda crawling infrastructure is provided.

1 Introduction

The current Web is changing from a static collection of web pages to a dynamic collection of services. More and more applications are published and consumed each day on the Web as services. This trend is visible not only on a large open scale platform such as the Web but also in closed, business settings such as companies' platforms. Big industrial players from IT and communications are currently in the process of changing their infrastructure allowing business partners to access their functionalities as services. The service-oriented perspective promoted by Service Oriented Architectures (SOA) pushes the notion of service as the central notion, abstracting from the underlying implementation and hardware. However, the paradigm shift introduces a set of new challenges such as how to organize, find, rank and select services in a scalable fashion given the continuously growing number of services. Possible solutions for all these new challenges will strongly depend on one important aspect, namely how services are modelled and described. In the context of this deliverable we survey the most important approaches from the large landscape of service description approaches.

Most services available today on the Web or in closed industrial settings are described using WS-* standards[1]. Given this fact, we provide first on overview of the Web Services Description Language (WSDL), the main WS-* standard for Web service description. In WSDL services are described in terms of the messages, operations and ports. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. The design principles underling WSDL are very much programming oriented being supportive in the context of developing distributed internet applications. However as pointed out in [29], Web services are not actually following Web principles, the usage of Web communication medium being the only thing Web service have in common with the Web. WSDL based Web services are tightly coupling the applications they integrate. The Web, on the other hand is based on opposite principles, information being published in a persistent and widely accessible manner. Recently, following closely the Web principles, REST architectural style is being applied in the development and publishing of services, called RESTful services [43]. In this approach services are viewed as resource and can be identified by their URLs. Service clients that want to use these resources access a particular representation by transferring application content using a small globally defined set of remote methods that describe the action to be performed on the resource. Although the number of RESTful services is increasing, there are currently no description languages for this kind of services. Current descriptions of RESTful services are mainly provided as plain texts. Both WS-* based and RESTful services descriptions are not formal descriptions that could be 'automatically' process by machines. With the emergence of the Semantic Web [28] that promotes the vision of machine processable description for Web resources, a new type of service descriptions that address the problem of automation were proposed, namely Semantic Web services. Various Semantic Web services approaches have been developed both in Europe and in USA. Their overall goal is to provide comprehensive service modelling frameworks that will enable a certain degree of automation for service related tasks such as discovery, composition, selection, etc.

The growing number of services introduces a new set of challenges one of the most important one being how to discover the most relevant services given a user request. This overall task, known as service discovery, has been addressed in the past year in many

[1] According to seekda.com the number of WSDL services available online on July 27, 2008 are 27.574

approaches. Registry based solutions (e.g. UDDI [41]), portals and service search engines have been proposed mainly for WSDL descriptions. Logic-based approaches that employ reasoning support to determine the degree of match between services and user requests have been proposed mainly for Semantic Web services descriptions.

In this deliverable we analyze existing approaches for service descriptions, service discovery and service crawling. The aim is to provide an overview of existing service description approaches, second to review existing techniques for service crawling and discovery that are using the service description approaches reviewed in the first step. The overall analysis provided in this deliverable would serve as input for upcoming deliverables in WP5, namely D5.2.1 “Service Crawling Techniques and Report on Available Information on The Current Web” that aims to develop crawling techniques for service descriptions indentified in the current deliverable and D5.3.1 “On the Creation of Rich Service Description and Specification Of Reasoning Usage In Service Discovery” that uses will focus on providing discovery techniques using the reasoning techniques developed in WP3. The new SOA4All discovery techniques that will be developed in D5.3.1 will take into account the weak and strong points of existing discovery techniques analyzed in the current deliverable.

The deliverable is organized as follows. Section 2 contains an analysis of state of the art approaches on service description. Three groups of service description approaches are investigated, namely (1) WS-* standards for Web service description, focusing on WSDL services, (2) RESTful Web services that are closely following Web principles applied to service descriptions. A special interest is given to Web API documentations and also to Web2.0 sources relevant for service descriptions such as tagging, service characterizations and ratings in blogs and (3) Semantic Web services approaches that are bringing Semantic Web technologies to annotated Web services. Section 3 surveys the most significant approaches for service discovery, including registry-based, portal-based and logic based approaches. Section 4 contains an analysis of existing techniques for crawling of web resources, the emphasis being mostly on focus crawling techniques. Section 5 provides an outlook for crawling techniques; containing initial ideas for SOA4All crawling techniques as well as a short description of the service description data already collected using the seekda crawling infrastructure. Furthermore this section provides an outlook for the overall discovery approach in SOA4All. Finally Section 6 concludes the deliverable.

2 State of the art on service description

This section surveys existing approaches for service description. Section 2.1.1 provides an overview of the standard language used to describe most of the available services on the internet, namely Web Services Description Language (WSDL). Section 2.2 discusses service descriptions in the context of RESTful services that are closely following Web principles. Finally, Section 2.3 surveys the most relevant Semantic Web services frameworks for service descriptions.

2.1 WSDL

The Web Services Description Language is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. WSDL describes Web services in two levels — an XML-based reusable abstract interface and the concrete details regarding how and where this interface can be accessed. All descriptions in WSDL are centered on the Web service and all terminology follows the service's point of view, for example input messages are messages coming into the service from the network and output messages are messages generated by the service and sent to the network. The rest of this section provides an overview of the WSDL language, describing various aspects of WSDL descriptions, based on WSDL version 2.0, namely about abstract Web service interfaces (Section 2.1.1), binding them to concrete wire protocols and endpoints (Section 2.1.2) and finally about the overall organization of WSDL documents (Section 2.1.3). Finally Section 2.1.4 details the relevant differences in the older version, WSDL 1.1. The following subsections are common with sections available in WP1 deliverables (D1.2.1 and D1.1.1).

2.1.1 Web Service Interface

On the abstract level, a Web service interface is described in terms of data schemas and simple message exchanges. In particular, WSDL models *interfaces* as sets of related *operations*, each consisting of one or more messages. For example an *interface* of a ticket booking Web service can have operations for querying for a trip price and for the actual ticket booking:

```
01  <interface name="BookTicketInterface">
02    <operation name="queryPrice" pattern="http://www.w3.org/ns/wsdl/in-out">
03      <input element="tns:TripSpecification"/>
04      <output element="tns:PriceQuote"/>
05      <outfault ref="tns:TripNotPossible"/>
06    </operation>
07    <operation name="bookTicket" pattern="http://www.w3.org/ns/wsdl/in-out">
08      <input element="tns:BookingRequest"/>
09      <output element="tns:Reservation"/>
10      <outfault ref="tns:CreditCardNotValid"/>
11      <outfault ref="tns:TripNotPossible"/>
12    </operation>
13    <fault name="TripNotPossible" element="tns:TripFailureDetail" />
14    <fault name="CreditCardNotValid" element="tns:CreditCardInvalidityDetail" />
15  </interface>
```

Listing 1 Illustrative example of a WSDL interface

In WSDL, an operation represents a simple exchange of messages that follows a specific message exchange pattern (MEP). The simplest of MEPs, "In-Only", allows a single

application message to be sent to the service, and "Out-Only" symmetrically allows a single message to be sent by the service to its client. Somewhat more useful is the "Robust-In-Only" MEP, that also allows a single incoming application message but in case there is a problem with it, the service may reply with a fault message. Perhaps the most common MEP is "In-Out", which allows an incoming application message followed either by an outgoing application message or an outgoing fault message. Finally, an interesting MEP commonly used in messaging systems is "In-Optional-Out" where a single incoming application message may (but need not) be followed either by a fault outgoing message or by a normal outgoing message, which in turn may be followed by an incoming fault message (i.e. the client may indicate to the service a problem with its reply).

Particular messages (incoming, outgoing) in an operation, reference XML Schema element declarations to describe the content. Fault messages, however, reference faults defined on the interface level (see above the <outfault> element), with the intention that semantically equivalent faults can be shared by different operations. Additionally, there may be multiple fault references for the same MEP fault message — in effect WSDL faults are typed and one operation can declare that it can result in any number of alternative faults (apart from the single success message).

2.1.2 Web Service Endpoints, Bindings

In order to communicate with a Web service described by an abstract interface, a client must know how the XML messages are serialized on the network and where exactly they should be sent. In WSDL, on-the-wire message serialization is described in a *binding* and then a *service* construct enumerates a number of concrete *endpoint* addresses.

A binding generally follows the structure of an interface and specifies the necessary serialization details. The WSDL specification contains two predefined binding specifications, one for SOAP (over HTTP) and one for plain HTTP. These bindings specify how an abstract XML message is embedded inside a SOAP message envelope or in an HTTP message, and how the message exchange patterns are realized in SOAP or HTTP. Due to extensive use of defaults, simple bindings only need to specify very few parameters, as in the example below. A notable exception to defaulting in binding are faults, as in SOAP every fault must have a so called fault code with two main options, Sender or Receiver, indicating who seems to have a problem. There is no reasonable default possible for the fault code.

Bindings seldom need to contain details specific to a single actual physical service, therefore in many cases they can be as reusable as interfaces, and equivalent services by different providers only need to specify the different endpoints, sharing the interface and binding descriptions.

The *service* construct in WSDL represents a single physical Web service that implements a single interface. The Web service can be accessible at multiple endpoints, each potentially with a different binding, for example one endpoint using an optimized messaging protocol with no data encryption for the secure environment of an intranet and a second endpoint using SOAP over HTTPS for access from the Internet.

```
01 <binding
02         name="SOAPTicketBooking"
03         interface="tns:BookTicketInterface"
04         type="http://www.w3.org/ns/wsdl/soap"
05         wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/" >
```

```
06     <fault ref="TripNotPossible" wsoap:code="soap:Receiver"/>
07     <fault ref="CreditCardNotValid" wsoap:code="soap:Sender"/>
08  </binding>
09
10  <service
11          name="STI2TicketBooking"
12          interface="tns:BookTicketInterface">
13     <endpoint
14              name="normal"
15              binding="tns:SOAPTicketBooking"
16              address="http://sti2.example.org/tickets" />
17  </service>
```

Listing 2 Example of a WSDL binding and service

2.1.3 WSDL Documents

Apart from the interfaces, bindings and services described above, WSDL documents can contain further elements, enclosed in the root <description> element.

In order to facilitate true reuse of interfaces or bindings, WSDL documents can be modularized by using include and import mechanisms. When a WSDL document is parsed, imports and includes are resolved so the resulting model is not aware that some pieces may have come from different actual files.

As a container for data type information, WSDL documents have a section called <types>. Actual schemas can either be embedded directly in this section or referred to using the appropriate import statements. For example external XML Schema documents can be imported by putting the <xs:import> element directly in the <types> section. By default, WSDL uses XML Schema to describe data, but WSDL extensibility allows other data type systems to be used instead.

Finally, every element in a WSDL document can be annotated with documentation elements or it can contain extensibility elements or attributes.

2.1.4 Note on the differences between WSDL 2.0 and WSDL 1.1

This note details the differences between WSDL version 1.1 [11], a specification authored by several companies and submitted to the W3C as the basis for standardization work, and WSDL version 2, the resulting draft standard. While this document uses the cleaner version 2 of WSDL, actual deployment prefers WSDL 1.1 because WSDL 2 is not yet finished and implemented. This note aims to limit any confusion stemming from the situation that some readers may only be familiar with WSDL 1.1.

The first notable difference is that several constructs from WSDL 1.1 were renamed in WSDL 2. In particular, *portType* in WSDL 1.1 is known as *interface* in WSDL 2 and *port* in WSDL 1.1 (occurring within a *service*) is now known as *endpoint*. Also, the WSDL document root element is called *definitions* in WSDL 1.1 and *description* in WSDL 2. Importantly, the intention of all these renamed constructs is unchanged between the two WSDL versions.

A larger difference is that while WSDL 2 uses XML Schema element declarations to describe messages, WSDL 1.1 had a special construct, *message*, that contained potentially several

parts, each referencing a single XML Schema element or type declaration. However, the use of multiple *parts* in a single *message* is usually translatable to a single element containing a sequence of elements (one for each *part*), making the different approaches in WSDL 1.1 and in WSDL 2 equivalent for all practical purposes.

2.2 RESTful Web services

Following closely Web principles, the REST architectural style has been applied in the development and publishing of services called RESTful services [43]. In this approach services are viewed as resource and can be identified by their URLs. Service clients that want to use these resources access a particular representation by transferring application content using a small globally defined set of remote methods that describe the action to be performed on the resource. In the rest of this section we survey briefly the REST principles in Section 2.2.1 and RESTful service descriptions in Section 2.2.2

2.2.1 REST principles

Representation state transfer or shortly REST is an architectural style introduced by Roy Fielding in his dissertation [32]. It refers to a collection of network architecture principles which outline how resources are defined and addressed. According to [33] the REST architectural style is based on four principles:

- *Resource identification through URI*. Resources are identified by URIs, which provide a global addressing space for resource and service.
- *Uniform interface*. There is a uniform interface for or accessing resources, which consists of URIs, methods, status codes, headers, and content distinguished by MIME type. The methods used are compared with the database technology methods create, read, update, delete operations(CRUD). They are: PUT, GET, POST, and DELETE. PUT creates a new resource, which can be then deleted using DELETE. GET retrieves the current state of a resource in some representation. POST transfers a new state onto a resource.
- *Self-descriptive messages*. There is a decoupling between resources and their representation. Resources content can be accessed in a variety of formats (e.g., HTML, XML, plain text, PDF, JPEG, etc.).
- *Stateful interactions through hyperlinks*. Every interaction with a resource is stateless, i.e., request messages are self-contained. Stateful interactions are based on the concept of explicit state transfer. Several techniques exist to exchange state, e.g., URI rewriting, cookies, and hidden form fields. State can be embedded in response messages to point to valid future states of the interaction.

An interesting comparison of the architectural styles used in systems based on WS-* and RESTful services is presented in [33]. The main characteristics of REST versus RPC which translate in differences between RESTful services versus WS-* based services is that: (1) commands are defined in simple terms: resources to be retrieved, stored / get, set—difficult to do many joins and (2) Nouns are key aspects, REST being about exchanging resources and concepts.

2.2.2 RESTful service descriptions

Nowadays the number of RESTful service available on the Web is increasing. Some of the most popular resource-oriented services include: services that expose the Atom Publishing Protocol, Amozon S3 (Simple Storage Service)[2], Yahoo's web service[3], read only services (e.g. search engines). A RESTful service can be defined as being a set of Web resources,

[2] http://aws.amazon.com/s3

[3] http://developer.yahoo.com

interlinked, being data-centric and machine oriented.

2.2.2.1 The Web Application Description Language (WADL)

Although the number of RESTful services is increasing, there are currently no standard description languages for this kind of services. Most current descriptions of RESTful services are mainly provided as plain texts. One timid description language, with a poor adaption so far is the Web Application Description Language WADL. WADL [38] is an XML language designed to provide a machine processable protocol description format for use with HTTP-based Web applications, especially those using XML to communicate.

A WADL document is defined using the following elements:

- ***Application*** is a top level element that contains the overall description of the service. It might contain grammars, resources, method, representation and fault elements.
- ***Grammars*** element acts as a container for definitions of any XML structures exchanged during execution of the protocol described by the WADL document. Using the sub-element ***include*** one or more structures can be included
- ***Resources*** element that acts as a container for the resources provided by the application
- ***Resource*** describes a single resource provided by the Web application. Each resource is identified by an URI and the resources parent element. It can contain the following sub-elements: ***path_variable*** that is used to parameterize the identifiers of the parent resource, zero or more ***method*** elements and zero or more ***resource*** elements.
- ***Method*** element describes the input to and output from an HTTP protocol method that may be applied to a resource. A method element might have two child elements: a ***request*** element that describes the input to be included when applying an HTTP method to a resource and a ***response*** element that describes the output that results from performing an HTTP method on a resource. A request element might contain ***query variable*** elements
- ***Representation*** element describes a representation of a resource's state and can either be declared globally as a child of the ***application*** element, embedded locally as a child of a request or ***response*** element, or referenced externally.
- ***Fault*** element is similar to a ***representation*** element in structure but differs in that it denotes an error condition.

Listing 3 contains the WADL description of Yahoo search APIs[4]

```
1 <?xml version="1.0"?>
2 <application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3 xsi:schemaLocation="http://research.sun.com/wadl wadl.xsd"
4 xmlns:tns="urn:yahoo:yn"
5 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
6 xmlns:yn="urn:yahoo:yn"
7 xmlns:ya="urn:yahoo:api"
8 xmlns="http://research.sun.com/wadl">
9 <grammars>
10 <include
11 href="NewsSearchResponse.xsd"/>
12 <include
13 href="http://api.search.yahoo.com/Api/V1/error.xsd"/>
```

[4] http://developer.yahoo.net/.

```
</grammars>

<resources base="http://api.search.yahoo.com/NewsSearchService/V1/">
<resource uri="newsSearch">
<method href="#search"/>
</resource>
</resources>

<method name="GET" id="search">
<request>
<query_variable name="appid" type="xsd:string" required="true"/>
<query_variable name="query" type="xsd:string" required="true"/>
<query_variable name="type" type="xsd:string"/>
<query_variable name="results" type="xsd:int"/>
<query_variable name="start" type="xsd:int"/>
<query_variable name="sort" type="xsd:string"/>
<query_variable name="language" type="xsd:string"/>
</request>
<response>
<representation mediaType="application/xml" element="yn:ResultSet"/>
<fault id="SearchError" status="400" mediaType="application/xml"
element="ya:Error"/>
</response>
</method>
</application>
```

Listing 3 WADL Example

2.2.2.2 Communication with RESTful services

The communication with a RESTful service can be done based on XML documents that are received or returned by such services. There is a growing number of services that are returning simple data structures (numbers, arrays, etc.) that are serialized using JSON. JSON[5] which stands for JavaScript Object Notion is a lightweight data-interchange format, easy for humans to read and for machines to generate and read. The language is built on two structures: (1) a collection of name/value pairs which corresponds in other languages to an object, record, struct, hash table and (2) an ordered list of values, which in most languages correspond to an array, list, vector or sequence.

2.3 Semantic Services frameworks

The research aim of making Web content more machine processable, also known as Semantic Web [28], has been applied in the past years in the context of Web services usage giving birth to a new research area know as Semantic Web services. Semantic Web services augment existing service descriptions, usually described using WSDL, by providing formal, machine processable representation of what a service can do (functionality), how other services or clients can interact with a service in order to consume its functionality (behaviour) and what conditions over the first two types of descriptions (non-functional properties). Formal, machine processable descriptions of services will support the automation of tasks, such as Web service discovery, composition and execution. In this context, this section gives an overview of existing approaches to Semantic Web services, including WSMO approach (Section 2.3.1), OWL-S (Section 2.3.2), the SWSF approach (Section 2.3.3) and IRS-III

[5] http://www.json.org/

(Section 2.3.4).

2.3.1 WSMO Approach

The major initiative in the area of Semantic Web services initiated in Europe is the WSMO initiative. In this section we provide a general overview of the WSMO approach including: The Web Service Modelling Ontology (WSMO) – a conceptual model for Semantic Web services, the Web Service Modelling Language (WSML) – a language providing a formal syntax and semantics for WSMO, and the Web Service Modelling Execution Environment (WSMX) – an execution environment for WSMO descriptions formalized in WSML.

2.3.1.1 The Web Service Modelling Ontology

WSMO [14] provides ontological specifications for the core elements of Semantic Web services. In fact, Semantic Web services aim at an integrated technology for the next generation of the Web by combining Semantic Web technologies and Web services, thereby turning the Internet from an information repository for human consumption into a world-wide system for distributed Web computing. Therefore, appropriate frameworks for Semantic Web services need to integrate the basic Web design principles, those defined for the Semantic Web, as well as design principles for distributed, service-orientated computing of the Web. WSMO is, therefore, based on the following design principles:

- Web Compliance: WSMO inherits the concept of Universal Resource Identifier (URI) for unique identification of resources as the essential design principle of the Word-Wide Web. Moreover, WSMO adopts the concept of Namespaces for denoting consistent information spaces, supports XML and other W3C Web technology recommendations, as well as the decentralization of resources.
- Ontology Based: Ontologies are used as the data model throughout WSMO, meaning that all resource descriptions as well as all data interchanged during service usage are based on ontologies. Ontologies are a widely accepted state-of-the-art knowledge representation, and have thus been identified as the central enabling technology for the Semantic Web. The extensive usage of ontologies allows semantically enhanced information processing as well as support for interoperability; WSMO also supports the ontology languages defined for the Semantic Web.
- Strict Decoupling: Decoupling denotes that WSMO resources are defined in isolation, meaning that each resource is specified independently without regard to possible usage or interactions with other resources. This complies with the open and distributed nature of the Web.
- Centrality of Mediation: As a complementary design principle to strict decoupling, mediation addresses the handling of heterogeneities that naturally arise in open environments. Heterogeneity can occur in terms of data, underlying ontology, protocol, or process. WSMO recognizes the importance of mediation for the successful deployment of Web services by making mediation a first class component of the framework.
- Ontological Role Separation: Users, or more generally clients, exist in specific contexts which will not be the same as for available Web services. For example, a user may wish to book a holiday according to preferences for weather, culture, and childcare, whereas Web services will typically cover airline travel and hotel availability. The underlying epistemology of WSMO differentiates between the desires of users or clients and available services.
- Description versus Implementation: WSMO differentiates between the descriptions of Semantic Web services elements (description) and executable technologies (implementation). While the former requires a concise and sound description framework based on appropriate formalisms in order to provide a concise for semantic descriptions, the latter is concerned with the support of existing and emerging execution technologies for the Semantic Web and Web services. WSMO

aims at providing an appropriate ontological description model, and to be complaint with existing and emerging technologies.

- Execution Semantics: In order to verify the WSMO specification, the formal execution semantics of reference implementations like WSMX as well as other WSMO-enabled systems provide the technical realization of WSMO. This principle serves as a means to precisely define the functionality and behavior of the systems that are WSMO compliant.
- Service versus Web service: A Web service is a computational entity which is able to achieve a user goal by invocation. A service, in contrast, is the actual value provided by this invocation ([19]). WSMO provides means to describe Web services that provide access (searching, buying, etc.) to services. WSMO is designed as a means to describe the former and not to replace the functionality of the latter.

The rest of this section briefly outlines the conceptual model of WSMO. The elements of the WSMO ontology are defined in a meta-meta-model language based on the Meta Object Facility (MOF) [13]. MOF defines an abstract language and framework for specifying, constructing, and managing technology neutral meta-models. The four WSMO top-level elements, namely Ontologies, Web service, Goals and Mediators are described below, a MOF representation for each of these elements being provided as well.

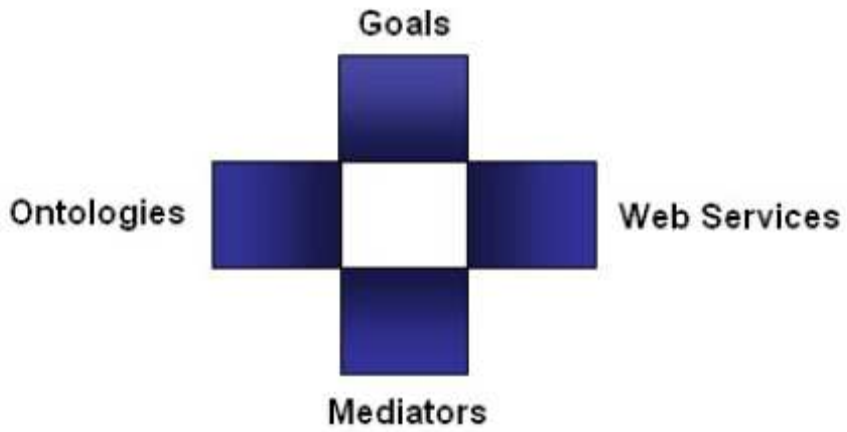

Figure 1 WSMO top level elements

In order to allow complete item descriptions, every WSMO element is described by annotations. These are based on the Dublin Core (DC) Metadata Set [12] for generic information item descriptions.

Ontologies:
Ontologies provide the formal semantics for the terminology used within all other WSMO components. Using MOF, we define an ontology as described in the listing below:

```
Class ontology
 hasAnnotation type annotation
 importsOntology type ontology
 usesMediator type ooMediator
 hasConcept type concept
 hasRelation type relation
 hasFunction type function
 hasInstance type instance
```

```
hasAxiom type axiom
```

Listing 4 WSMO Ontology class

A set of annotations are available for characterizing ontologies; they usually include the DC Metadata elements. Imported ontologies allow a modular approach for ontology design and can be used as long as no conflicts need to be resolved between the ontologies. When importing ontologies in realistic scenarios, some steps for aligning, merging, and transforming imported ontologies in order to resolve ontology mismatches are needed. For this reason ontology mediators are used (OO Mediators). Concepts constitute the basic elements of the agreed terminology for some problem domain. Relations are used in order to model interdependencies between several concepts (respectively instances of these concepts); functions are special relations, with a unary range and a n-ary domain (parameters inherited from relation), where the range value is functionally dependent on the domain values, and instances are either defined explicitly or by a link to an instance store, that is, an external storage of instances and their values

Web services:
WSMO provides service descriptions for describing services that are requested by service requesters, provided by service providers, and agreed between service providers and requesters. In the listing below, the common elements of these descriptions are presented.

```
Class webService
 hasAnnotation type annotation
 importsOntology type ontology
 usesMediator type {ooMediator, wwMediator}
 hasNonFunctionalProperties type nonFunctionalProperty
 hasCapability type capability multiplicity = singlevalued
 hasInterface type interface
```

Listing 5 WSMO Web service class

Within the service class the annotations and imported ontologies attributes play a role that is similar to that found in the ontology class. Non-functional properties attribute was added, being used mainly to describe quality of service properties. An extra type of mediator (WW Mediator) is also included, in order to deal with protocol and process-related mismatches between Web services.
The final two attributes define the two core WSMO notions for semantically describing Web services: a capability which is a functional description of a Web Service, describing constraints on the input and output of a service through the notions of preconditions, assumptions, postconditions, and effects; and Web service interfaces which specify how the service behaves in order to achieve its functionality. A service interface consists of a choreography which describes the interface for the client-service interaction required for service consumption, and an orchestration which describes how the functionality of a Web Service is achieved by aggregating other Web services.

Goals:
A goal specifies the objectives that a client may have when consulting a Web Service, describing aspects related to user desires with respect to the requested functionality and behavior. Ontologies are used as the semantically defined terminology for goal specification. Goals model the user view in the Web Service usage process and therefore are a separate top level entity in WSMO.

```
Class goal
 hasAnnotation type annotation
 importsOntology type ontology
 usesMediator type {ooMediator, wwMediator}
 hasNonFunctionalProperties type nonFunctionalProperty
 requestsCapability type capability multiplicity = singlevalued
 requestsInterface type interface
```

Listing 6 WSMO Goal class

As presented in listing above, the requested capability in the definition of a goal represents the functionality of the services the user would like to have, and the requested interface represents the interface of the service the user would like to have and interact with.

Mediators:
The concept of Mediation in WSMO addresses the handling of heterogeneities occurring between elements that shall interoperate by resolving mismatches between different used terminologies (data level), on communicative behavior between services (protocol level), and on the business process level. A WSMO Mediator connects the WSMO elements in a loosely-coupled manner, and provides mediation facilities for resolving mismatches that might arise in the process of connecting different elements defined by WSMO. The description elements of a WSMO Mediator are its source and target elements, and the mediation service for resolving mismatches, as shown in the listing below.

```
Class mediator
 hasAnnotation type annotation
 importsOntology type ontology
 hasNonFunctionalProperties type nonFunctionalProperty
 hasSource type {ontology, goal, webService, mediator}
 hasTarget type {ontology, goal, webService, mediator}
 hasMediationService type {goal, webService, wwMediator}
```

Listing 7 WSMO Mediator class

WSMO defines different types of mediators for connecting the distinct WSMO elements: OO Mediators connect and mediate heterogeneous ontologies, GG Mediators connect Goals, WG Mediators link Web services to Goals, and WW Mediators connects interoperating Web services resolving mismatches between them.

2.3.1.2 The Web Service Modelling Language (WSML)

The Web Service Modeling Language [15] is a formal language for describing ontologies, goals, Web services and mediators. WSML follows the WSMO conceptual model being based on a set of well-known logical formalisms including: Description Logics [16], Logic Programming [18], F-Logic [17] and First Order Logic. These formalisms are taken as starting points for the development of a number of WSML language variants. WSML has a set of five variants: WSML-Core, WSML-Flight, WSML-Rule, WSML-DL and WSML Full. WSML-Core is based on the intersection of Description Logics and Logic Programming, more precisely on Datalog programs. It has the least expressive power but provides a low formal complexity and is decidable. By extending WSML-Core in the direction of Logic Programming with default negation, cardinality constraints, n-ary relations with arbitrary parameters and meta-modeling features a new language, WSML-Flight, is defined.
A further extension in the same direction with function symbols results in a new language variation called WSML-Rule. WSML-Rule no longer requires safety of rules. The only

differences between WSML-Rule and WSML-Flight are in the logical expression syntax. Extensions of WSML-Core extension to a full-fledged description logic resulted in WSMLDL. WSML-Full is based on First Order Logic and acts as umbrella language, unifying all the above varieties.

2.3.1.3 The Web Service Modelling Execution Environment (WSMX)

The Web Service Modeling Execution Environment (WSMX)[6] is the reference implementation for WSMO [14]. WSMX aims to provide a test bed for WSMO and as well to demonstrate the viability of using Semantic Web Services as a means to achieve dynamic interoperation between business partners. WSMX uses Semantic Web technologies to discover, mediate, select and invoke Web services based on their formal descriptions. In short, WSMX functionality could be summarized as performing discovery, mediation, selection and invocation of Web services on receiving a user goal specified in WSML [15], the underlying formal language of WSMO. The user goal is first matched against the formal descriptions of Web services registered with WSMX. In case of success, one or more service descriptions (ranked according to user preference) can be returned. The most appropriate service selected by the user is further invoked and the result is given back to user. Prior the invocation step, WSMX ensures that the data provided for the service invocation is in the format that Web service expects. If necessary a data mediation process is performed to assure the inter-operability between different entities. Presently, the WSMX architecture relies on a set of loosely-coupled main components that provide functionality for each step of Web service usage process: discovery, selection, mediation and invocation.

Being one of the major SWS approach, WSMO has been used in many European funded projects as a solution to describe Semantic Web services. It provides a rich support to describe various aspects of services. Having a framework that provides rich support could be sometimes less beneficial if we consider the learning curve of developers that want to annotate services and the tool support required. In some cases, a simpler, lighter support would be more appropriate. In this context, SOA4All project will developed a lighter version of WSMO, called WSMO-Lite that provides simple annotation support for services following the same principles described previously in this section.

2.3.2 OWL-S

OWL-S (2004), part of the DAML program[7], is an OWL-based Web Service Ontology; it aims at providing building blocks for encoding rich semantic service descriptions, in a way that builds naturally upon OWL. Very often the OWL-S ontology is referred to as a language for describing services, thus reflecting the fact that it provides a vocabulary that can be used together with the other aspects of the OWL to create service descriptions.
The OWL-S ontology mainly consists of three interrelated sub-ontologies, known as the profile, process model, and grounding. The profile is used to express 'what a service does,' for purposes of advertising, constructing service requests, and matchmaking; the process model describes 'how it works, to enable invocation, enactment, composition, monitoring, and recovery; and the grounding maps the constructs of the process model onto detailed specifications of message formats, protocols, and so forth (normally expressed in WSDL).

OWL-S has been the first approach for an overall framework for describing Semantic Web

6 www.wsmo.org

7 http://www.daml.org/

Services, starting in 2001 and has as predecessor DAML-S[8]. OWL-S defines an ontology system for describing Web Services, using OWL as the description language. The OWL-S upper level ontology comprises four major elements: ***Service***, ***Service Profile***, ***Service Model*** and ***Service Grounding*** which are illustrated in Figure 2.

- the ***Service*** concept serves as an organizational point of reference for declaring Web Services; every service is declared by creating an instance of the Service concept
- the ***Service Profile*** holds information for 'service advertisement' which is used for Web Service Discovery. This is the name of the service, its provider and a natural language description of the service, as well as a black-box description of the Service (specifying the input, output, preconditions and effects (short: IOPE).
- the ***Service Model*** contains descriptive information on the functionality of a service and its composition out of other services, described as a process. The model defines three types of processes (atomic, simple, and composite processes), whereof each construct is described by IOPEs, as in the Service Profile, with optional conditions over these.
- the ***Service Grounding*** gives details of how to access the service, mapping from an abstract to a concrete specification for service usage. Although not restricted to one grounding technology, WSDL is favored for this.

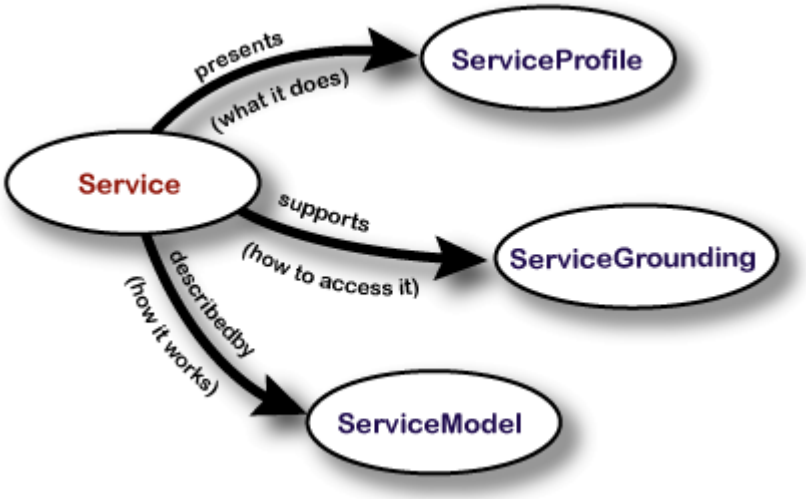

Figure 2 OWL-S top level elements

Each instance of Service will present a ServiceProfile description, be describedBy a ServiceModel description, and support a ServiceGrounding description. More details about each OWL-S top level element are provided in the following subsections.

2.3.2.1 OWL-S Service Profiles

The Service Profile provides means to describe the services offered by the providers, and the services needed by the requesters. No representation of services is imposed by the Service Profile, but rather, using the OWL sub-classing it is possible to create specialized representations of services that can be used as service profiles. However, for pragmatic reasons, OWL-S provides one possible representation through the class Profile. A service, defined through the OWL-S Profile, is modeled as a function of three basic types of information:

- The Organization that Provides the Service: The contact information that refers to the entity that provides the service (e.g., contact information may refer to the

[8] http://www.daml.org/services

maintenance operator that is responsible for running the service, or to a customer representative that may provide additional information about the service, etc.).

- The Function the Service Computes: The transformation produced by the service. The functional description includes the inputs required by the service and the outputs generated; the preconditions required by the service and the expected effects that result from the execution of the service.
- A Host of Features that Specify Characteristics of the Service: The descriptions of these features include the category of a given service (e.g., The category of the service within the UNSPSC classification system), quality rating of the service (e.g., some services may be very good, reliable, and quick to respond; others may be unreliable, sluggish, or even malevolent), and an unbounded list of service parameters that can contain any type of information (the OWL-S Profile provides a mechanism for representing such parameters).

The most essential type of information presented in the profile, that will play a key role during the discovery of the service, is the specification of what functionality the service provides. The OWL-S Profile emphasizes two aspects of the functionality of the service:

- The Information Transformation: Represented by inputs and outputs of the service, and
- The State Change produced by the Execution of the Service: Represented by the preconditions and effects of the service.

No schema to describe inputs/outputs/preconditions/effects (IOPE) instances is provided by the OWL-S Profile. However, such a schema exists in the Process ontology. It is expected that the IOPE's published by the Profile are a subset of those published by the Process, thus it is expected that the Process part of a description will create all the IOPE instances and the Profile instance can simply point to these instances. The properties of the Profile class that the OWL-S Profile ontology defines for pointing to IOPE's are summarized as follows:

- **hasParameter**: Ranges over a Parameter instance of the Process ontology; it's role is solely making domain knowledge explicit.
- **hasInput**: Ranges over instances of Inputs as defined in the Process ontology.
- **hasOutput**: Ranges over instances of type Output, as defined in the Process ontology.
- **hasPrecondition**: Specifies one of the preconditions of the service and ranges over a Precondition instance defined according to the schema in the Process ontology.
- **hasResult**: Specifies one of the results of the service.

Result class in the Process ontology; it specifies under what conditions the outputs are generated. This parameter also specifies what domain changes are produced during the execution of the service.

2.3.2.2 OWL-S Service Models

As the OWL-S Profile describes only the overall function the service provides, a detailed perspective on how to interact with the service is needed. This interaction can be viewed as a process, and OWL-S defines the ServiceModel subclass in order to provide means to define processes. The view that OWL-S takes on processes is that a process is not necessary a program to be executed, but rather a specification of the ways a client may interact with a service. A process can generate and return some new information based on information it is given and the world state. Information production is described by the inputs and outputs of the process. A process can as well produce a change in the world. This transition is described by the preconditions and effects of the process.
Informally, any process can have any number of inputs, representing the information that is, under some conditions, required for starting the process. Processes can have any number of outputs, the information that the process provides to the requester. Inputs and outputs are

represented as sub-classes of a general class called Parameter; (every parameter has a type, specified using a URI). There can be any number of preconditions, which must all hold in order for a process to be successfully started. A process can have any number of effects. Outputs and effects can depend on conditions that hold true of the world state at the time the process is performed. Preconditions and effects are represented as logical formulas. OWL-S treats such expressions as literals, either string literals or XML literals. The latter case is used for languages whose standard encoding is in XML, such as SWRL [24] or RDF [26]. The former case is for other languages such as KIF [25] and PDDL [27]. Processes are connected to their IOPEs using the following properties:

- **hasParticipant** which ranges over the Participant class.
- **hasInput** which ranges over the Input class.
- **hasOutput** which ranges over the Output class.
- **hasLocal** which ranges over the Local class.
- **hasPrecondition** which ranges over the Condition class.
- **hasResult** which ranges over the Result class.

A process involves at least two parties. One is the client, from whose point of view the process is described, and another is the service that the client deals with. Both the client and the service are referred to as participants; they are directly linked to a process using the hasParticipant property. Inputs and outputs specify the data transformation produced by the process; they are directly linked to a process using the hasInput and hasOutput properties. Inputs specify the information that the process requires for its execution. Inputs may come directly from the client or may come from previous steps of the same process. Outputs specify the information that the process generates after its execution. The presence of a precondition for a process means that the process cannot be performed successfully unless the precondition is true; preconditions are directly linked to a process using the hasPrecondition property. The execution of a process may result in changes of the state of the world (effects), and the generation of information by the service (referred to as outputs). Such coupled outputs and effects are not directly linked to a process, but through the term result (i.e., through the hasResult property).
Although the above properties are common to all processes defined in
OWL-S, there can be three types of processes:

- **Atomic Processes**: Description of services that expects one (possibly complex) message and returns one (possibly complex) message in response.
- **Composite Processes**: Processes that maintain some state; each message the client sends advances it through the process.
- **Simple Processes**: processes used as elements of abstraction, that is, a simple process may be used either to provide a view of (a specialized way of using) some atomic process, or a simplified representation of some composite process (for purposes of planning and reasoning).

Atomic processes are similar to the actions a service can perform by engaging it in a single-step interaction; composite processes correspond to actions that require multi-step interactions, and simple processes provide an abstraction mechanism to enable multiple views of the same process. Atomic processes are directly invocable and do not consist of any sub-processes; their execution is a single-step execution (as far as the service requester is concerned), that is they take an input message, do something, and then return their output message. On the other side, composite processes are decomposable into other (atomic, simple, or composite) processes; their decomposition can be specified by using control constructs. The control constructs supported in OWL-S include: *sequence, split, split + join, choice, any-order, if-then-else, iterate, repeat-while, repeat-until.*

2.3.3 The SWSF approach

Semantic Web Services Framework (SWSF) [20] is another approach for Semantic Web Services, being proposed and promoted by Semantic Web Services Language Committee[9] (SWSLC) of the Semantic Web Services Initiative[10] (SWSI). It is based on two major components: an ontology and the corresponding conceptual model by which Web services can be described, called Semantic Web Services Ontology (SWSO) and a language used to specify formal characterizations of Web services concepts and descriptions called Semantic Web Services Language (SWSL). This section provides a general overview of the two core components of SWSF approach for SWS namely: SWSO—Semantic Web Service Ontology (Section 2.3.3.1) and SWSL—Semantic Web Service Language (Section 2.3.3.2).

2.3.3.1 The Semantic Web Services Ontology (SWSO)

SWSO presents a conceptual model for semantically describing Web services and an axiomatization, formal characterization of this model given in one of the two variants of SWSL: SWSL-FOL based on First-Order Logic or SWSL-Rules based on Logic programming. The resulting ontologies are called: FLOWS—First-Order Logic Ontology for Web Services, which relies on First-Order Logic semantics, and ROWS-Rule Ontology for Web Services, which relies on Logic Programming semantics. Since both representations share the same conceptual model we will focus our overview on FLOWS, the derivation of ROWS from FLOWS being straightforward.

The development of FLOWS ontology was influenced by the OWL-S ontology and the lessons learned from developing this ontology. Another fundamental aspect in the development of FLOWS is the provision of a rich behavioral process model based on Process Specification Language (PSL) [21]. FLOWS can be seen as an extension/refinement of OWL-S ontology with a special focus on providing interoperability or semantics to existing standards in Web services area (e.g., BPEL, WSDL, etc.) Although there are many similarities between FLOWS and OWL-S ontologies, one important difference is the expressiveness of the underling language. FLOWS is based on First-Order logic, which means it has a richer, more expressive, support than OWL-S which is based on OWLDL, on description logics formalisms.

Being based on First-Order Logic, FLOWS makes use of logic predicates and terms to model the state of the world. Features from situation calculus, like the use of fluents, predicates, and terms which vary over time, were introduced to model the change of the world. Invariant predicates and terms are called in relations in SWSO.

The FLOWS ontology consists of three major components: Service Descriptors, Process Model, and Grounding. The Service Descriptors are used to provide basic descriptive information about the service. The Process Model is used to describe how the service works. The Grounding is used to link the semantic, abstract descriptions of the service provided in SWSO to detailed specifications of messages, protocols, and so forth used by Web services.

In the rest of this section we take a closer look at the elements that are part of the FLOWS Service Descriptors, the FLOWS Process Model and the FLOWS Grounding.

Service Descriptors

9 http://www.daml.org/services/swsl/

10 http://www.swsi.org/

Service Descriptors are the components of FLOWS ontology which provide basic information about a service. By basic information is meant nonfunctional meta-information and/or provenance information. These kinds of descriptions are often used to support the automation of service related tasks like service discovery. They include information like name, textual description, version, etc, which are properties inherited from the OWL-S Profile. A Service Descriptor may include the following individual properties: (1) *Service Name* – this property refers to the name of the service and may be used as a unique identifier; (2) *Service Author* – this property refers to the authors of the service which can be people or organizations; (3) *Service Contact Information* – this property contains a pointer for the agents or people requiring more information about the service; (4) *Service Contributor* – this property refers to the entity responsible for updating the service description; (5) *Service Description* – this property contains the textual description of the service; (6) *Service URL* – this property contains the URL associated with the service; (7) *Service Identifier* – this property contains an unambiguous reference to the service; (8) *Service Version* –
this property contains an identifier to the specific version of the service; (9) *Service Release Date* – this property contains the release date of the service; (10) *Service Language*—this property specifies the language of the service; (11) *Service Trust* – this property described the trustworthiness of the service; (12) *Service Subject* – this property refers to the topic of the service; (13) *Service Reliability* – this property contains and entity used to indicate the dependencies of the service; (14) *Service Cost* – this property contains the cost of invocation for the service.

Process Model

The Process Model is that part of FLOWS ontology which offers the needed constructs to describe the behavior of the service. The Process Model extends towards the Web services requirements the generic ontology for processes provided by PSL approach, by adding two fundamental elements: (1) the structured notion of atomic process as found in OWL-S and (2) the infrastructure for specifying various forms of data flow. The core part of the PSL extended by FLOWS is called PSL Outer Core and the resulting FLOWS sub-ontology is called FLOWS Core. Based on these extensions FLOWS Process Model ontology can be regarded as a combination of six ontology modules namely:

- *FLOWS-Core:* Introduces the basic notions of activities as activities composed of atomic activities.
- *Control Constraints*: Axiomatize the basic constructs common to workflow- style process models.
- *Ordering Constraints*: Support the specification of activities defined by sequencing properties of atomic processes.
- *Occurrence Constraints*: Support the specification of nondeterministic activities within services.
- *State Constraints*: Support the specification of activities which are triggered by states that satisfy a given condition.
- *Exception Constraints*: Provides support for modeling exceptions.

As part of the FLOWS-Core some basic terms are defined:

- *Service:* A service is defined as an object which has associated a set of service descriptors and an activity that specifies the process model of the service, activities called service activities.
- *Atomic Process:* An atomic service is a PSL activity, that is, in general a sub-activity of the activity associated with the service. Associated with each atomic process are (multiple) input, output, precondition, and effects. The inputs and the outputs are the inputs and outputs of the program which realizes the atomic process. The

preconditions are conditions that must be true in the word for the atomic process to be executed. Finally, effects are the side effects of the execution of the atomic process. All these are expressed as First-Order logic formulae.

- *Message:* A message is an object in FLOWS-Core ontology which has associated a message type and a payload (body).
- *Channel:* A channel is an object in FLOWS-Core ontology which holds messages that have been sent and may or may not have received.

2.3.3.2 The Semantic Web Services Language (SWSL)

SWSL is a language for describing, in a formal way, Web services concepts and descriptions of individual services. SWSL comes in two variants which are based on two well-known formalisms: First-Order Logic and Logic Programming. The two sub-languages are SWSL-FOL and SWSL-Rules. The design of both languages was driven by compliance with Web principles, like usage of URIs, integration with XML built-in types and XML-compatible namespaces, and import mechanisms. Both languages are layered languages where every layer includes a number of new concepts that enhance the modeling power of the language.
SWSL-Rules is a logic programming language which includes features from Courteous logic programs [22], HiLog [23] and F-Logic [17], and can be seen as both specification and implementation language. SWSL-Rules language provides support for service-related tasks like discovery, contacting, policy specification, and so on. It is a layered-based languages as shown in Figure 3.

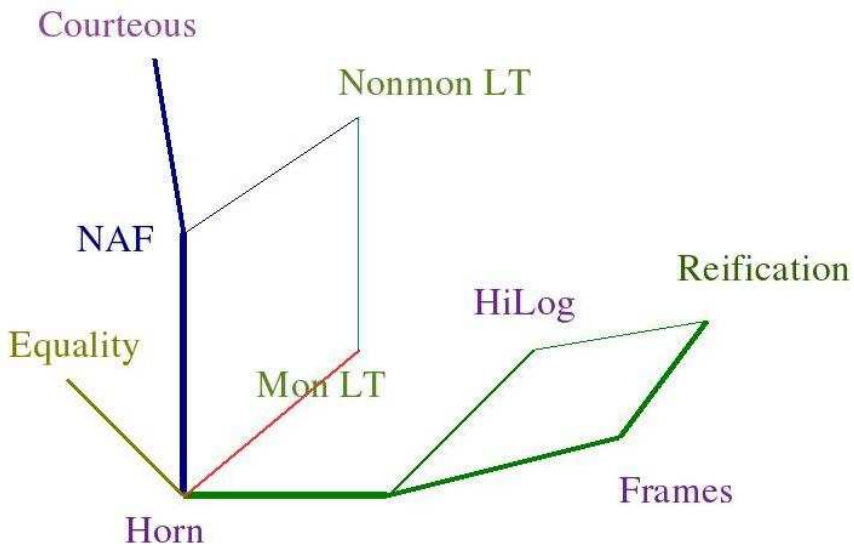

Figure 3 Layered structure of SWSL-Rules

The core of the SWSL-Rules language is represented by pure Horn subset of SWSL-Rules. This subset is extended by adding different features like (1) disjunction in the body and conjunction and implication in the head – this extension is called monotonic Loyd-Topor (Mon LT) [18], (2) negation in the rule body interpreted as nation as failure—this extension is called NAF. Furthermore, the Mon LT can be extended by adding quantifiers and implication in the rule body resulting in what is called nonmonotonic Loyd-Topor (Nonmon LT) extension. Other envisioned extensions are towards: (1) Courteous rules (Courteous) whit two new features: restricted classical negation and prioritized rules, (2) HiLog – enables meta-programming, (3) Frames – add object oriented features like frame syntax, types, and inheritance, (4) Reification—allows rules to be referred and grouped. Finally, equality can be possible extension as well. SWSL-FOL is a First-Order logic which includes features from HiLog and F-Logic. Some of the extensions provided for SWSL-Rules apply for SWSL-FOL

as well. The only restriction is that the initial languages should have monotonic semantics. The resulting extensions depicted in Figure X are SWSL-FOL + Equality, SWSL-FOL + HiLog, and SWSL-FOL + Frame.

2.3.4 IRI-III

IRS-III [10] is a framework and implemented platform which acts as a broker mediating between the goals of a user or client, and available deployed Web services. The IRS uses WSMO as its basic ontology and follows the WSMO design principles. The rest of this section presents the principles which have influenced the IRS (Section 2.3.4.1), and IRS extensions to WSMO (in Section 2.3.4.2). In the rest of the section the terms 'IRS' and 'IRS-III' are used interchangeably.

2.3.4.1 Principles Underlying IRS-III

IRS-III is based on the following design principles:

- Supporting Capability Based Invocation: IRS-III enables clients (human users or application programs) to invoke a Web service simply by specifying a concrete desired capability. The IRS acts as a broker finding, composing, and invoking appropriate Web services in order to fulfill the request.
- Ease of Use: IRS interfaces were designed so that much of the complexity surrounding the creation of SWS-based applications are hidden. For example, the IRS-III browser hides some of the complexity of underling ontology by bundling up related class definitions into a single tabbed dialog window.
- One Click Publishing: A corollary of the above-design principle. There are many users who have an existing system which they would like to be made available but have no knowledge of the tools and processes involved in turning a stand alone program into a Web service. Therefore, IRS was created so that it supported 'one click' publishing of stand alone code written in a standard programming language (currently, we support Java and Lisp) and of applications available through a standard Web browser.
- Agnostic to Service Implementation Platform: This principle is in part a consequent of the one click publishing principle. Within the design of the IRS there is no strong assumption about the underlying service implementation platform. However, it is accepted the current dominance of the Web services stack of standards and consequently program components which are published through the IRS also appear as standard Web services with a SOAP-based end point.
- Connected to the External Environment: When manipulating Web services, whether manually or automatically, one needs to be able to reason about their status. Often this information needs to be computed on-the-fly in a fashion which integrates the results smoothly with the internal reasoning. To support this we allow functions and relations to be defined which make extra-logical calls to external systems – for example, invoking a Web service. Although, this design principle has a negative effect on ability to make statements about the formal correctness of resulting semantic descriptions, it is necessary because our domain of discourse includes the status of Web services. For example, a user may request to exchange currencies using 'today's best rate.' If our representation environment allows us to encode a current-rate relation which makes an external call to an appropriate Web service or Website then this will not only make life easier for the SWS developer, but also make the resulting descriptions more readable.
- Open: The aim is to make IRS-III as open as possible. The IRS-III clients are based on Java APIs which are publicly accessible. More significantly, components of the IRS-III server are Semantic Web services represented within the IRS-III framework. This feature allows users to replace the main parts of the IRS broker with their own Web services to suit their own particular needs.
- Inspectibility: In many parts of the life cycle of any software system, it is important that the developers are able to understand the design and behavior of the software being

constructed. This is also true for SWS applications. This principle is concerned with making the semantic descriptions accessible in a human readable form. The descriptions could be within a plain text editor or within a purpose built browsing or editing environment. The key is that the content and form are easily understandable by SWS application builders.

2.3.4.2 Extension to WSMO

The IRS-III ontology is currently based on the WSMO conceptual model with a number differences mainly derived from the fact that in IRS-III the aim is to support capability driven Web service invocation. To achieve these goals, Web services are required to have input and output roles. In addition to the semantic type the soap binding for input and output roles is also stored. Consequently, a goal in IRS-III has the following extra slots has-input-role, has-output-role, has-input-role-soap-binding, and has-outputrole-soap-binding.
Goals are linked to Web services via mediators. More specifically, the WG Mediators found in the used-mediator slot of a Web service's capability. If a mediator associated with a capability has a goal as a source, then the associated Web service is considered to be linked to the goal.
Web services which are linked to goals 'inherit' the goal's input and output roles. This means that input role definitions within a Web service are used to either add extra input roles or to change an input role type.
When a goal is invoked the IRS broker creates a set of possible contender Web services using the WG Mediators. A specific web service is then selected using an applicability function within the assumption slot of the Web service's associated capability. As mentioned earlier the WG Mediators are used to transform between the goal and Web service input and output types during invocation.
In WSMO the mediation service slot of a mediator may point to a goal that declaratively describes the mapping. Goals in a mediation service context play a slightly different role in IRS-III. Rather than describing a mapping, goals are considered to have associated Web services and are therefore simply invoked.
IRS clients are assumed to be able to formulate their request as a goal instance. This means that it is only required choreographies between the IRS and the deployed Web services. In IRS-III choreography execution thus occurs from a client perspective [10], that is to say, to carry out a Web service invocation, the IRS executes a web service client choreography which sends the appropriate messages to the deployed Web service. In contrast, currently, WSMO choreography describes all of the possible interactions that a Web service can have.

2.3.5 SAWSDL

SAWSDL [30] proposes a mechanism to augment the Web service functional descriptions, as represented by WSDL with semantics. More specifically SAWSDL proposes a set of extension attributes for the Web Services Description Language and XML Schema definition language that allows description of semantics aspects of services. SAWSDL is a W3C recommendation since August 2007. It has been produced by the SAWSDL consortium, which includes some of the SOA4All partners (i.e. STI Innsbruck, OU, IBM). SAWSDL work was motivated by the need of creating a common agreed specification given the growing number of Semantic Web services approaches (e.g. WSDL-S, WSMO, OWL-S), some of these approaches following a top-down approach to described services (WSMO, OWL-S) some others following a bottom-up approach (WSDL-S). The approach followed by SAWSDL was the bottom-up approach with significant influence from WSDL-S [31]. In this section we briefly present the principles SAWSDL is based on (in Section 2.3.5.1), and we shortly describe the extensibility elements used and the annotations that can be created (in Section

2.3.5.2).

2.3.5.1 Aims and Principles

Starting from the assumption that a semantic model of the Web service already exists, SAWSDL describes a mechanism to link this semantic model with the syntactical functional description captured by WSDL. Using the extensibility elements of WSDL, a set of annotations can be created to semantically describe the inputs, outputs and the operation of a Web service. By this the semantic model is kept outside WSDL, making the approach agnostic to any ontology representation language.

The advantage of such an approach is that it is an incremental approach, building on top of an already existing standard and taking advantage the already existing expertise and tool support. In addition the user can develop in WSDL in a compatible manner both the semantic and operational level aspects of Web services.

SAWSDL work is guided by a set of principles, the most important of them being listed below:

- *Build on existing Web services' standards*. Standards represent a key point in creating integration solutions, and as a consequence, WSDL-S promotes an upwardly compatible mechanism for adding semantics to Web services.
- *Annotations should be agnostic to the semantics representation language*. Different Web service providers could use different ways of representing the semantic descriptions of their services and furthermore, the same Web service provider can choose more than one representation form in order to enable its discovery by multiple engines. Consequently, WSDL-S does not prescribe what semantic representation language should be used and allows the association of multiple annotations written in different semantic representation languages.
- *Support annotation of XML Schema data type*. As XML Schema is an important data definition format and it is desirable to reuse the existing interfaces described in XML, SAWSDL supports the annotation of XML Schemas. These annotations are used for adding semantics to the inputs and outputs of the annotated Web service. In addition, an important aspect to be considered is the creation of mappings between the XML Schema complex types and the corresponding ontological concepts. As SAWSDL does not prescribe an ontology language, the mapping techniques would be directly dependent of the semantic representation language chosen.

In the next subsection we present in more details the extensibility elements of WSDL and how they can be used in annotating the inputs, outputs and operations of Web services.

2.3.5.2 Semantic Annotations

SAWSDL introduces the following terminology:

- **Semantic Model:** A semantic model is a set of machine-interpretable representations used to model an area of knowledge or some part of the world, including software. Examples of such models are ontologies that embody some community agreement, logic-based representations, etc.
- **Concept:** A concept is an element of a semantic model. This specification makes no assumptions about the nature of concepts, except that they must be identifiable by URIs. A concept can for example be a classifier in some language, a predicate logic relation, the value of the property of an ontology instance, some object instance or set of related instances, an axiom, etc.

- **Semantic Annotation:** A semantic annotation in a document is additional information that identifies or defines a concept in a semantic model in order to describe part of that document. In SAWSDL, semantic annotations are XML attributes added to a WSDL or associated XML Schema document, at the XML element they describe. Semantic annotations are of two kinds: explicit identifiers of concepts, or identifiers of mappings from WSDL to concepts or vice versa.
- **Semantics:** Semantics refers to sets of concepts identified by annotations.

SAWSDL proposes two basic semantic annotation constructs to be used in annotating the interfaces, operations, faults in WSDL and simple types, complex types, elements and attributes in XSD:

- **modelReference:** extension attribute that denotes a one-to-one mapping between XML or WSDL elements and concepts in some semantic model;
- **schemaMapping:** two extension attributes **lifingSchemaMapping** and **loweringSchemaMapping** that can be added to XSD elements or complex types to associate them with semantic data (used for one-to-many and many-to-one mappings); schemaMapping attributes are used in the post-discovery issues of using a web services.

Each of these elements can be used to create annotations. Listing 8 presents a SAWSDL annotation example for a purchase order interface borrowed from [30].

```
wsdl:description
  targetNamespace="http://www.w3.org/2002/ws/sawsdl/spec/wsdl/order#"
  xmlns="http://www.w3.org/2002/ws/sawsdl/spec/wsdl/order#"
  xmlns:wsdl="http://www.w3.org/ns/wsdl"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:sawsdl="http://www.w3.org/ns/sawsdl">

  <wsdl:types>
    <xs:schema targetNamespace="http://www.w3.org/2002/ws/sawsdl/spec/wsdl/order#"
      elementFormDefault="qualified">
      <xs:element name="OrderRequest"
        sawsdl:modelReference="http://www.w3.org/2002/ws/sawsdl/spec/ontology/
purchaseorder#OrderRequest"
        sawsdl:loweringSchemaMapping="http://www.w3.org/2002/ws/sawsdl/spec/mapping/
RDFOnt2Request.xml">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="customerNo" type="xs:integer" />
            <xs:element name="orderItem" type="item" minOccurs="1" maxOccurs="unbounded" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:complexType name="item">
        <xs:all>
          <xs:element name="UPC" type="xs:string" />
        </xs:all>
        <xs:attribute name="quantity" type="xs:integer" />
      </xs:complexType>
      <xs:element name="OrderResponse" type="confirmation" />
      <xs:simpleType name="confirmation"
```

```
        sawsdl:modelReference="http://www.w3.org/2002/ws/sawsdl/spec/ontology/
purchaseorder#OrderConfirmation">
       <xs:restriction base="xs:string">
         <xs:enumeration value="Confirmed" />
         <xs:enumeration value="Pending" />
         <xs:enumeration value="Rejected" />
       </xs:restriction>
     </xs:simpleType>
   </xs:schema>
  </wsdl:types>

  <wsdl:interface name="Order"
     sawsdl:modelReference="http://example.org/categorization/products/electronics">
    <wsdl:operation name="order" pattern="http://www.w3.org/ns/wsdl/in-out"
       sawsdl:modelReference="http://www.w3.org/2002/ws/sawsdl/spec/ontology/
purchaseorder#RequestPurchaseOrder">
     <wsdl:input element="OrderRequest" />
     <wsdl:output element="OrderResponse" />
    </wsdl:operation>
  </wsdl:interface>
</wsdl:description>
```

Listing 8 SAWSDL annotation example

The annotations in this example appear as *modelReference* and *loweringSchemaMapping* attributes on schema and WSDL elements. Each *modelReference* shown above identifies the concept in a semantic model that describes the element to which it is attached. For instance, the *OrderRequest* element is described by the "OrderRequest" concept in the ontology whose URI is "http://www.w3.org/2002/ws/sawsdl/spec/ontology/purchaseorder." A *loweringSchemaMapping* is also attached to the *OrderRequest* element to point to a mapping, in this case an XML document, which shows how the elements within the *OrderRequest* can be mapped from semantic data in the model.

To annotate WSDL documents the *modelReference* attributes is used. The following WSDL elements can be annotated using the *modelReference* attribute: interfaces, operations, faults. A *modelReference* on a WSDL *interface*, *operation* or *fault* element provides a reference to a concept or concepts in a semantic model that describe the Interface, Operation or Fault.

The *modelReference* attribute can also be used to annotate entities of XML Schema, including simple types, complex types, elements and attributes. Furthermore XML Schema entities can be annotated using the extensions attributes *liftingSchemaMapping* and *loweringSchemaMapping* For concrete example on these annotations we refer the reader to [30].

3 State of the art on service discovery

This section discussed the state of the art on service discovery. The surveyed approaches are grouped to a certain extent according to the service descriptions on which they operate. We start first by summarizing existing efforts to provide registry based approaches, such as UDDI, that are mainly used in conjunction with WSDL service descriptions. Such approaches as detailed in Section 3.1 have been successful in closed, industrial settings but as motivated in Section 3.2 they have failed in open, public domains. Main reasons for failure in open environments are mainly the dynamicity of the environments and not so intuitive and easy to use interfaces. Portal based approaches for search of WSDL services are described in Section 3.2. Finally, logic based approaches for discovery developed in the context of Semantic Web services projects are described in Section 3.4.

3.1 Registries

3.1.1 The Registry/Repository concepts

A registry/repository service provides a foundation for a SOA governance program. It sits at the intersection of design, development, discovery, staging, provisioning, and management of services. A registry service is also an important component in a SOA runtime infrastructure because it provides a central point of reference for information about the services, enabling information exchange among all the products used to implement a managed services network.

SOA governance is “the ability to organize, enforce and re-configure service interactions in an SOA[11]”. Linked to this definition, we can identify two main phases in an SOA governance called design time and runtime. Ability to organize appends at design time with the registry/repository concepts. Ability to enforce and reconfigure appends at runtime with the service platform interface between the service runtime layer and the registry/repository layer.

3.1.2 The registry non competing standards

The industry has defined two non competing standard registry specifications, namely UDDI and ebXML. Both specifications are being developed and standardized at the Organization for the Advancement of Structured Information Standards (OASIS). Both specifications define standards for a general-purpose registry service.

3.1.2.1 UDDI [8]

Universal Description, Discovery and Integration (UDDI) is a platform-independent, XML-based registry for businesses worldwide to list themselves on the Internet. UDDI is an open industry initiative, sponsored by OASIS, enabling businesses to publish service listings and discover each other and define how the services or software applications interact over the Internet. A UDDI business registration consists of three components:

- White Pages — address, contact, and known identifiers;
- Yellow Pages — industrial categorizations based on standard taxonomies;
- Green Pages — technical information about services exposed by the business.

UDDI was originally proposed as a core Web service standard. It is designed to be interrogated by SOAP messages and to provide access to Web Services Description Language documents describing the protocol bindings and message formats required to interact with the web services listed in its directory.

[11] Michael Wheaton – Sun

UDDI was written in August, 2000, at a time when the authors had a vision of a world in which consumers of Web Services would be linked up with providers through a public or private dynamic brokerage system. In this vision, anyone needing a service such as credit card authentication, would go to their service broker and select one supporting the desired SOAP or other service interface and meeting other criteria. In such a world, the publicly operated UDDI node or broker would be critical for everyone. For the consumer, public or open brokers would only return services listed for public discovery by others, while for a service producer, getting a good placement, by relying on metadata of authoritative index categories, in the brokerage would be critical for effective placement.

The UDDI was integrated into the Web Services Interoperability (WS-I) standard as a central pillar of web services infrastructure. The UDDI specifications supported a publicly accessible Universal Business Registry in which a naming system was built around the UDDI-driven service broker. IBM, Microsoft and SAP announced they were closing their public UDDI nodes in January 2006 [2].

Some assert that the most common place that a UDDI system can be found is inside a company where it is used to dynamically bind client systems to implementations. They would say that much of the search metadata permitted in UDDI is not used for this relatively simple role. However, the core of the trade infrastructure under UDDI, when deployed in the Universal Business Registries (now being disabled), has made all the information available to any client application, regardless of heterogeneous computing domains.

3.1.2.2 ebXML [9]

An ebXML Registry is "an information system that securely manages any content type and the standardized metadata that describes it. It provides a set of services that enable sharing of content and metadata between organizational entities in a federated environment. An ebXML Registry may be deployed within an application server, a web server or some other service container. The registry may be available to clients as a public, semi-public or private web site. The ebXML Registry thus provides a stable store where submitted information is made persistent. Such information is used to facilitate business to business relationships and transactions."

In this context, submitted content for an ebXML Registry includes, but is not limited to: XML schema and documents, process descriptions, ebXML Core Components, context descriptions, UML models, information about organizations, and software components.

The ebXML Registry Information Model (RIM) specification defines the types of metadata and content that can be stored in an ebXML Registry. The companion document ebXML Registry Services and Protocols (RS) defines the services provided by an ebXML Registry and the protocols used by clients of the registry to interact with these services.

According to the RIM specification, an ebXML Registry is capable of storing any type of electronic content such as XML documents, text documents, images, sound and video. Instances of such content are referred to as a RepositorytItems. RepositorytItems are stored in a content repository provided by the ebXML Registry. In addition to the RepositoryItems, an ebXML Registry is also capable of storing standardized metadata that may be used to further describe RepositoryItems. Instances of such metadata are referred to as a RegistryObjects, or one of its sub-types. RegistryObjects are stored in the registry provided by the ebXML Registry."

Although a few industry groups have endorsed ebXML Registry, the vendor community has essentially ignored this standard.

3.1.3 Today's registry/repository solutions

3.1.3.1 Vendors registries

Almost all biggest IT solution vendors propose Registry/Repository solution as standalone software of as part of biggest enterprise applications (like Application Server). Here is a list of the principal vendor's solutions:

UDDI based registries:

- IBM: UDDI Registry part of WebSphere Application Server[12]
- Microsoft: Microsoft enterprise UDDI services[13]
- SAP: Registry part of NetWeaver[14]
- HP/Systinet, BEA, Oracle and Tibco: Systinet Registry
- Software AG and Fujitsu: Centrasite[15]
- SOA Software: Workbench

ebXML based registries:

- SUN: Sun Registry Repository[16]

UDDI and ebXML based registries:

- WebMethods/Infravio: X-Registry[17]

Non standard based registries:

- IBM: WSRR[18]

As you can see in this vendor/product list, UDDI standard is the most supported registry standard. So, a minimal UDDI compliance (protocol support) is a mandatory capability of today's registries. It allows integration with a lot of already deployed registry solutions. But UDDI compliance is not sufficient to qualify a registry as a business-level application.

3.1.3.2 Open source registries

Besides these vendor products some open source solutions try to grow on:

- Apache: jUDDI (UDDI implementation)[19]
- University of Hong Kong: freebXML[20] (ebXML implementation)
- OW2: Dragon Governance Platform
- WSO2: WSO2 Registry[21]

---

12 http://www-306.ibm.com/software/webservers/appserv/was/

13 uddi.microsoft.com/

14 http://www.sap.com/platform/netweaver/index.epx

15 http://www.infoq.com/zones/centrasite/overview

16 http://www.sun.com/products/soa/registry/index.html

17 http://www1.webmethods.com/PDF/datasheets/Infravio_X-Registry_Datasheet.pdf

18 http://www-306.ibm.com/software/integration/wsrr/

19 http://ws.apache.org/juddi/

20 freexml.org

– Mule: Mule Galaxy

3.1.4 Dragon: an emerging Open Source Registry/Repository

Dragon is the open-source distributed semantic registry developed by EBM WebSourcing in the context of the OW2 consortium.

3.1.4.1 Dragon main functionalities

The Dragon SOA project provides a full set of functionalities that targets large scale SOA:

At design time: a registry/repository:

- That allows to store information about services, SLA contracts, and others meta-data such as semantic properties.
- It allows service lookup and discovery based on meta-data as well as service life-cycle management.
- It provides impact analysis functionalities facilitating service and processes updates. This impact analysis is based on dependency management between services, artefacts etc.
- It validates service artefacts during registration by enforcing registration policy (such as WS Basic Profile conformance etc.).

At run time: a Service Platform interface (such as JBI interface with PEtALS[22]) for:

- Policies enforcement of QoS attributes.
- SLA enforcement with a special emphasis on consumer/provider contracts.
- Dynamic composition and routing.
- Management of service versions.
- Monitoring of SOA indicators (QoS, service usage/reuse, development time).

[21] http://wso2.org/projects/registry

[22] PEtALS is an open source Enterprise Service Bus : http://petals.objectweb.org/

3.1.4.2 Dragon architecture overview

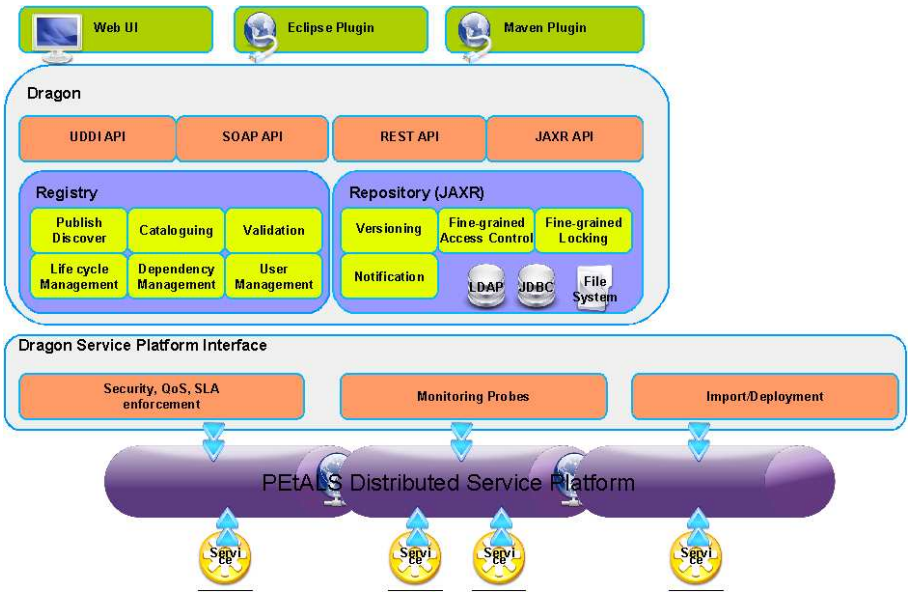

Figure 4 Dragon Architecture

Dragon SOA Governance Platform is composed of two main components:

Dragon registry/repository:

Provides classical registry/repository functionalities provided by UDDI based registry but also includes some enhanced governance functionalities like:

- Repository allows storing metadata about services that include all information necessary for governance processes: WSDL, SLA policies, composition models, transformation and mapping sheets, semantics, etc.
- Registry allows service lookup and discovery based on different flavours: UDDI v3 compliant API cataloguing, policy and ontology based semantic search.
- Registry provides management interfaces to customize services cataloging, validation policies and other tasks.
- A life-cycle manager allows to manage life-cycle of services, policies, SLA contracts and other meta-data about services.
- Dependency management and versioning provides impact analysis and migration helpers for services updates and replacements.

Dragon Service Platform Interface:

It is the communication layer between Service Runtime Plaform (PEtALS Service Platform) and Dragon registry/repository:

- Security, QoS, SLA enforcement: allows to take into account necessary policies enforcement: specified in SLA contract, enterprise wide policies. The example of the JBI approach allows to set up a pluggable strategy for governance based on a set of pluggable policy enforcement service engines: it may be seen as an intermediate layer between the consumer and the target service enforcing policies such as access control, security and availability

– Monitoring probes: probes (such as JBI "Probes") in conjunction with monitoring GUI allow controlling service availability, life-cycle, QoS, usage.

– Import/Deployment APIs: on one hand, it allows to import deployment environment information like deployed services and deployment environments (Petals nodes and components). On the other hand, it allows deploying new services, enforced policies.

Dragon distributed registry

In the scope of the SOA4All project, Dragon Governance Platform will be extended in order to provide a highly distributed registry that could be connected to the distributed technical registry of the PEtALS service platform. The technical registry simply manipulates technical information (endpoints, interfaces, WSDL descriptions...) used for the routing of messages between ESB managed services. By connecting to this technical registry, Dragon registry shall provide a high level vision of the managed services.

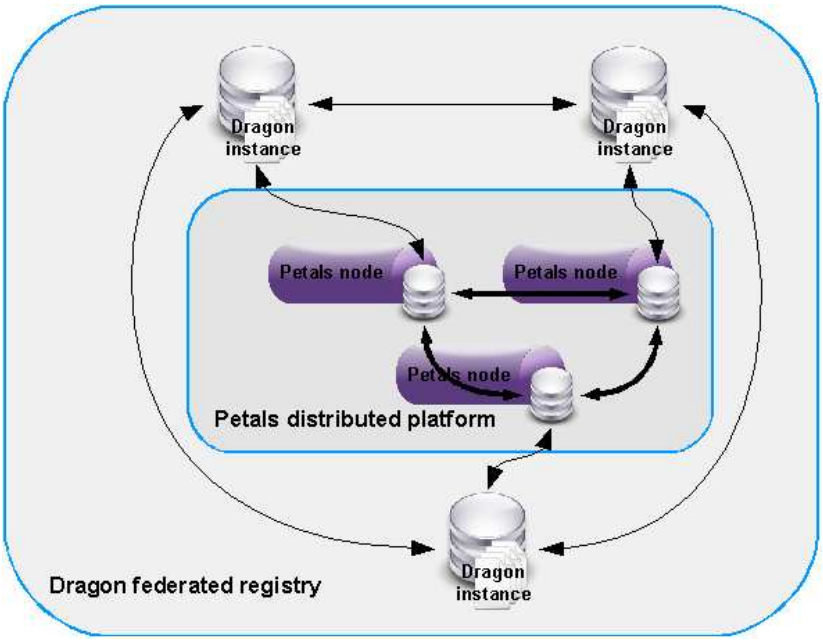

Figure 5 Dragon federated registry

Dragon distributed registry will propose federated queries, selective replication and cross registry reference between registry objects.

3.1.4.3 Dragon Data Model

Dragon data model is based on CBDI-SAE Meta Model for SOA[23].

CBDI Meta Model Overview

This data model is divided into nine packages described below:

– **Service package**: defines the notion of service, as an idea. It defines classification, visibility, and relationship of the services.

– **Business modeling package**: provides a way to model Business Domain, related services, policies and processes.

[23] http://www.cbdiforum.com/public/meta_model_v2.php

- **Specification package**: provides a way to model service specification including operations, dependencies, versions (linked to WSDL Port Types)
- **Implementation package**: defines the notion of Deployable Artefacts, packaged as Automation Units that support a particular Service, Application or Use Case. (linked to SU and SA)
- **Deployment and Runtime package**: defines a way to model service Endpoints running on specified Execution Environments. (linked to WSDL Bindings and Port)
- **Solution modeling package**: provides a way to model Use Cases that supported by Processes.
- **Organization package**: provides a way to model Organizations, their members related to their jobs.
- **Technology package**: provides a way to define Execution Environment where Automation Units can be deployed.
- **Policy package**: provides a way to model Policies applicable to Organization or Service Domain and Business Domain.

This data model has been chosen as a reference for the Dragon data model because of its exhaustive coverage of the main aspect of SOA Governance. Moreover, UDDI data model can be mapped to this data model allowing implementing UDDI APIs on top of it.

3.2 Portals

Within this section we will provide an overview of public portals that are dedicated to Web services. Such specialized portals gather public Web services either by using focused crawlers or by relying on manual registration. They mainly offer a search functionality via a Web interface, some offer as well a browsing functionality.

In the following we shortly describe the main public service portals amongst those that do not actively crawl the Web but rely instead on manual registration of the services by the service providers or portal users:

- *RemoteMethods* - The RemotheMethod[24] Web services directory supports finding and comparing Web services from various providers. The portal that is operated by InfoGenius, Inc. puts actually quite some emphasis on advertisement: besides banners service providers can pay to increase their ranking within the listings.
- *StrikeIron* - StrikeIron[25] is a marketplace of commercial services. It supports the commercialization of Web services and tries to simplify the publishing, finding and subscribing to Web Services by a broader audience of both service providers and users.
- *Wsoogle* - Wsoogle[26] is the successor of Woogle[27]. Wsoogle claims to operate its own crawler to automatically keep the repository up-to-date. They obtain similar operations by comparing operation input and output names with each other. Based upon this data the Web site provides 15 different categories of services that can be browsed. Their categorization technology is based on the Woogle technology [40].

[24] http://www.remotemethods.com/

[25] http://www.strikeiron.com/

[26] http://wsoogle.com/directory.do

[27] http://data.cs.washington.edu/webService/

- *Xmethods* – Xmethods[28] is probably the oldest reference for publicly available Web services. They only provide a simple, long list of Web services and for each Web service a details page with some basic information on the service.
- *ProgrammableWeb* – ProgrammableWeb is a community-driven service directory. Its listing does not mainly contain WSDL service descriptions but also RESTful services and mashups, i.e. composition of services. They do thus have, as compared to the other portals looked at so far, a broader view on the description of a Web service.

Based upon the findings in [38] and [39] we can say that the results are not highly promising. The number of available services for the portals we looked at ranges between 80 (ProgrammableWeb) and 638 (StrikeIron). Nevertheless, due to their specialization, the portals are a convenient way to actually find Web services. For more details on the analysis of Web service discovery in such public portals we refer to [38] and [39].

All the previous Web service portals require either the service providers or portal users to manually register services. In the following we describe the so far only Web service portal and search engine that actively crawls the Web for services, the seekda[29] Web Service Search Engine. The seekda engine contains at the moment of writing this (July 2008) more than 27.000 service descriptions, services not meaning WSDL descriptions but specific seekda services as will be described in Section 5 of this deliverable. These services are provided by more than 7.000 providers.

The seekda portal offers different means for searching for services. Besides a classical keyword search and an advanced search that allows to search using multiple criteria, seekda offers some non-standard Web service search functionalities: portal users can look for services (1) browsing using a tag cloud, (2) browsing service providers by country, (3) browsing for the most used web services, and (4) browsing through recently found Web services. Services can be bookmarked for a later re-use or visit and they can be invoked directly from the portal (live web service tester).

28 http://www.xmethods.com/ve2/index.po

29 http://seekda.com/

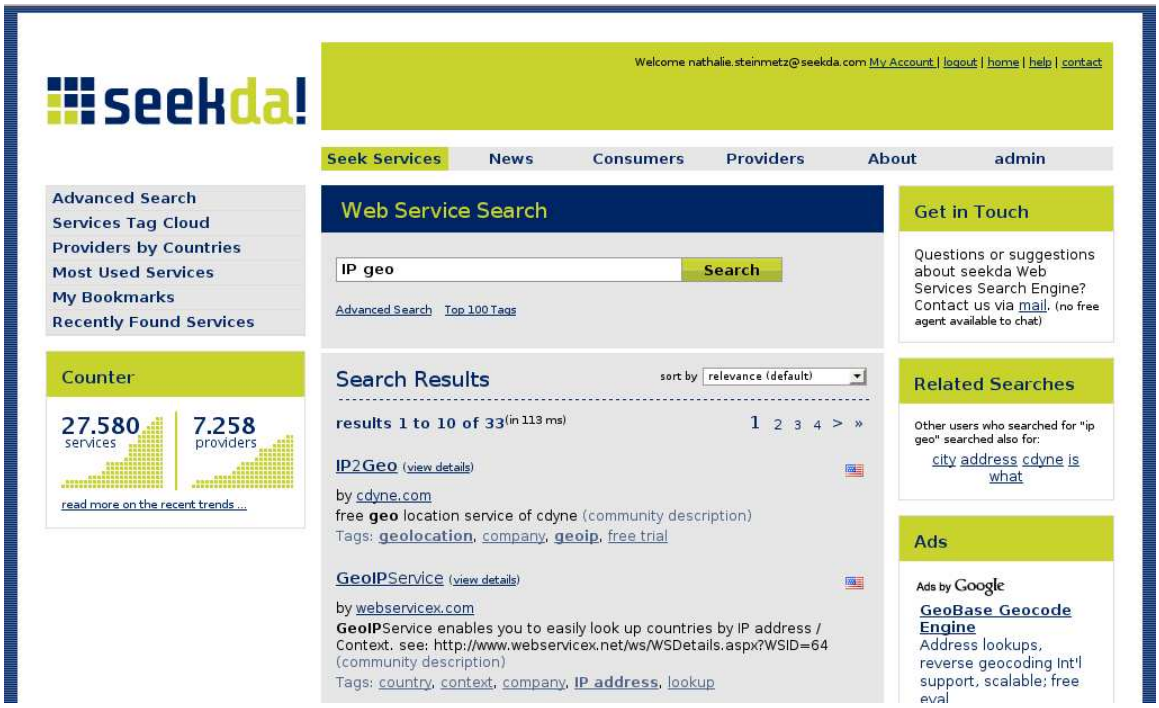

Figure 6 seekda Web Service Search Engine

Although portal users are also encouraged to manually add service descriptions to the registry in case their service is not yet in the index, this is not the usual way. The seekda crawlers are continuously crawling the Web for service descriptions and for related information to these services (as will be explained in Section 4.5). The seekda Web 2.0 portal furthermore encourages users to build a community and to actively take part by e.g. describing the services, tagging them, evaluating them, etc.

3.3 Standard Search Engines

Another approach to Web service discovery is the usage of universal search engines such as Google, Yahoo, Alexa or MSN. As they cover with their crawling engines huge parts of the accessible Web one would expect that they do not only have a high coverage of normal Web pages but also of publicly available Web services. Actually, as results from [38], the number of services that we can find using standard search engines exceeds by far those that we can find using the vertical Web portals, as described in Section 3.2. The biggest disadvantage of searching Web services in standard search engines is though the fact that there is no way to restrict the search to Web service descriptions. One can search for URLs that contain the keyword “wsdl” or for the filetype “asmx”, what fits for services published using Microsoft .NET, but it is not sure whether the results do really resolve to WSDL documents. [39] states that concerning such a search executed in the search engine Alexa[30], only 12% of the resulting URLs actually resolved to WSDL documents. If one assumes similar results for Google and any other universal search engines, the number of valid discovered service descriptions stays bigger than the one for the services discovered on the vertical portals. But the results stay underneath the results obtained by a specialized Web service engine that uses focused crawling techniques, like seekda.

3.4 Logic based approaches

Having described some of the most relevant approaches for service discovery that are

30 http://www.alexa.com/

registry or portal based, as presented in the previous sections, we turn our attention to logic based approaches for discovery. By logic based approaches we understand those discovery approaches that are using descriptions of services and requests formalized using languages based on logical formalisms (Description Logics, First Order Logic, Logic programming) and furthermore employ a reasoner to determine the degree of match between services and requests. In this section we provide a short overview of a set of discovery approaches based on some of the service description languages presented in Section 2.

3.4.1 WSMO discovery

The conceptual model of WSMO Discovery is provided in [34]. WSMO Discovery provides a complete framework for discovery that includes three major steps: *Goal Discovery*, *Web Service Discovery* and *Service Discovery*. Before we provide more details about each step of the WSMO Discovery approach let us introduce one important distinction that was introduced in WSMO in the context of discovery, namely the distinction between a *Web service* and a *service.* A *service,* as in [37], is defined as being a provision of value in some domain (not necessary monetary value). A *Web service* on the other hand is defined as a computational entity accessible over the Internet (using Web service Standards and Protocols).

Given these definitions we have the following relation between the notions:

- Service corresponds to a concrete execution of a Web service (with given input values)
- Web service provides a set of services to its client; one service for each possible input binding

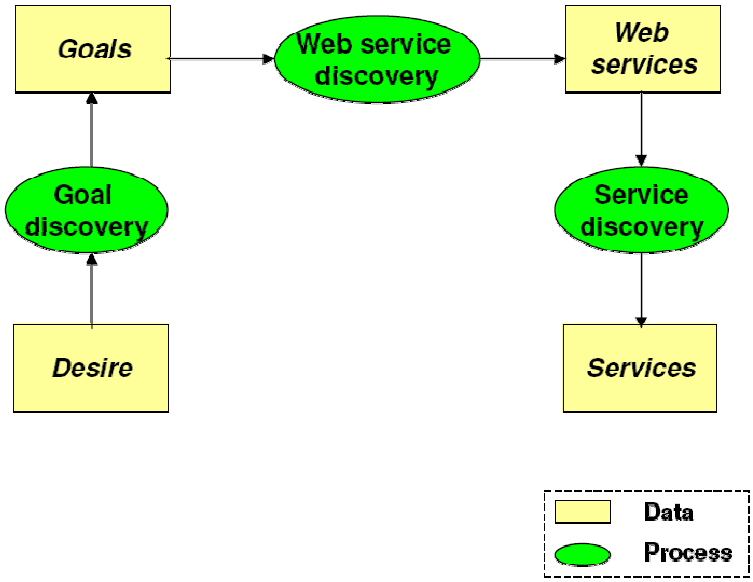

Figure 7 WSMO Discovery model

The overall WSMO Discovery model is illustrated in Figure 7. The first step, ***Goal Discovery*** is about discovering abstract goal descriptions (goals described in WSML) given the input provided by the user (e.g. keywords, logical expressions, both) that represents his/her concrete goal description. The second step, ***Web Service Discovery*** is about how to find abstract web service descriptions (Web Services described in WSML) given the previous found abstract goal. The last step, ***Service Discovery*** is about finding real services whose abstract descriptions where discovered in the previous step. Within Web Service Discovery step three principle approaches are considered: *Syntactical approaches*, *Lightweight semantic approaches* and *Heavyweight semantic approaches*. Syntactical approaches include: keyword-based search, natural language processing techniques, controlled

vocabularies. Lightweight semantic approaches include: ontologies, Action-Object-Modelling, Coarse-grained semantic description of a service. Heavyweight semantic approaches imply that service capability is described in detail and states are taken into account.

A special attention in WSMO is given to the relation between discovery and mediation. This relation is more than natural when we think about the heterogeneity of the environment with different users and services using different terminologies. In order to make communication possible between different parties mediation is required. WSMO proposes a discovery mechanism in strong mediation support.

In the matchmaking process, WSMO Discovery distinguishes between four types of matchmaking:

1. ***Exact Match:*** In this case of matching, the service whose description matches the request description is able to deliver all relevant objects and in the same time no irrelevant objects will be delivered by the services.
2. ***Plug-in Match:*** In this case of matching, the service, whose description matches the request description is able to deliver all relevant objects but might deliver objects which are considered as irrelevant for the goal, too.
3. ***Subsumption Match:*** In this case of matching, the service whose description matches the request description is able to deliver only relevant objects but not necessary all of them.
4. ***Intersection Match:*** In this case of matching, the service, whose description matches the request description, is able to deliver some relevant objects, but might deliver objects which are considered as irrelevant for the goal, too.

The discovery approaches described above are implemented as part of the Web Service Modelling Execution Environment (WSMX). WSMX contains a set of discovery components including a component that uses syntactic matching (keyword-based discovery), a discovery component that uses lightweight semantic descriptions (lightweight DL-based discovery), a QoS discovery component, etc. The WSML datasets used by these components are registered with WSMX repositories. Such descriptions and repositories were/are developed in the context of various projects.

3.4.2 DAML-S/OWL-S discovery approaches

Many approaches for discovery and matchmaking using DAML-S/OWL-S were proposed [35], [36]. In [35] a DAML-S semantic matching between advertisements and request is proposed. The matching algorithm is based on subsumption reasoning in DAML+OIL. A service profile and a request are considered to match when all the outputs of the request goal are matched against all, or a subset of service output, and as well all the inputs of the service are matched against all, or a subset of request goal. In [36], they distinguished between different degrees of matching:

1. ***Exact Match:*** In this case the outputs, respectively the inputs being matched are exactly the same.
2. ***Plug-in Match:*** In this case the output of the service subsumes the output of the request.
3. ***Subsumes Match:*** In this case the output of the request subsumes the output of the service
4. ***Fail:*** No matching services were found for the request goal.

In [34] a different approach for discovery using DAML-S is proposed. Compared with the previous approach all the entities of service profile are used, namely: inputs, outputs (like in

the previous approach) and as well preconditions and effects. They have implemented a prototype based on RACER[31]. Different degrees of matching are consider as well:

1. ***Exact Match:*** In this case the advertisement A and the request R are equivalent concepts.
2. ***Plug-in Match:*** In this case the request R is a sub-concept of advertisement A.
3. ***Subsumes Match:*** In this case the request R is a super-concept of advertisement A.
4. ***Intersection Match:*** In this case the intersection of request R and advertisement A is satisfiable.
5. ***Disjoint Match:*** None of the matches presented above.

The strength of the match is decreasing from the Exact Match to Disjoint Match. By using a Description Logic reasoning procedure to detect possible matching, this approach inherits the time consuming operation of classifying the profiles in profile hierarchy.

[31] http://www.sts.tu-hamburg.de/~r.f.moller/racer

4 State of the art on crawling

Common approaches in Service discovery mostly focus on restricted sets of services. These can be on the one side Semantic Web Service descriptions that describe the functionalities of the services in a rather complex and complete way. They come along with discovery methods that need themselves complex reasoning methods in the background. On the other side we have both public and private registries, as described in Section 3 of this deliverable, that provide access to restricted sets of services. Most of these registries work over services that are manually registered by the providers of the services. Another approach, which is followed by the public Web service search engine of seekda, deals with services on a very large scale, i.e., on Web scale. They actively crawl the Web for service descriptions and do thus collect public Web services from all over the world, not anticipating that people always register their services with them.

In this latter approach Web service discovery is reduced to a special information retrieval (IR) problem. The crawled Web services and related documents are indexed, i.e. their contained keywords are represented in an inverted index. A query to the search engine consists of one or more keywords that are then matched by the engine against the collected documents. What it returns is a (ranked) list of results. That is the process of an IR system represented by the process of a classical search engine.

In the following this chapter will provide an introduction into the topic of Web crawling in general (Sections 4.1, 4.2, 4.3 and 4.4), with a focus on special techniques for crawling for Web services (Section 4.5).

4.1 Web Crawlers

The big success of search engines today is in part due to innovative and effective solutions for web crawling. Search engines actually make the most widespread use of Web crawlers when they collect pages on the Web to build their indexes. A web crawler, also called robot or spider, is a software program that starts with a set of URIs, fetches the documents (e.g. HTML pages, service descriptions, images, audio files, etc.) available at those URIs, extract the URIs from the documents fetched in the previous step and start over the process previously described. The crawler automatically downloads Web pages and follows links in the pages, this way moving from one Web page to another. This gives us already a hint on how we can modify this classical crawler behavior to focus a crawl on Web service descriptions. We can, e.g., decide whether we want to follow the links from a specific page or not, depending on various criteria, as, e.g., what top level domain this page belongs to, how many links the page contains (it could be a so-called “link farm”), etc. We will investigate such issues more deeply in the coming section on focused crawling.

Now crawling the Web and downloading Web pages sounds rather easy. However, one big issue is the fact that the Web is not static, but quite the contrary, i.e. very dynamic. There are billions of documents available on the Web and crawling all data and furthermore maintaining a good ‘freshness’ of the data becomes almost impossible. The Web changes rapidly, new pages are added, existing pages are modified and old pages are deleted. To always keep the crawled data up to date we would need to continuously crawl the Web, revisiting all pages we have once crawled. We could do so by over and over again repeating the same crawl and building “snapshots” of the Web or of the part of the Web we are visiting. Whether we need to do this depends a lot on the intention of our crawl. Do we want to archive part of the Web, do we want to crawl one specific part in-depth or do we want to get an idea of how many links lead to a specific document type? [5] describes such possible intentions and corresponding crawling strategies, thereby proposing an adaptive revisiting strategy that is meant to be used for repeated crawls. We will describe two major crawling strategies, incremental and snapshot, in Section 4.3.

In general we can say that different crawling strategies are used for different types of crawlers. Crawler types are thus related to the different intentions they pursue when crawling the Web. The main crawl types are:

- broad or universal crawling: large crawls with a high bandwidth usage where the crawler fetches a large number of Web sites and goes as well into a high depth on each crawled site.
- focused or topical crawling: a number of criteria are defined that limit the scope of a crawl; the crawler fetches similar pages topic-wise, e.g.
- continuous crawling: the crawler continuously visits all URLs in its frontier, i.e. the frontier cannot grow fast and the crawl should be scoped.

Usually crawlers do implement a set of policies that address the issues raised by the different crawling strategies, as e.g. how to handle the dynamics of the Web, etc. [1]. In general we can say that different policies are used for different types of crawlers:

- a *selection policy* that states which page to download,
- a *re-visiting policy* that states when to check that a page has changes,
- a *politeness policy* that states how to avoid overloading websites and
- a *parallelization policy* that states how to parallelize the crawling functionality.

A list with open source and proprietary web crawler implementations is available at [1].

In the following, Section 4.2 will describe in more detail the basic crawl steps and Section 4.4 will give an overview of the above mentioned crawler types. Section 4.5 will provide an insight into specific crawling techniques for Web services.

4.2 Basic Crawl Steps

What a crawler basically does is executing different specific steps in a sequential way. The crawler starts by taking a set of seed pages, i.e. the URLs which it starts with. It uses the URLs to build its frontier, i.e. the list of unvisited URLs of the crawler. In the scope of one crawl this frontier is dynamic as it is extended by the URLs extracted from already visited pages. The edge of a frontier will be limited by the number of URLs found in all downloaded documents (and by politeness restrictions that are followed for different servers). If a frontier is not set any limit and if the crawler disposes over unlimited hardware resources, it may grow indefinitely. This can be avoided by limiting the growth of the frontier, either by, e.g., restricting the number of pages the crawler may download from a domain, or by restricting the number of overall visited websites, what would at the same time limit the scope of the crawl.

Whatever frontier strategy is chosen, the crawler proceeds in the same way with the URLs it gets from the frontier. So once a URL is taken from the frontier it traverses the following steps:

- the crawler checks whether this page is intended to be fetched, i.e. whether there are no rules or policies that exclude this URL
- the document the URL points to is fetched
- the crawler extracts links from the downloaded document
- based on given rules the crawler decides whether it wants to permanently store the downloaded documents
- feed the extracted links to the frontier

These steps are executed for all URLs that are crawled by the Web crawler. As it would (a) absolutely minimize the speed of a crawl and (b) be a wastage of resources, a crawler does not proceed the URLs one by one. Although a crawler has only one frontier, the frontier has multiple queues. Queues can be built based on different schemes: e.g. one queue per host. Additionally it is often possible to rank the queues within the frontier which makes then that

certain queues are served earlier by the frontier than others.

A very important issue is as well the ranking of the URLs in the queues. When a crawl is set up, it must be decided what URLs get what priorities and get thus removed either early or late from a queue to be processed further.

4.3 Crawling Strategies

When a Web crawl is being designed, this always happens with a specific intention. Often this intention is revealed by the originator of a crawl. Big search engine operators as Google, Yahoo or MSN have other intentions than, e.g., national institutions that want to archive one whole country's Web (as it was done, e.g., by the Nordic Web Archive[32]) or news providers that just crawl the Web for news on one specific topic. So depending on the purpose of a crawl, different crawling strategies may be followed. Two major strategies, as described in [5] are incremental and snapshot crawling.

In a snapshot strategy the crawler visits a URL only once. If the same URL is discovered again it is considered as duplicate and discarded. Using this strategy the frontier is extended continuously with only new URLs and a crawl can spread quite fast. Using this strategy the crawl operator can take a snapshot of (part of) the Web at one specific moment in time. This snapshot can be done for a broad scope without major problems, just extending the duration of the crawl. The snapshot strategy is though not appropriate for doing crawls that allow to follow the changes on Web sites, as, amongst others, depending on how long a crawl takes it might take a long time until a page is revisited.

The incremental crawling strategy is, as opposed to the snapshot strategy, optimal for doing continuous crawls, i.e., crawls that allow capturing changes on Web sites. This makes that a URL needs to be visited multiple times: when an already visited URL is rediscovered it is not rejected but instead put into the frontier again. Using the incremental strategy the frontier queues will never empty and a crawl could go on for an indefinite long time. If the scope of the crawl is not limited, the frontier will not only revisit known URLs over and over again but it will at the same time continue to expand the crawl field, even though much slower than using the snapshot strategy.

Both crawling strategies can be used with different types of crawls, as will be described in the next section. [5] mentions an example of how both crawling strategies can complement each other: a part of the Web that shall be crawled can be analyzed with regard to how often the pages change. The snapshot strategy can then be used to crawl the given part in a broad and extensive manner once (or on a regular basis), while the incremental strategy can be added to follow the changes on sites that have been identified as changing frequently (e.g. news sites).

4.4 Crawler Types

This section provides an overview of four major crawl types. We will explain the purposes that underlie the single types as well as specific issues that appear with these crawler types.

4.4.1 Broad Crawling

Broad crawling (or universal crawling) is the type of crawling that can be used with the purpose of crawling a large part of the Web, if not even the whole Web. Not only the amount of collected Web data is important, but as well the completeness of coverage of single Web sites. Big universal search engines like Google, Yahoo or MSN operate such broad crawls. This crawler type can use both crawling strategies as described in Section 4.3. Using the snapshot strategy in a repeated manner over the same seeds, the crawl operator would get

[32] http://nwa.nb.no/

single snapshots of the crawled part of the Web that might though be temporally quite apart. Using the incremental strategy, one crawl could lead to incremental updates of a search engine index, or of any other data repository used.

According to [6] the major issues in broad crawls are:

- *performance* – A broad crawler often needs to handle not only thousands, but billions of Web documents in the smallest possible amount of time. That makes that scalability is a crucial factor in large universal crawls. A huge number of documents needs to be fetched from the Web (bandwidth issue), processed (CPU issue) and stored permanently (disk space issue). Single points that can help improving the performance of a broad crawl include the minimization of Domain Name System (DNS) lookups that the crawler makes to resolve host names to IP addresses, the careful distribution of crawl jobs on a large number of crawl machines, etc.
- *trade-off between freshness, importance and coverage* – As we said already before, the Web is very dynamic. That makes that new pages are added constantly while existent pages get modified or removed. To achieve a high coverage a crawler needs to find the new pages (frontier extension) while to achieve a high freshness it needs to frequently revisit pages (constant frontier). A solution to this problem might be to limit the frequent revisits to pages that once are recognized as really changing frequently (e.g. news sites). Also not each broad crawl operator is actually interested in crawling the whole Web, the purpose might as well be to crawl pages on some specific topic, or others. If the intention is to crawl a large number of sites, it might be necessary to limit the coverage of single Web sites, i.e. limit the depth with which a site is crawled. Although this makes that not each site will be crawled completely, this trade-off enlarges the coverage of the crawl with regard to the whole Web or the intended part of it.

4.4.2 Focused Crawling

As compared to the broad crawler, the intention of the focused (or topical) crawler is to collect pages from a specific domain, category, topic or similar. There exist several ways to implement focused crawlers and to limit the scope of a crawl: by limiting the URLs to be visited to certain given domains, by doing similarity checks between fetched pages and a given set of example pages, by checking fetched pages for keywords related to a given topic, by using supervised learning mechanism where classifiers work over a set of labelled example pages, etc. An example of a topical crawler is the vertical seekda search engine, that as opposed to universal search engines as Google, Yahoo or MSN, focuses its crawls on Web services and related information.

Same as for the broad crawl, the focused crawl can be based upon both crawling strategies as described in Section 4.3. This depends on the importance of collecting the changes of Web sites accurately. A crawler that is intended to collect news pages on a specific topic will use an incremental strategy while a crawler whose purpose it is to collect all pages on one topic in a quite complete manner, will preferably opt for the snapshot strategy. Depending on the scope limitation of a focused crawl, completeness might be an issue or not, as the crawler will be able to crawl the allowed sites in-depth if the scope is sufficiently limited.

An important issue in topical crawling, where the crawl is intended to collect pages related to a specific topic, is how to assign priorities or costs to URLs in queues or to the frontier queues themselves. Such priorities need to be distributed by the time the URL is put into a frontier queue, which makes that there need to be heuristics that help decide on how to assign costs or priorities to URLs. When the goal of the topical crawl is to detect pages related to a specific topic, we need to find good ways, be it heuristics or guesses, to determine whether a yet unvisited page might be on that topic or not. Such heuristics can be based on an analysis of the page where the link comes from, on the domain of the link, or on

many other points.

4.4.3 Continuous Crawling

Continuous crawling is the type of crawling that can be used to accurately follow changes on Web sites, mostly on restricted parts of the Web. This crawler type uses an incremental crawling strategy as described above in Section 4.3. That is after having visited a URL the crawler enqueues it again so that it will be revisited.

There are three important issues to mention concerning such continuous crawls:

- *resources usage* – When we crawl large portions of the Web (e.g. one country's Web) and store the data to build an archive, we need a lot of disk storage space. That is we need to seize all possible opportunities to not unnecessarily increase the need of disk space. One such method would be to detect whether a revisited page has changed since the last visit or not, and to only store it in case it has changed.
- *crawl scope* – As well important in continuous crawls is the fact to limit the scope of a crawl. If the scope is not limited, the frontier will grow indefinitely and will not allow the revisiting of URLs in a reasonably small time frame. This again will lead to the fact that lots of intermediate changes in frequently changing Web sites will not be crawled and will be lost.
- *politeness policies* – We talked before about a reasonable time frame within which we want a crawler to revisit URLs to not loose too many changes on Web sites. This leads us to the obligation of crawling operators to respect certain politeness policies, whereas different Web servers may set up different politeness restrictions. Such politeness policies are necessary to prevent crawlers from crawling certain servers too aggressively, what in the worst case could lead to the crawler being blacklisted.

4.5 Web Service Crawling Techniques

We so far gave an overview on crawling in general, that is mostly applied to crawling for normal Web pages (HTML). Now in the scope of SOA4All we will focus our crawling activities on crawling for Web services and their related information. With the current development of Service Oriented Architectures, the number of services available online are considerably increasing. More and more companies, organisations and persons in general are publishing their WSDL-based and RESTful services on the Web. With the emergence of Semantic Web and Semantic Web services technologies it is envisioned that the number of Semantic Web services published online will start growing fast as well.

The fact as such that many services are available does not yet help any potential service user. The users need to be aware of the services and need to be able to search and find them. Therefore crawling the Web for services, be it WSDL, RESTful or Semantic Web services, becomes an important challenge.

To our knowledge there are very few service crawlers out there so far. One of the existing service crawlers was developed by seekda. seekda's crawler is using (and extending) an existing Web crawler, namely the Internet Archive open source crawler Heritrix [5]. Heritrix is an archival crawler which was developed in the intention to be used for producing archived periodic snapshots of a large portion of the Web. Web search engines, such as Google, are also collecting WSDL descriptions from the Web. As pointed out in [7], crawling services using classical search engines that search over the whole Web results into a higher percentage of resulting active services than using a crawler over the classical service registries only (as e.g. UDDI). But, as we have explained in Section 3, it does not result in the same number of services than the one we can find with a specialized Web service crawler, as, e.g., the one from seekda.

So instead of using standard Web search engines to seek for Web services we will, in the scope of SOA4All, use the specific Web service search engine of seekda and develop further methods to detect Web services. As described in Section 3, the seekda search engine contains more than 27.000 services up to this moment. These services do only consist of WSDL service descriptions, there are no RESTful or Semantic Web services yet. In the following we provide an insight into the approach pursued so far for crawling the Web in a focused way for Web services and related information.

As described in Section 4.2 on basic crawl steps, a crawl for Web services needs to start from a set of seed URLs. These seeds contain, e.g., services that we know already, commonly known service repositories, Web pages that publish or promote Web services or that simply talk about Web services. As we look for services and related information - which is mostly stored in textual documents - we can in our focused crawl already reject a lot of content by default, like images, audio or video files. What we want to look at specifically are textual documents, such as HTML pages, XML files, PDF documents, and other. The targeted documents are all types of files that could be either directly a service description or some related information.

So far we work with the premise that every service is described with a WSDL interface specification. I. e. during the crawl process we check whether a fetched page is in XML and if so, whether it is a valid WSDL description. In a first step the crawl will be focusing on WSDL 1.1 descriptions, as they are prominently used on the Web, as compared to WSDL 2.0 descriptions. After having detected Web services we try to gather more information around the service endpoint (like, e.g., their geographic location or liveliness). Beside the Web service descriptions themselves and endpoint related information, our crawls will as well focus on all relevant sort of service related information. Such information can be quite divers: documents pointing to the service, the service provider's service definition, documents pointing to that definition and vice versa, user forums talking about the service, blogs, etc. As a first step we consider those resources that are directly conneted to the service by a link graph, i.e. that include links pointing to the service interface description and vice versa. This can be extended to regard not only links of first grade (i.e. direct links), but as take into accounts documents from links of higher grade (e.g. a page that links to a page that links to a service).

We expect to gather pages that include general descriptions of the service functionality, FAQs, pricing pages, etc. While the resources located in the same domain (where the service is hosted) will mainly hold descriptive and terms and licenses related information, pages on different domains will likely include information that talks about and eventually ranks the service. In order to be able to gather as much related information as possible, our crawler will, e.g., crawl the sites of service providers more deeply than other sites.

Only a part of the services available on the Web are described using the WSDL standard. Another very dominant service scheme is the one of RESTful services. This is why we will need to extend the crawler to as well take into account RESTful services and mashups (bundled services), so-called Web APIs. We will develop methods to identify these services, based on aspects like the lexical analysis of the pages (e.g. usage of CamelCases, usage of certain keywords related to freqeuently used operation names or HTTP methods) and the analysis of the URIs (e.g. URIs that contain query strings).

A third type of services that the crawler will be extended to are Semantic Web services. We will develop methods to identify different kinds of Semantic Web service descriptions, focusing on WSML-Lite and MicroWSMO descriptions that will be used throughout the SOA4All project to semantically annotate services.

5 Outlook of service crawling techniques

This chapter will first give an overview on the data that is resulting from the crawler that seekda and the University of Innsbruck are operating jointly in the scope of SOA4All. Then we will give an outlook on the upcoming challenges that we will cope with in the scope of the project.

5.1 Crawled service data

After crawling the Web for Web services and related information, as described in Section 4.4, we have three different kind of information:

- *WSDL service descriptions* – We first of all have a large amount of WSDL service descriptions (seekda has currently more than 133.000 WSDL descriptions). Usually there are multiple WSDL files out there that correspond to one single service (due, e.g., to a multiple hosting of the same services). The opposite case also happens, i.e. that one WSDL description contains multiple actual services. We handle these cases by assigning the WSDL descriptions to unique services, this way, e.g., removing duplicates from our services. We first extract the provider from the service description and build a new unique (seekda) URL for the service. This URL contains the provider's name. In a second step we then add the service name to it (e.g. http://seekda.com/providers/cdyne.com/IP2Geo).
- *link graphs* – During the crawl we build link graphs that contain information about which page(s) links to which other page(s). This information is useful to help denoting information that is related to services. Such related information may, e.g., be Web pages that link directly to Web service descriptions (what we call an inlink of first grade).
- *related information* – The hugest amount of data we have after a crawl iteration is the related information that we collected. The fetched documents are stored in the ARC file format (Internet Archive): these archives aggregate data in approximately 100MB large files, the file starts with a special ARC header and concatenates then the single Web pages. With each archive we produce an index file during the crawl that allows us to quickly jump to a specific offset in a specific ARC file and extract the corresponding archive record.

Using these three kind of data that we collect during a crawl we do a first analysis step. After having specified the unique services and their corresponding WSDL descriptions, we go through the link graphs to collect all (by out-/inlink) related information and build an index that assigns related information directly to services (the unique seekda services). After this analysis step we have services, WSDLs and information that we deem related to the services.

5.2 Outlook of service crawling techniques

In the following we will give a short outlook on issues that we intend to tackle in the scope of SOA4All and that shall improve our Web service crawling process and the analysis of the gathered information:

- identification of services – We will provide methods to identify non-WSDL services, like REST, JSON, mashups, as well as semantic service descriptions. As a start we will try to investigate common characteristics of RESTful services, both concerning their structure and their descriptions on normal HTML Web pages. This will help us build criteria that we can use to start crawling for such non-WSDL services. At the same time we will start crawling for semantic service descriptions, like WSML, WSMO-Lite or MicroWSMO.
- identification of related information – So far we identify related information using the link graphs that are built during the crawls. We would like to enhance this process in

that direction that we are able to extract related information that is not linking to the service descriptions. We might find such information by (1) crawling domains of service providers more intensively and (2) doing term vector similarity checks during the crawl to estimate whether a page is related to a service or not.

- analysis of related information – After we have collected related information, be it in the way we do it so far, by using link graphs, or in the way we intend to do it as well, by checking for page similarity, we have a huge amount of (probably) related information. Now we do not want to stop here. The next step will be to properly analyze the extracted Web pages. Such analysis can go into two directions: (1) classifying the related information and (2) extracting concrete information from the data.

  Here the classification is clearly the easier part, although it is already not an easy task as such. For a classification we need to find out what a page is most probably about, e.g., is it a pricing page, the provider's contact page, a service documentation, a terms and conditions page, a FAQ, etc. To do so we can start by analyzing such pages for a restricted set of services and searching for indications that allow us to identify specific page types (e.g., a pricing page most probably contains an unusual high amount of numbers (as compared to normal Web pages) and monetary ISO codes (e.g. EUR, USD) or signs (e.g. €, $). The second analysis goal of extracting concrete information is harder to reach. After having successfully classified the related information we would like to extract concrete information out of them. Such concrete information would include, e.g., pricing schemes or prices (extracted from the pricing page), terms and conditions (extracted from the terms and conditions page), telephone number and email from the provider (extracted from the provider's contact page), etc.

Enhancing the gathering and analysis of related information can in a later step help to improve our crawl process. The knowledge about where we find the most related information will enable us to focus the crawler even more on finding information that is related to services.

5.3 Outlook of SOA4All service discovery

Based on the analysis of various discovery approaches (see Section 3) and other approaches for discovery related tasks such as crawling (see Section 4), this section provides a unified outlook of service discovery in the SOA4All project.

We start by identifying what the core components and functionalities that are required in order to build a discovery approach scalable to billions of services as SOA4All aims. Further on we give an informal overview of possible interaction between them. Please note that this section is an outlook of the overall service discovery approach in SOA4All not a complete specification of this discovery approach. Follow-up deliverables in WP5 – Service Location, due to M12 and M18, might find this investigation an interesting input that could be refined and extended.

From an architectural point of view we envision that the following components are required as part of an overall scalable discovery solution: (1) Crawler, (2) Service repository, (3) Reasoner and (4) Service Discovery components. The relation and role of each of these components in the overall discovery picture is going to be discussed in the rest of this section.

As pointed out in Section 4, in open, public and large scale setting, searching for services becomes a hard challenge given the high number of services as well as their distributed and unknown locations. Most of the existing solutions analysed in Section 3, including public and private registries, portals, standard search engines and logic-based discovery approaches

have been designed with the scalability dimension in mind. They work for relative small number of services available in relative closed environments. An approach that deals with services on a very large scale, i.e., on Web scale requires a crawler component able to collect service from all over the world. Therefore we envision a SOA4All discovery solution in which a crawler component plays an active role in finding available services on the Web and collecting their descriptions and related information relevant for further discovery processing.

Another component that is required as part of the overall discovery approach is the Service Registry. The registry should provide scalable solutions for storing and managing service descriptions. Such a repository should be based on Web-based principles (publish, read/write paradigm) to allow easy access to service descriptions. The crawler component will store the retrieved service descriptions in the registry

Besides being a scalable solution to the order of billions of services, the SOA4All discovery approach needs to offer a reasonable degree of accuracy when given a user request. A reasoner component is needed to enable intelligent matching of user request and service descriptions. However, the scalability aim should not be neglected and thus scalable, robust and fast reasoning components need to be integrated, rather then reasoners based on complex formalisms (as most of the approaches investigated in Section 2).

Last but not least scalable discovery approaches are needed. The discovery work done [34] identifies different techniques and level of semantic descriptions that are required as part of an overall discovery approach. Based on the level of semantics used to describe services and user requests the following approaches are indentified: (1) keyword-based, (2) leightweight and (3) heavyweight discovery. Given the fact that scalability is a central requirement in SOA4All less expressive formalisms are preferable, resulting in lower. processing times. Discovery algorithms and methods are required that investigate the usage of light descriptions (i.e annotations, classifications, tags) as processing data. The discovery component(s) will make direct use of the reasoning component during the discovery process. Another required interaction is between the discovery component and the registry.

6 Conclusions

This deliverable examined a number of current service description languages on one hand and service discovery and crawling techniques on the other hand. In particular we provided an overview of the Web Services Description Language, the WS-* standard for Web service description. The main elements of a WSDL description were described and intuitive examples were provided for each of these elements. We also investigated RESTful services in terms of what they are and how they are described. We furthermore looked at the facilities offered by WSDL and RESTful services. Last but not least we provided an extended overview of the most important frameworks for service description in Semantic Web services domain. We examined how the influence of the Semantic Web has brought new opportunities for service oriented computing with Web services as the base infrastructure. Five different approaches for modelling Semantic Web services were described – WSMO, OWL-S, SWSF, IRS-III and SAWSDL.

Having provided an overall overview of existing approaches for service description we looked at approaches for service discovery and crawling. For service discovery we investigated two different types of approaches: (1) registry based approaches, including UDDI and ebXML based approaches and (2) logic based approaches for Semantic Web service discovery that employ reasoning support to determine the degree e of match between services and user requests.

As a supporting task for service discovery, we looked at service crawling techniques. We first provided an overview of the crawling task in the context of Web with an emphasis on focus crawling techniques. Last but not least we provided an outlook for service crawling techniques that we envision being appropriate in the context of SOA4All project. An initial set of WSDL crawled services provided by seekda that can be used by other SOA4All components has been as well described.

7 References

[1] http://en.wikipedia.org/wiki/Web_crawler

[2] F. Menczer, 1997. ARACHNID: Adaptive Retrieval Agents Choosing Heuristic Neighborhoods for Information Discovery. In *Proceedings of the 14th International Conference on Machine Learning*. Morgan Kaufmann, San Francisco, CA, 227–235.

[3] F. Menczer and R. Belew. 1998. Adaptive information agents in distributed textual environments. In *Proceedings of the 2nd International Conference on Autonomous Agents*. Minneapolis, MN, 157–164.

[4] COTHEY, V. 2004. Web-crawling reliability. *J. Amer. Soc. Inform. Sci. Techn. 55*, 14, 1228–1238.

[5] Kristinn Sigurdsson. Adaptive Revisiting with Heritrix. Master Thesis. Universtiy of Iceland. 2005.

[6] Bing Liu. Web Data Mining – Exploring Hyperlinks, Contents and Usage Data. Springer. 2006.

[7] Eyhab Al-Masri and Qusay H. Mahmoud, Discovering Web Services in Search Engines, IEEE Internet Computing, pag. 74-77, 2008

[8] http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=uddi-spec and http://en.wikipedia.org/wiki/UDDI

[9] http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=regrep and http://xml.coverpages.org/ni2005-02-14-a.html

[10] J. Domingue, L. Cabral, F. Hakimpou, D Sell, E. Motta. 2004. Irs-III: A platform and infrastructure for creating WSMO-based semantic web services. In Proceedings of the Workshop on WSMO Implementations (WIW 2004), Frankfurt, Germany, September 2004. CEUR.

[11] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana: *Web Services Description Language (WSDL) 1.1*, W3C Note, 15 March 2001, available at http://www.w3.org/TR/2001/NOTE-wsdl-20010315

[12] S. Weibel, J. Kunze, C. Lagoze, and M. Wolf. (1998). RFC 2413 - Dublin Core Metadata for Resource Discovery. Technical report.

[13] Object Management Group Inc. (OMG). 2002. Meta Object Facility (MOF) specification v1.4.

[14] Roman D, Lausen H, Keller U, editors. 2007. The Web Service Modeling Ontology (WSMO). WSMO Working Draft D2v1.4, February 2007. Available from http://www.wsmo.org/TR/d2/v1.4/.

[15] N. Steinmetz and I. Toma, editors, The Web Service Modeling Language (WSML). WSML Working Draft D16v1, July 2008. Available from http://www.wsmo.org/TR/d16/d16.1/v0.3/

[16] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider (eds.), *The description logic handbook*, Cambridge University Press, 2003.

[17] M. Kifer, G. Lausen, and J. Wu, *Logical foundations of object-oriented and framebased languages*, JACM 42 (1995), no. 4, 741–843.

[18] J. W. Lloyd, *Foundations of logic programming (2nd edition)*, Springer-Verlag, 1987.

[19] C. Preist 2004. A conceptual architecture for semantic web services. In 3rd International Semantic Web Conference (ISWC2004). Springer Verlag: XX, November 2004.

[20] Semantic Web Services Framework. SWSF Version 1.0. 2005. Available from http://www.daml.org/services/swsf/1.0/.

[21] M. Gruninger, 2003.A guide to the ontology of the process specification language In Handbook on Ontologies in Information Systems, Studer R, Staab S (eds). Springer-Verlag: XX.

[22] BN. Grosof. 1999. *A Courteous Compiler From Generalized Courteous Logic Programs To Ordinary Logic Programs*. IBM Report included as part of documentation in the IBM CommonRules 1.0 software toolkit and documentation, released on http:// alphaworks.ibm.com. July 1999. Also available at: http://ebusiness.mit.edu/ bgrosof/#gclp-rr-99k.

[23] W. Chen and M Kifer. *HiLog: A foundation for higher-order Logic Programming.* Warren DS. 1993. Journal of Logic Programming 15:3, 187–230.

[24] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: A semantic web rule language combining OWL and RuleML, 2003. Available at http://www.daml.org/2003/11/swrl/.

[25] Knowledge Interchange Format: Draft proposed American National Standard (dpans). Technical Report 2/98-004, ANS, 1998. Also at http://logic.stanford.edu/kif/dpans.html.

[26] G. Klyne and J. J. Carroll. Resource description framework (rdf): concepts and abstract syntax, 2004. W3C Recommendation. Available at http://www.w3.org/TR/2004/REC-rdf-concepts-20040210.

[27] PDDL-The Planning Domain Definition Language V. 2. Technical Report, report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[28] T. Berners-Lee, J. Hendler and O. Lassila. 2001. The semantic web. Scientific American 284(5):34–43.

[29] D. Fensel: Triple-space computing: Semantic Web Services based on persistent publication of information: In *Proceedings of the IFIP International Conference on Intelligence in Communication Systems, INTELLCOMM 2004*, Bangkok, Thailand, November 23-26, 2004.

[30] J. Farrell, H. Lausen: Semantic Annotations for WSDL and XML Schema. W3C Recommendation 28 August 2007. Available at: http://www.w3.org/TR/sawsdl/.

[31] R. Akkiraju, J. Farrell, J. Miller, M. Nagarajan, M. Schmidt, A. Sheth, and K. Verma. Web Service Semantics – WSDL-S. Technical note, April 2005. Available at http://lsdis.cs.uga.edu/library/download/WSDL-S-V1.html.

[32] R. Fielding. *Architectural Styles and The Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. http://doi.acm.org/10.1145/1367497.1367606

[33] C. Pautasso, O. Zimmermann, F. Leymann: Restful web services vs. "big'" web services: making the right architectural decision. WWW 2008: 805-814

[34] U. Keller, R. Lara, A. Polleres, I. Toma, M. Kiffer and D. Fensel, D.: *WSMO discovery*. Working Draft D5.1v0.1, WSMO, 2004. Available from http://www.wsmo.org/2004/d5/D5.1/v0.1/.

[35] L. Li, I. Horrocks.: *A software framework for matchmaking based on semantic web technology*. In Proceedings of the 12th International Conference on the World Wide Web, Budapest, Hungary, May 2003.

[36] M. Paolucci, T. Kawamura, T. Payne, and K. Sycara.: *Semantic matching of web services capabilities*. In Proceeding of The First International Semantic Web Conference (ISWC2002), Sardinia, Italy, 2002.

[37] C. Preist, *A Conceptual Architecture for Semantic Web Services*, In Proceedings of the 3rd International Semantic Web Conference 2004 (ISWC 2004), November 2004, 395–409.

[38] M.J. Hadley: The Web Application Description Language. Available at http://research.sun.com/techrep/2006/smli_tr-2006-153.pdf

[39] H. Lausen and T. Haselwanter. *Finding Web Services*. 1st European Semantic Technology Conference, Vienna, Austria, June 2007.

[40] D. Bachlechner, K. Siorpaes, H. Lausen and D. Fensel. Web service discovery – a reality check. In 3rd European Semantic Web Conference, 2006.

[41] X. Dong, A. Y. Halevy, J. Madhavan, E. Nemes and J. Zhang. Similarity search for web services. In VLDB, pages 372-383, 2004.
[42] T. Bellwood (2002). UDDI version 2.04 API specification. Available from http://uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.htm
[43] L. Richardson, S. Ruby. RESTful Web Services – Web Services for the Real World. O'Reilly 2007, ISBN-10: 0-596-52926-0