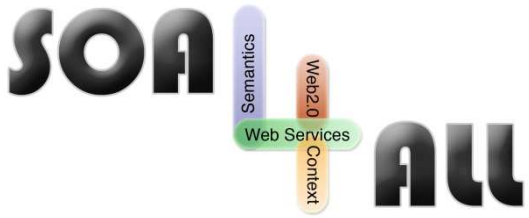

Project Number:	**215219**
Project Acronym:	**SOA4All**
Project Title:	**Service Oriented Architectures for All**
Instrument:	**Integrated Project**
Thematic Priority:	**Information and Communication Technologies**

D5.3.1 First Service Discovery Prototype

Activity:	Activity 2- Core Research and Development	
Work Package:	WP5- Service Location	
Due Date:		M12 Rejected and resubmitted M18
Submission Date:		11/09/2009
Start Date of Project:		01/03/2008
Duration of Project:		36 Months
Organisation Responsible of Deliverable:		UKARL
Revision:		1.0
Author(s):	Sudhir Agarwal UKARL Martin Junghans UKARL Olivier Fabre EBM Ioan Toma UIBK Jean-Pierre Lorre EBM	

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
1.0	29.01.2009	Initial Version	Sudhir Agarwal
2.0	05.02.2009	Section 2 and Section 4 completed	Olivier Fabre, Sudhir Agarwal
3.0	06.02.2009	Executive Summary, Sections 1 and 5	Sudhir Agarwal
4.0	09.02.2009	Section 3	Ioan Toma
5.0	10.02.2009		Sudhir Agarwal
5.1	19.02.2009	Review	Lai Xu
6.0	23.02.2009	Added some references	Olivier Fabre
6.1	03.03.2009	Updated Executive Summary, Section 1, 2, 5 and 6	Sudhir Agarwal
6.2	05.03.2009	Updated Section 3	Ioan Toma
6.3	05.03.2009	Updated Section 4	Olivier Fabre
6.4	05.03.2009	Updated Section 3	Ioan Toma
6.5	05.03.2009	Final Corrections	Sudhir Agarwal
7.0	14.08.2009	Semantic Discovery elaborated; Introduction, Summary, Conclusion revised; Related Work added	Sudhir Agarwal, Martin Junghans
7.1	29.08.2009	Internal review	Yosu Gorroñogoitia (ATOS)
7.2	01.09.2009	Review comments addressed: Architectural description of semantic discovery elaborated; fixes	Martin Junghans
7.3	04.09.2009	Review comments addressed	Jean-Pierre Lorre
7.4	07.09.2009	Internal review	Carlos Pedrinaci (OU)
Final	07.09.2009	Review addressed, Contribution EBM merged	Martin Junghans
Final	11.09.2009	Final Editing	Malena Donato (ATOS)

Table of Contents

List of Figures

Glossary of Acronyms

Acronym	Definition
API	Application Programming Interface
D	Deliverable
DL	Description Logic
EC	European Commission
GUI	Graphical User Interface
HTML	Hypertext Markup Language
PDF	Portable Document Format
RDF	Resource Description Framework
SOA	Service Oriented Architecture
UDDI	Universal Description Discovery and Integration
WP	Work Package
WSML	Web Service Modeling Language
WSMO	Web Service Modeling Ontology
WSDL	Web Service Description Language
W3C	World Wide Web Consortium

Executive summary

Efficient discovery of services is a central task in the field of Service Oriented Architectures (SOA). Service discovery techniques enable users, e.g., end users and developers of a service-oriented system, to find appropriate services for their needs by matching user's goals against available descriptions of services. The more formal the service descriptions are, the more automation of discovery can be achieved while still ensuring comprehensibility of a discovery technique.

Currently, only invocation related information of Web services is described with the W3C standard WSDL while the functionality of a service, i.e., what a service actually does, is described in form of natural language documents. SOA4All develops WSMO-Lite, a lightweight semantic service description language. However, there are still a lot of Web services that are not annotated with WSMO-Lite.

In the scope of this deliverable, we have developed service discovery techniques in order to provide users, e.g., end users, developers, and annotators, with a discovery component that is capable to consider semantic as well traditional non-semantic descriptions of Web services.

The SOA4All service discovery is covered by two types of discovery mechanisms: full text based discovery and semantic discovery. Full text based discovery makes use of the Web service descriptions provided by the WSDL service description files, the Web pages describing REST-ful Web services, and related documents found by the crawler. It allows users to search for services by entering keywords (similar to traditional Web search engines).

The semantic discovery mechanism considers functionalities of Web services formally described with pre-conditions and effects that are provided by WSMO-Lite. This discovery component enables users to enter a more structured goal (service classification, pre-conditions, and effects), then finds the services that match the goal by using the reasoning facilities provided by WP3. While keyword based search mechanisms may scale well, reasoning over logical expressions is computationally expensive. In this deliverable, we also present an approach that on one side considers the functionalities of Web services, on the other side has the potential of scaling to a large number of Web service descriptions.

1. Introduction

Service-oriented computing is an interdisciplinary paradigm that revolutionizes the very fabric of distributed software development, including not only complex enterprise applications, but also scientific, mobile, telecommunication, and embedded system based applications. The success encountered by the Web has shown that tightly coupled software systems are only good for niche markets, whereas loosely coupled software systems can be more flexible, more adaptive and often more appropriate in practice. Loose coupling makes it easier for a given system to interact with other systems, possibly legacy systems that share very little with it.

Web services lie at the crossing of distributed computing and loosely coupled systems. When applications adopt service-oriented architectures (SOA), they can evolve during their lifespan and adapt to changing or unpredictable environments more easily. If properly implemented, services can be discovered and invoked dynamically using non-proprietary mechanisms, while each service can still be implemented in a black-box manner. This is important from a business perspective, since each service can be implemented using any technology, independently of the others. What matters is that everybody agrees on the integration technology, and there is a consensus about this in today's middleware market: customers want to use Web technologies. Despite these promises, service integrators, developers, and providers need to create methods, tools, and techniques to support cost-effective development, as well as the use of dependable services and service-oriented applications. Discovery, which deals with finding appropriate Web services for the task at hand, is one of the central components needed for developing SOA applications.

The importance of the discovery of appropriate Web services grows with an increasing number of Web services, which demands for automated methods that are able to scale for millions of Web services that SOA4All aims at. Automation requires a service discovery that is based on the functionality of services, since simple matching of input and output types still requires a lot of human effort to figure out whether matching Web services offer the desired functionality.

Even though the API provided by UDDI [15] allows random searching for businesses, it does not allow for the selection of new business partners dynamically by a program. Furthermore, UDDI allows only keyword based syntactic search, which is a problem, e.g., when the requester and the provider use different terminology for describing the same thing or use the same terminology for describing different things.

The need for mechanisms to find Web services, which are suitable for the needs of the users, leads to the demand for Web service search engines and service discovery techniques. Within this deliverable, we therefore provide the requirements of a service discovery component, as well as its development based on two lightweight mechanisms that are prototypically implemented within the SOA4All platform.

1.1. Problem Description

Service orientation is a promising paradigm for businesses offering and consuming services within or across organizational boundaries. The increasing acceptance of service oriented architecture and the need for outsourcing business partially or completely will lead to large number of business offered as Web services. Therefore, there is a need for mechanisms, which automatically find Web services suitable for users' needs. That is, there is a need for Web service search engines as there is a need for Web page search engines like Google, Yahoo! and MSN.

However, there is one big difference between Web services and Web pages. Namely, the Web pages are primarily meant for human consumption, whereas Web services are primarily aimed to automate certain tasks by embedding them in larger processes. This main difference creates the demand for different techniques of discovering Web pages and Web services. While from the user's perspective it is the information need that serves as the main criteria for finding Web pages, in case of Web services, it is their functionality that a user has the greatest interest in the discovery process.

In order to enable automatic discovery of Web services according to their functionality, the following required components are identified.

- *A formal model of Web services*
  The formal model should contain all the artifacts that are relevant while searching for Web services according to their functionality. For instance, pre-conditions and effects of a Web service have to be modeled.

- *A Web service description language with formal semantics*
  The Web service description language should allow description of all the artifacts that are relevant for searching for Web services according to their functionality. For instance, such language has to be able to describe pre-conditions and effects syntactically. Of course, the formal semantics of the Web service description language should be a mapping from its syntax to the formal model of Web services.

- *A goal specification language that contains user's search criteria*
  In contrast to a Web service description, which describes the functionality that a Web service offers, a user-specified goal describes the constraints on the properties of Web services a user is interested in for achieving certain task at hand. The goal description also has to feature a formal semantics such that it maps the syntax to the aforementioned formal model.

- *Discovery algorithms*
  Web service discovery algorithms find Web services that offer the desired functionality, which is described by the user goal. When provided with such a goal description and a set of Web service descriptions, discovery algorithms identify those Web service descriptions that fulfill the criteria specified in the goal. Therefore, both the Web service description and the goal are mapped to the common formal model that represents the actual and desired Web service functionality, respectively. A match between goal and a Web service description is identified by a match between the actual (offered) and desired functionality within the formal model.

1.2. WP5 Architecture

This section identifies various components that are needed to build a complete and scalable service discovery solution. The positioning of the WP5 Service Discovery component in SOA4All, and its relationships to the other system components, and respective WPs, is depicted in Figure 1. At the beginning of this section we describe the relationships between the SOA4All components from the service discovery perspective. In a second part we will focus on specific components and their role in the context of service discovery.

WSDL service descriptions and other related documents in the Web are crawled using the techniques described in D5.1.2. The data about Web services available in SOA4All is provided by the seekda Crawler [16], which collects documents in WSDL format, as well as in

other formats like PDF, HTML, etc. that may contain information about Web services. Furthermore, the crawler component is able to extract metadata about Web services and stores them in a structured form, such as RDF. These documents can be used by the Provision Platform to semantically annotate the service descriptions.

The Crawl Data Importer imports the crawled data into the SOA4All Documents Repository. The data importer therefore uses the Crawler Data Read API to retrieve the data provided by the crawler (i.e., RDF Metadata and Documents Archive) and stores the data into the documents repository using the Repository Write API. The semantic annotations of services are stored in the Semantic Spaces. By this, the information in the repository will become accessible by other SOA4All components using the Repository Read API. The SOA4All repository manages various structured information, e.g., WSMO-Lite service descriptions, domain ontologies, and a classification hierarchy as WSML ontology and unstructured documents in a separate document repository.

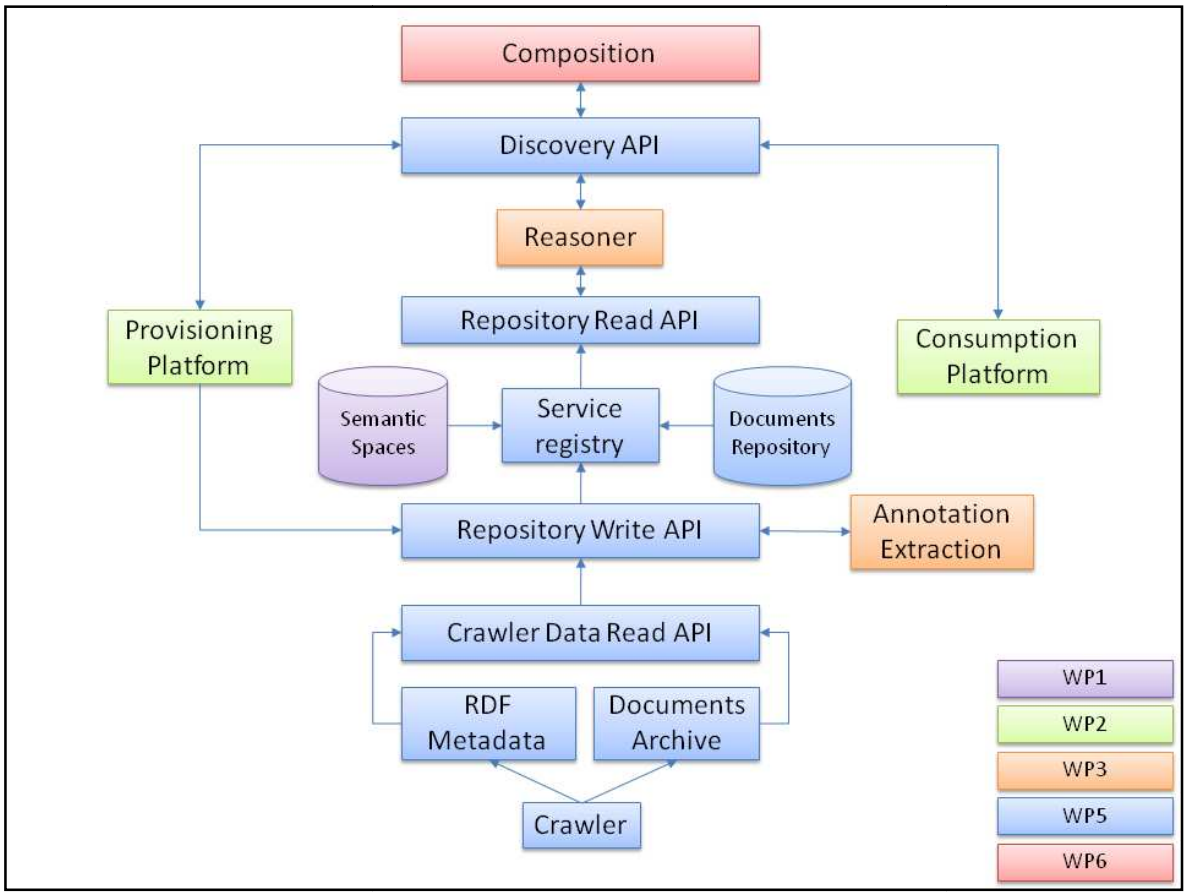

Figure 1: Service Discovery in SOA4All

The ranking component is meant for sorting the matches according to user preferences. The composition component computes combinations of services for fulfilling user requests. The service provisioning platform is a browser based graphical user interface for annotating and publishing service descriptions. The service consumption platform is also a browser based that allows to find and execute services. Except for the user interfaces, all the other components in SOA4All are Web services.

In the remainder of this section we provide a more detailed description of some specific

architectural components illustrated in Figure 1 and indicate their role in the overall service discovery approach.

- **Service Repository (Work Package 5)**

  As stated above, scalability is a targeted goal of the service discovery solution. It is planed that discovery scales to millions of services. One architectural component that discovery relies on to achieve its goal is the Documents Repository that stores information about Web services from the Web. The information is collected by the **service crawler** [6]. Such information about different kind of services (WSDL and REST-ful Web services) comes from various sources in various formats such as WSDL service description files, Web pages describing REST-ful services, related documents, Web service documentation, community feedback such as tagging, semantic description of the services if available, etc. An intelligent index structure is required to allow data to be analyzed and organized in a way that allows the posing of intelligent queries. There is a need for a clear defined interface and architectural interactions between the service discovery and service repository solutions.

- **Reasoning (Work Package 3)**

  The service discovery solution in SOA4All uses the reasoner to provide intelligent matching of goals and available Web service descriptions. Different reasoning functionalities, such as subsumption reasoning, satisfiability checking, instance retrieval, etc. are required by the service discovery component depending on the formal model used to specify services and user requests. To achieve comprehensibility of discovery, the discovery solution must integrate and use the reasoning support. A clearly defined interaction that results in a well defined interface between the two components is required. In an early project stage, two interaction approaches between the discovery and reasoning components can be utilized. First, both components can be exposed as services and the interaction between them is done via Web service standards. Second, the interaction between the discovery and reasoning components can be implemented through direct Java interface calls. While SOA4All is maturing, all services will be integrated into the Distributed Service Bus.

- **Service construction (Work Package 6)**

  The service discovery solution should support the service construction in dynamic and adaptive composition and reconfiguration of constructed services in reaction to environmental changes. Parametric templates that represent service compositions as well as the composition optimizer require at binding time concrete services that will be identified by the service discovery solution.

- **Service ranking (Work Package 5)**

  Service ranking and selection component will order the relevant services that were identified by the service discovery solution and finally select the best fit. From an architectural perspective, service discovery is expected to return a set of service descriptions that belong to the services matching the request.

- **Provisioning Platform (Work Package 2)**

The Provisioning Platform annotates service description in order to provide, enhance, or correct semantic annotations. This platform therefore relies on the service discovery to identify services to annotate, if the service location is not given in advance. Both the full text-based as well as the semantic discovery approach are required to be accessible by the Provisioning Platform.

- **Consumption Platform (Work Package 2)**

  The Consumption Platform that executes services and processes also requires a service discovery. At runtime of a process, the Consumption Platform may require another Web service that has to replace a Web service, which is not available anymore. The Consumption Platform must be able to discover Web services with certain functionality by generating a discovery goal which is sent to the semantic service discovery module, which returns a set of proper Web services to the Consumption Platform.

2. Purpose and scope of this deliverable

SOA4All is proposing a new paradigm where billions of services will be available for the users to interact with them. Thus, in order to enable an interaction with the right services, we need an efficient methodology to discover those amongst a huge number of services, which are relevant from the end user perspective. The present deliverable describes two main approaches for the discovery of Web service descriptions. While the full text based discovery is desired for end users that do not want or, for some reason, are not able to specify the Web service functionality formally, the semantic discovery allows for automation, e.g., required to bind process activities to Web services dynamically at runtime.

Discovery of Web services is one of the most important steps performed by a user in the overall Web services usage process. The discovered Web services are meant to be embedded in a larger process, which is then executed. The results presented in this deliverable will be mainly used by *Service Provisioning* and *Service Consumption* platforms developed in WP2, as well as by the *Semi-Automatic Composition* techniques developed in WP6.

Document Structure

The remainder of this document is structured in the following way.

- Section 3 provides a requirement analysis by considering the problem description (Section 1), existing discovery approaches (D5.1.1), and drawbacks of existing approaches. Based on the requirement analysis, we motivate the need for our discovery approaches.
- Section 4 provides an overview of the different approaches to add full text search to an existing application. Then, it shows how full text search could be added to a JAVA based service registry to provide more sophisticated search capabilities than a classical UDDI service registry. Finally, it describes in details the first SOA4All Service Registry prototype based on Dragon Service Registry.
- Section 5 covers the semantic discovery approach, which is addressed via three different discovery algorithms: (i) the classification-based approach, (ii) the functionality-based discovery that is based on the pre-conditions and effects, and (iii) the algorithm that combines classification and functionality in discovering Web services. All three semantic service discovery approaches consider the WSMO-Lite semantic descriptions of Web services for describing classification ontologies, pre-conditions and effects. The implementation of the third algorithm is also covered in Section 5. It shows the structure and provided functionality of the semantic discovery component, elaborates the usage scenarios, shows the interaction with components of the remaining system, and introduces the GUI.
- Section 6 discusses related work on service discovery and Section 7 presents some concluding remarks.

3. Discovery Requirements

This section provides an analysis of the Service Discovery component in SOA4All. The analysis is based on the architectural requirements and the relationship to the other components of the system. In addition, Section 3.1 provides more specific functional demands grouped as capability and property requirements.

These requirements are derived from the description of the service discovery problem introduced in Section 1 and on the analysis of the state of the art approaches in service discovery provided in [1]. The ultimate purpose is to design a scalable service discovery solution within the SOA4All platform.

At the end of this section, we outline our solution approach. The following Sections 4 and 5 will present our solution more detailed and discuss them thoroughly.

3.1. Functional Requirements

This subsection provides more specific functional requirements on the discovery component, distinguishing between its desired capabilities and properties.

Requirements on Capabilities of Discovery

The following list identifies the capabilities of a service discovery solution, as part of the SOA4All service delivery platform.

- The service discovery solution should consider various types or formalizations of user requests and service descriptions including lightweight annotations (WSMO-Lite descriptions of WSDL-based and REST-ful services) and free text-based descriptions (documents written in natural languages, e.g., PDF or HTML documents).
- The service discovery solution should be able to identify the relevant service from the immense pool of available services.
- The service discovery solution should find services based on a specific functionality based goal, which is a structured query in case of semantic discovery and a set of keywords in case of full text-based discovery.

Requirements on Properties of Discovery

The following list identifies the properties of a service discovery solution as part of the SOA4All service delivery platform:

- *Scalability*: One major objective of SOA4All is to deliver a *scalable* service delivery platform. To achieve this goal one needs first of all a scalable discovery solution. Scalability can be measured in terms of the amount of services and requests, as well as the time interval such entities are handled.
- *Accuracy*: Even though the focus of the SOA4All discovery approach is more on scalability, accuracy is a needed feature that should be also addressed. Accuracy can be expressed by soundness and completeness of the service discovery algorithm. Thus, the number of right services found and the number actually found services can measure the accuracy of service discovery.
- *Robustness*: The SOA4All discovery should function correctly in the presence of incomplete or invalid inputs. Robustness can be expressed by the ability to handle

and successfully process incomplete or invalid inputs for which the service discovery still discovers the right services.

- *Reliability*: The ability of a matching engine to perform its functions, to maintain its service quality. Reliability can be measured by the number of failures of the service discovery solution in a certain time interval.

3.2. Solution Approach

WSDL is the W3C standard for describing Web services. There are two major versions: WSDL 1.x [17] and 2.0 [18] that are both supported for the service discovery Even if WSDL documents are structured such that they can be processed automatically, the information WSDL allows to describe is often not sufficient for the purpose of discovery. The reason behind this is that WSDL allows the description of invocation related information, whereas while performing discovery a user is often concerned about the functionality of Web services. Without semantic Web technologies, the functionality of Web services is often described with natural language, e.g., in form of HTML pages and PDF documents. As a result, finding Web services with desired functionality requires a lot of human effort, since users have to read and understand the natural language documents related to a Web services, in order to decide, whether a Web service offers the desired functionality or not.

In SOA4All, functionality of Web services is described semantically with WSMO-Lite. Semantic annotation techniques developed in SOA4All will facilitate annotation of Web services with WSMO-Lite. Even though creating semantic annotations is mainly a manual task, the manual effort is not invested many times (in most cases only once), availability of semantic annotations help in saving human effort each time a user searches for Web services. Once, the semantic descriptions of Web services are available automatic functionality based discovery be performed. However, during the transition phase from "syntactic" descriptions to semantic descriptions, techniques are needed that help users as well as annotators to find services based on their syntactic descriptions.

In this report, we present a full text-based search that considers WSDL service descriptions and unstructured (e.g., HTML or PDF) documents in a natural language, as well as semantic discovery based on WSMO-Lite descriptions of WSDL-based and REST-ful services. The full text-based search aims at providing a way of retrieving services and related documents giving a set of keywords and a search scope. As stated earlier, such an approach, though highly scalable, lacks expressive search that consider the functionality of a Web service.

WSMO-Lite allows the description of the functionality of Web services with pre-conditions and effects as logical expressions. Furthermore, it proposes to use classification of services to achieve scalable discovery. In this report, we show how WSMO-Lite descriptions can be found based on their classification in a classification hierarchy. Even though this simple approach sounds similar to existing taxonomy based approaches, it differs from them in that the classification hierarchy can be an ontology and the discovery incorporates ontology reasoning. Still, this approach does not consider the pre-conditions and effects of Web services. Therefore, we then develop a discovery algorithm based on pre-conditions and effects and study its complexity. Based on the complexity analysis, we identify parameters that influence the complexity adversely and propose a mix with the classification-based technique to reduce the complexity of functionality based Web service discovery[1], providing an approach that has the potential to scale to a large number of Web services. After presenting the mathematical models and algorithms on the theoretical level, we show how our abstract syntax can be implemented by WSMO-Lite [19].

[1] Currently, we only focus on the functionality and do not consider non-functional properties of Web services.

4. Full Text-based Discovery

The aim of this section is to provide an overview of the full-text service discovery component architecture. The aim of this component is to provide a way of retrieving services and related documents giving a set of key words and a search domain (search on service name or only on related documents etc.). This component is based on the Dragon Service Registry [7], an open source Java based service registry. Dragon provides the full text search API part of the Repository Read API (see Figure 1). As it stores service description and related metadata and unstructured documents extracted from crawled archives, Dragon stands for the Repository Write API and the Document Repository depicted in Figure 1. This section provides an overview of the full text search concepts (see Section 4.1), a description of the software libraries that provides full text search capabilities and especially the ones involved in the Service Registry (Lucene [3] and Compass [2]) (see Section 4.2) and a description of the registry itself and its API (see Section 4.4).

4.1. Full Text Search Overview

In today's applications, search is becoming a "must have" requirement. Users expect applications (rich clients, web based, sever side, etc.) to provide snappy and relevant search results the same way Google does for the web. Let it be a recipe management software, a trading application, or a content management driven web site, users expect search results across the whole application business domain model.

Service discovery isn't an exception to the rule. Indeed, managing thousand, not to say millions services without the ability to efficiently search the one you need, will drive to miss one of the principal benefits of service architecture: reusability.

When dealing with a small number of documents it is possible for the full-text search engine to directly scan the contents of each document with each query, sequentially. This is what some rudimentary tools, do when searching.

However, when the number of documents to search is potentially large or the quantity of search queries to perform is substantial the problem of full text search is often divided into two tasks: indexing and searching. The indexing stage will scan the text of all the documents and build a list of search terms, often called an index [11], but more correctly named a $\mathrm{concordance}$. In the search stage, when performing a specific query, only the index is referenced rather than the text of the original documents.

The indexer will make an entry in the index for each term or word found in a document and possibly its relative position within the document. Usually the indexer will ignore $\mathrm{stop\ words}$, such as the English "the", which are both too common and carry too little meaning to be useful for searching. Some indexers also employ language-specific $\mathrm{stemming}$ [12] on the words being indexed, so for example any of the words "drives", "drove", or "driven" will be recorded in the index under a single concept word "drive".

4.2. Adding full text search to an application

There are two basic ways to add full text searching to an application. The first approach is to find a database that provides full text support. Many modern databases do provide this support, and it can be very easy to access. Adding searching capabilities to the database itself has the advantage of making searches available to any application that uses the database. However, these solutions are usually coarser than custom purpose search engines, and getting outside the confines of their architecture can sometimes be difficult.

The second approach is a stand-alone search engine. In general, these will have many more options for how you index your data and how you can access the data. Removing the functionality from the database means that you can avoid hits on the database or even move

this functionality to another server if you so desire. A disadvantage is that it is another moving piece of your application that will need to be set up. Beyond this initial hurdle there is a second issue. Since your search indexes will be truly removed from the data that they represent in the database, they're not going to automatically stay in sync. That means that we're going to have to make sure that updates, insertions and deletes are propagated to the search index as well. Happily, some search engine take care of this for us transparently.

In the following we describe two solutions that illustrate these two approaches.

4.2.1. Full text search provided by the database: MySQL [32] case

MySQL's full text search capabilities are a decent solution for quickly setting up full text search capabilities on a database. One of the advantages of having an index that's well integrated with the database is that your index is always up to date and you won't have to worry about making sure that updates and deletions are propagated to the full text index. Let's take a look at what a MySQL query would look like. We'll search a sample blog application for occurrences of the word "apple" that don't include "computer" and we'll return the relevancy score:

```
SELECT id, title,
        MATCH (title, text) AGAINST ('+Apple –Computer' IN BOOLEAN MODE) AS Score
FROM blog
WHERE
        MATCH (title, text) AGAINST ('+Apple -Computer' IN BOOLEAN MODE)
```

As you can see, this solution offers a pretty easy way to quickly improve the search capabilities of a database. There are, however, some issues with using MySQL full text search:

- Full text search is not available with InnoDB[2] tables, meaning that our searchable tables will not support transactions.
- It's not flexible enough to support different analyzers. The index will be created directly from the tables, meaning we won't have an opportunity to modify/format data before indexing.
- There's not a lot of support for specifying how and when indexing occurs.
- It will require database-specific calls, tying us to MySQL.

Because of these issues, applicable to other databases, the second approach, based on a stand-alone search engine, seems to be more relevant.

4.2.2. Full text search with a stand-alone search engine

Open source search engines: Lucene [3] and Sphinx [37]

The most popular open source search engines are Lucene and Sphinx.

Lucene is actually a Java library, so basically you don't have to install anything to use Lucene. It is a very powerful tool for indexing and searching documents. Lucene doesn't care do you index files, database records or something else. Lucene only supports simple text format indexing. So, it is your duty to prepare data (files, database record, etc.) and send it to Lucene index simple text documents. That means that you will need some extra work to index your database records, or whatever data you want to search; some open source

[2] InnoDB is a storage engine allowing MySQL database to support ACID transaction

library, like Compass described in the next section, do the job for you. Once created, index can be easily updated or expanded by adding/updating documents from your scripts and programs.

Lots of popular heavy traffic sites are powered by Lucene search like Wikipedia [33] or Digg [34], so the performance and scalability is not under the question. Lucene is also rich with search options providing boolean search, phrase search, wildcard search, term search etc.

Lucene was developed by Doug Cutting during 1997-8 [13]. Lucene is a Java-based open source toolkit for text indexing and searching. It is one of the projects of Apache Jakarta and is licensed under the Apache Software License. This license allows to easily integrating Lucene to all SOA4All components with no additional fees.

The other popular open source search engine, Sphinx is a full text search engine and it is ready-made to be used with open source databases like MySQL (MyISAM, InnoDB) and PostgreSql. It is very easy for installation. Configuration is simple as well. Basically, you will specify the SQL connection parameters and the SQL query to fetch the data from your database, and the Sphinx indexer will create the index for your database.

Indexing is very fast, but the problem is that there is no way to update index partially, for example to update/add only couple of records. Instead, you have to re-index an entire dataset from the beginning for each update you made on your application data.

Sphinx supports all major programming languages, including PHP, Python, Java, Ruby and C++. All tools used by SOA4All to index and store services related document (like Dragon Service Registry) are Java based. So this Sphinx specificity isn't very relevant in SOA4All context.

Sphinx is distributed under GPL and commercial licensing. So it is necessary to pay for a license to embed Sphinx into an application.

It should be noted that these search engine libraries are not full-featured search applications that one can start using 'as is'. They are software libraries, with indexing and searching capabilities that can be integrated with various applications. Lucene is the most popular one and is encapsulated into many high level libraries, like Compass or Hibernate Search described in the next section. So as they propose similar search capabilities and performance levels, the most relevant solution, in the SOA4All context, seems to be Lucene.

Full text search with Lucene

Building a full-featured search application using Lucene primarily involves indexing data and searching indexed data.

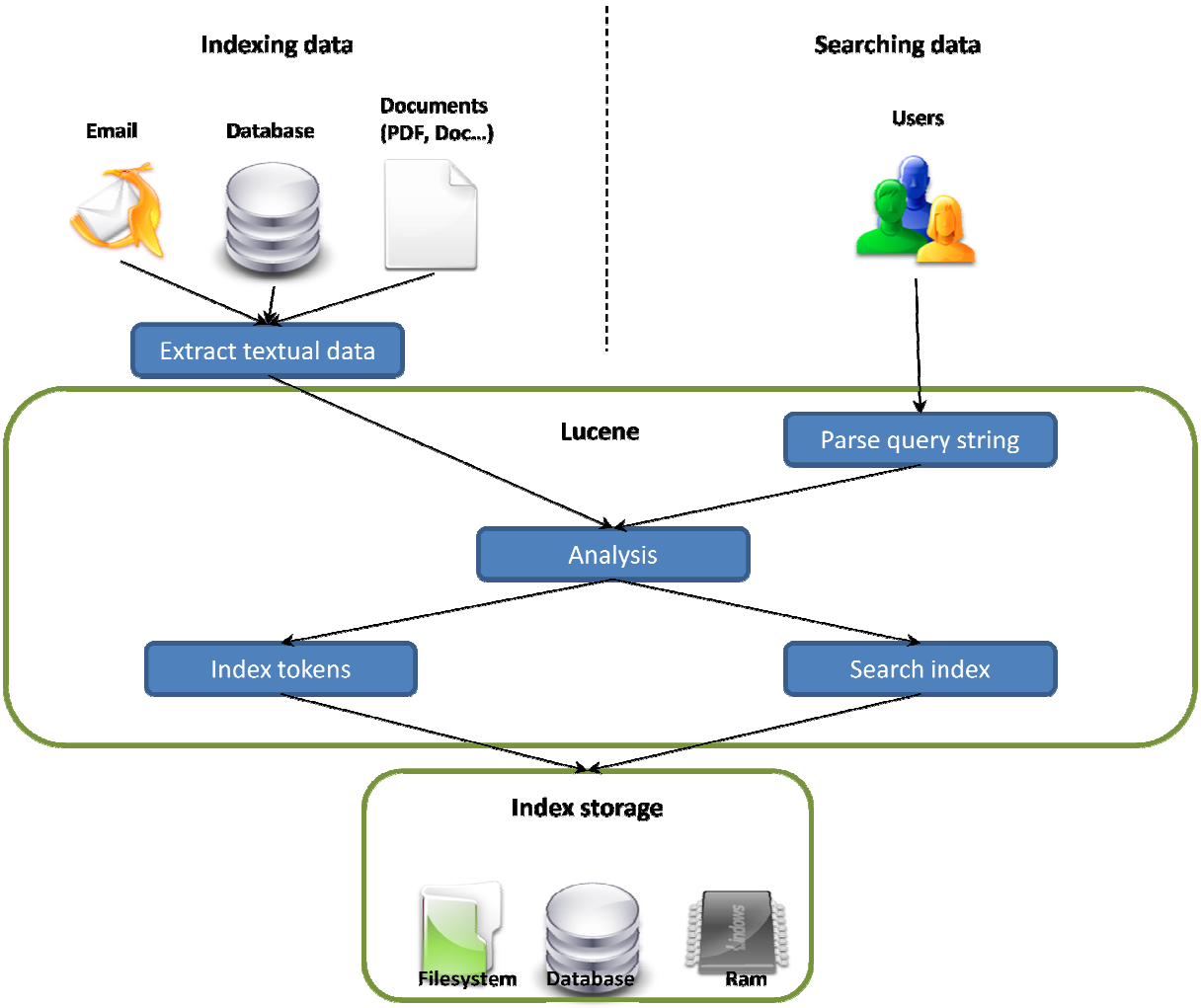

Figure 2: Indexing and searching data with Lucene

1. Indexing data

Lucene lets you index any data available in textual format. Lucene can be used with almost any data source as long as textual information can be extracted from it. You can use Lucene to index and search data stored in HTML documents, Microsoft® Word documents, PDF files, and more. The first step in indexing data is to make it available in simple text format. You can do this using custom parsers and data converters. This type of converters are provided by high level libraries like Compass or some Lucene related projects like Tika [35].

The indexing process

Indexing is a process of converting text data into a format that facilitates rapid searching. A simple analogy is an index you would find at the end of a book. That index points you to the location of topics that appear in the book.

Lucene stores the input data in a data structure called an *inverted* index, which is stored on the file system or memory as a set of index files. Most Web search engines use an inverted index. It lets users perform fast keyword look-ups and finds the documents that match a given query. Before the text data is added to the index, it is processed by an analyzer (using an analysis process).

Analysis

Analysis is converting the text data into a fundamental unit of searching, which is called as *term*. During analysis, the text data goes through multiple operations: extracting the words,

removing common words, ignoring punctuation, reducing words to root form, changing words to lowercase, etc. Analysis happens just before indexing and query parsing. Analysis converts text data into tokens, and these tokens are added as terms in the Lucene index.

Lucene comes with various built-in analyzers, such as WhitespaceAnalyzer, StandardAnalyzer, StopAnalyzer, SnowballAnalyzer, and more. These differ in the way they tokenize the text and apply filters. As analysis removes words before indexing, it decreases index size, but it can have a negative effect on precision query processing. You can have more control over the analysis process by creating custom analyzers using basic building blocks provided by Lucene. Here is a description of the main build-in analyzers:

- SimpleAnalizer: divides text at non-letter characters and puts text in lowercase.
- StandardAnalyzer: tokenizes text based on a sophisticated grammar that recognizes: e-mail addresses; acronyms; Chinese, Japanese, and Korean characters; alphanumeric; puts text in lowercase; removes stop words.
- StopAnalyzer: removes stop words (not useful for searching) and puts text in lowercase.
- WhitespaceAnalyzer: splits tokens at whitespace.

Adding data to an index

Lucene indexes are made of two main concepts: Fields and Documents.

A *Field* represents a piece of data queried or retrieved in a search. The *Field* encapsulates a field name and its value. Lucene provides options to specify if a field needs to be indexed or analyzed and if its value needs to be stored.

And a *Document* is a collection of fields.

Indexing a text file involves wrapping the text data in fields, creating a document, populating it with fields, and adding the document to the index. Lucene also supports boosting documents and fields, which is a useful feature if you want to give importance to some of the indexed data.

2. Searching indexed data

Searching is the process of looking for words in the index and finding the documents that contain those words. Building search capabilities using Lucene's search API is a straightforward and easy process.

Lucene users can request Lucene index by submitting queries to the Lucene search API. The different types of queries supported are described below:

- TermQuery: the most basic query type for searching an index. TermQuery can be constructed using a single term. The term value should be case-sensitive, but this is not entirely true. It is important to note that the terms passed for searching should be consistent with the terms produced by the analysis of documents, because analyzers perform many operations on the original text before building an index. For example, consider the e-mail subject "Job openings for Java Professionals at Bangalore." Assume you indexed this using the StandardAnalyzer. Now if we search for "Java" using TermQuery, it would not return anything as this text would have been normalized and put in lowercase by the StandardAnalyzer. If we search for the lowercase word "java," it would return all the mail that contains this word in the subject field.
- RangeQuery: you can search within a range using RangeQuery. All the terms are arranged lexicographically in the index. Lucene's RangeQuery lets users search terms within a range. The range can be specified using a starting term and an ending

term, which may be either included or excluded. For example, you could search for dates from 01/01/2002 to 02/02/2003.

- BooleanQuery: you can construct powerful queries by combining any number of query objects using BooleanQuery. For example, you could search for emails containing the word “apple” but not the word “computer”.
- PhraseQuery: you can search by phrase using PhraseQuery. A PhraseQuery matches documents containing a particular sequence of terms. PhraseQuery uses positional information of the term that is stored in an index. The distance between the terms that are considered to be matched is called *slop*. By default the value of slop is zero. PhraseQuery also supports multiple term phrases. For example, you could search for a document containing exactly the following sentence “a big apple”.
- WildcardQuery: a WildcardQuery implements a wild-card search query, which lets you do searches such as arch* (letting you find documents containing architect, architecture, etc.). Two standard wild cards are used: “*“ replacing zero or more characters, “?“ replacing one character. There could be a performance drop if you try to search using a pattern in the beginning of a wild-card query, as all the terms in the index will be queried to find matching documents. To avoid this, Lucene forbid the use of wildcards as first character of a search.

4.3. Integrating Lucene with other architecture components

There are two main search engine integration libraries based on Lucene: Compass [2] and Hibernate search [36]. Hibernate Search is closely coupled with Hibernate ORM so it will be impossible to choose another ORM library if needed. Moreover, it doesn’t provide a good transaction management and no cache support, implying robustness and performance problems. Compass provides this type of features. Compass is clearly more mature than Hibernate Search so it appears as the best solution to integrate Lucene full text search capabilities to the Service Discovery component.

Compass provides a breadth of features geared towards integrating search engine functionality. The next diagram shows the different Compass modules, followed by a short description of each one.

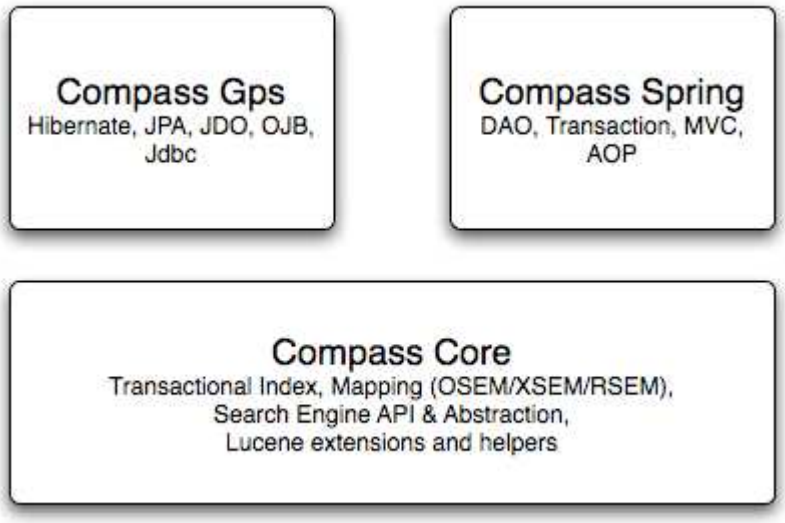

Figure 3: Overview of Compass

Compass Core is the most fundamental part of Compass. It holds Lucene extensions for transactional index, search engine abstraction, ORM (Object Relational Mapping) like API,

transaction management integration, different mappings technologies (OSEM, XSEM and RSEM), and more. The aim of Compass core is to be usable within different scenarios and environments, and simplify the core operations done with a search engine.

To be able to search for documents/resources, the full-text search engine has to build a Lucene index from these different resources. Compass provides three different mapping technologies depending on the type of resource you have to index:

OSEM (Object/Search Engine Mapping) provides the ability to map Java Objects to the underlying Search Engine using Java 5 Annotations or simple XML mapping files. OSEM provides a rich syntax for describing Object attributes and relationships. The OSEM files/annotations are used by Compass to extract the required property from the Object model at run-time and inserting the required meta-data into the Search Engine index.

XSEM (Xml/Search Engine Mapping) provides the ability to map XML structure to the underlying Search Engine through simple XML mapping files. XSEM provides a rich syntax for describing XML mappings using Xpath expressions. The XSEM files are used by Compass to extract the required xml elements from the xml structure at run-time and inserting the required meta-data into the Search Engine index.

RSEM (Resource/Search Engine Mapping) is used when no specific domain model is defined for the application. A resource mapping definition needs to be defined for "types" of resources, with at least one resource id definition (a resource must be identifiable).

Compass Gps aim at integrating with different content sources. The prime feature is the integration with different ORM frameworks (Hibernate [8], JPA [38], JDO [9], OJB [10]), allowing for almost transparent integration between a search engine and an ORM view of content that resides in a database. Other features include a Jdbc integration, which allows to index database content using configurable SQL expression responsible for extracting the content.

Compass Spring integrates Compass with the Spring Framework. Spring, being an easy to use application framework, provides a simpler development model (based on dependency injection and much more). Compass integrates with Spring in the same manner ORM Frameworks integration is done within the Spring Framework code-base. It also integrates with Spring transaction abstraction layer, AOP support, and MVC library.

Dragon Service Registry is based on Hibernate ORM technology and Spring Framework. So, Compass has appeared as the best solution to integrate a search engine in Dragon.

4.4. Full Text-based Service Discovery based on Dragon Service Registry

Classical UDDI based service registries provide service discovery API based on poor SQL based search API. More, they don't allow to search on unstructured documents (PDF or HTML specification, test reports, etc. written in a natural language); they only support search on pre-defined categories (like NAICS [39] or UNSPSC [40]). Dragon, a high performance SOA Governance solution, aims at filling these lacks by including more sophisticated search components described in the previous sections. One of the critical components of the Dragon solution is the registry/repository. The registry allows storing service metadata like its name, purpose, category, provider, user etc. The repository stores raw service related documents like WSDL files, HTML pages or PDF describing the service etc. Registry/Repository provides search API based on full text search engine API to allow users (human or not) discovering services and related documents.

4.4.1. Dragon data model

Dragon data model is based on the CBDI [14] Meta Model for SOA v2. This meta model is not only about Services but about SOA in its entirety.

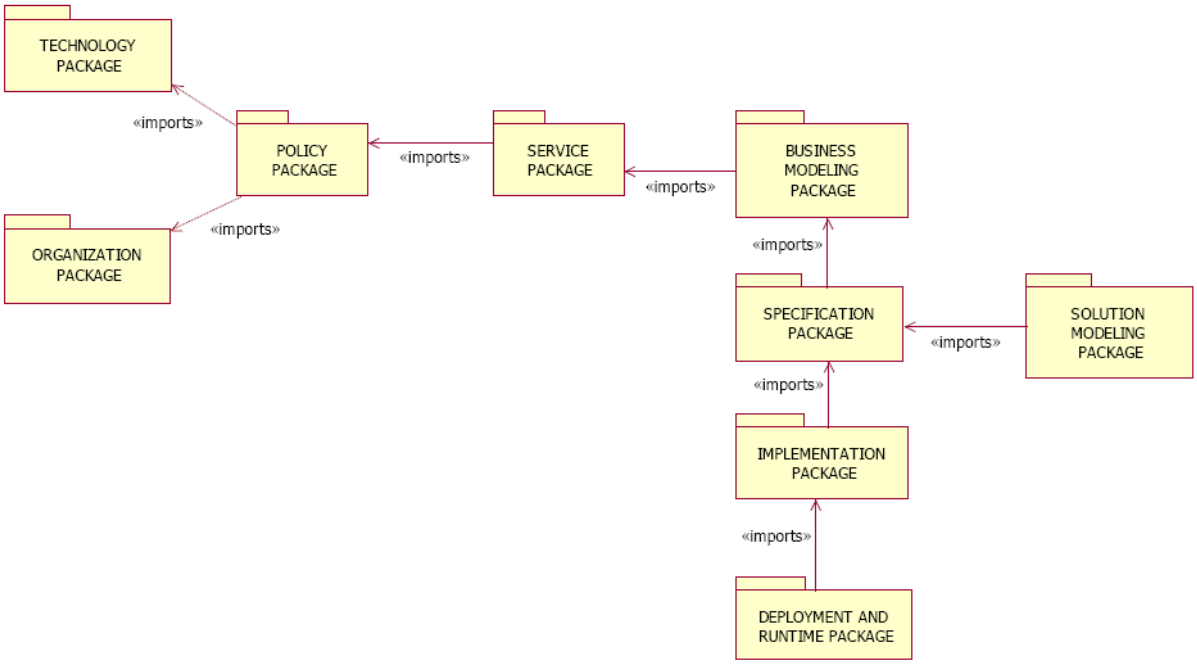

Figure 4: CBDI Data model packages

Here is a description of each package:

- **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.
- **Specification package**: provides a way to model service specification including operations, dependencies, versions...
- **Implementation package**: defines the notion of Deployable Artifacts, packaged as Automation Units that support a particular Service, Application or Use Case.
- **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.
- **Deployment and Runtime package**: defines a way to model service Endpoints running on specified Execution Environments.

So, when considering the notion of service in Dragon we describe not only its specification but also its implementation, its runtime representation, its links with organizations and persons (consumer, provider, owner, manager, etc.), the business processes in which it's involved, etc. But 'Service' is clearly a core concept of Dragon data model.

4.4.2. Service Registration

Nowadays, it is possible to register services in Dragon registry/repository by providing its WSDL description, a crawling archive file containing service descriptions or the address of a

REST service providing service description. The last two capabilities are particularly interesting in the SOA4All point of view because they allow populating a Service registry from crawled data, with customizable full-text based indexation and search. The registry population process from a WSDL is described below:

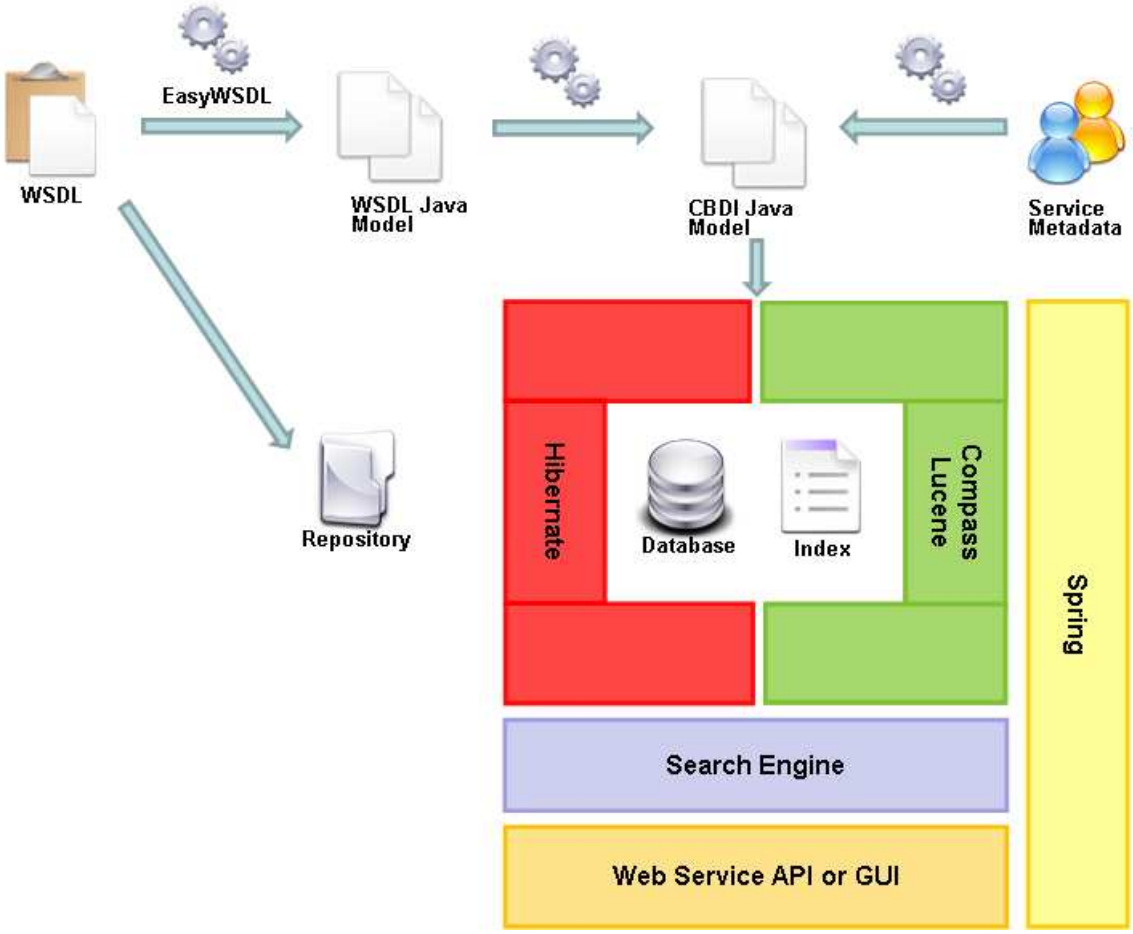

Figure 5: WSDL registration process

When a WSDL must be registered in Dragon, it is stored 'as is' in the Dragon repository for further retrievals. Then, it is converted according to the EasyWSDL[3] library into a Java model. From this, all interesting part of the WSDL Java model are extracted to populate the CBDI based Dragon data model. The Dragon data model can be enriched by human provided meta-data (like tags, comment, etc.). This data model is persisted in a relational database thanks to Hibernate and a Lucene index is automatically created thanks to Compass. Dragon provides an advanced search API that uses Lucene index to retrieve services and related WSDL descriptions.

For the moment, only OSEM mapping is used to populate the Lucene index from the CBDI Java model, because all interesting parts of the WSDL description are mapped to this model. Database and index mapping customizations are described in the next section.

[3] EasyWSDL is a powerful WSDL parsing library; it can be used to parse both WSDL 1.1 and WSDL 2.0 descriptions and manipulates them in an unified object model (based on the WSDL 2.0 entities)

4.4.3. Database and Index Population Customization

Hibernate uses JPA annotations to map POJOs (Plain Old Java Objects) to tables in a relational database. It also provides some additional annotations to handle some cases not supported by the Sun specification like Java Enum mapping.

Compass also provides a set of annotations to map POJO to Lucene index.

Here is a code snippet illustrating POJO to database and index mapping:

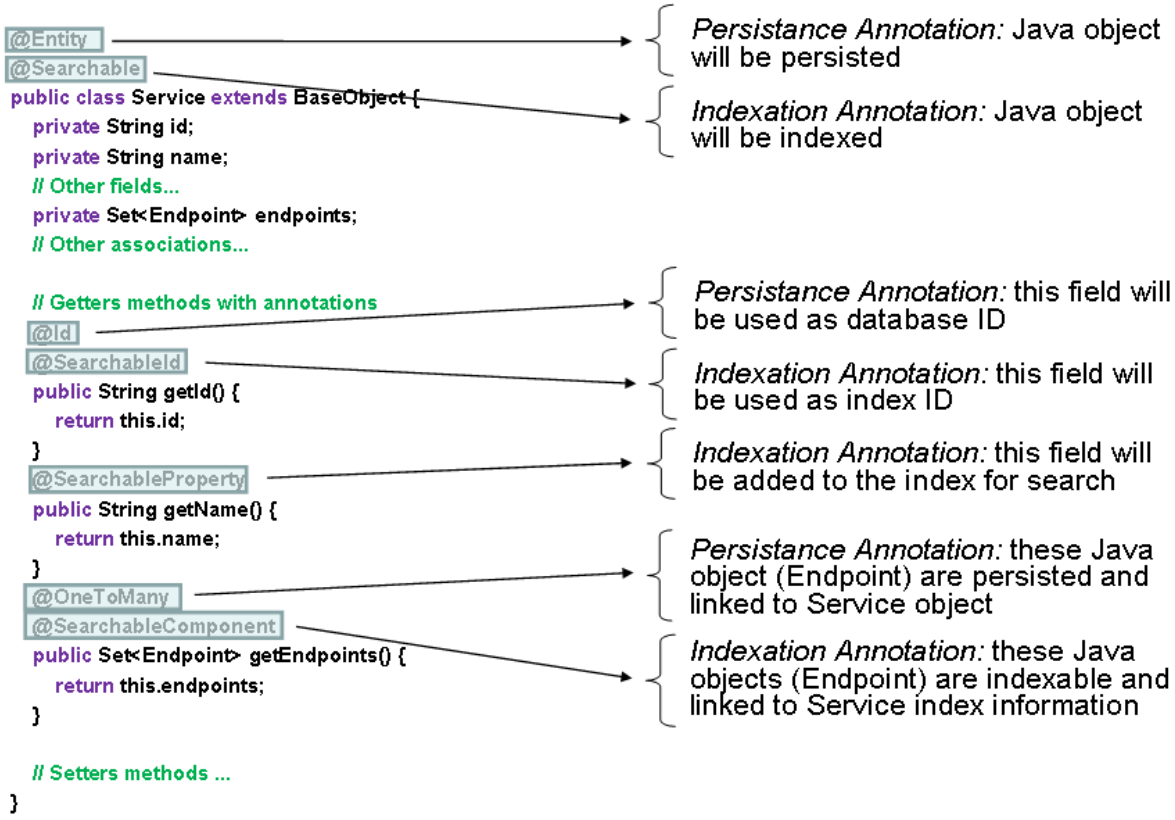

Figure 6: Hibernate and Compass annotations

With this type of annotations it could be very easy to optimize the way Dragon persist and index information extracted from service descriptions and related documents provided by the crawler engine.

4.4.4. Searching for services in Dragon Registry

Once registration and indexation processes done, Dragon provides a search API and a GUI based on indexed data to retrieve services and related document. Here is a snapshot of the GUI:

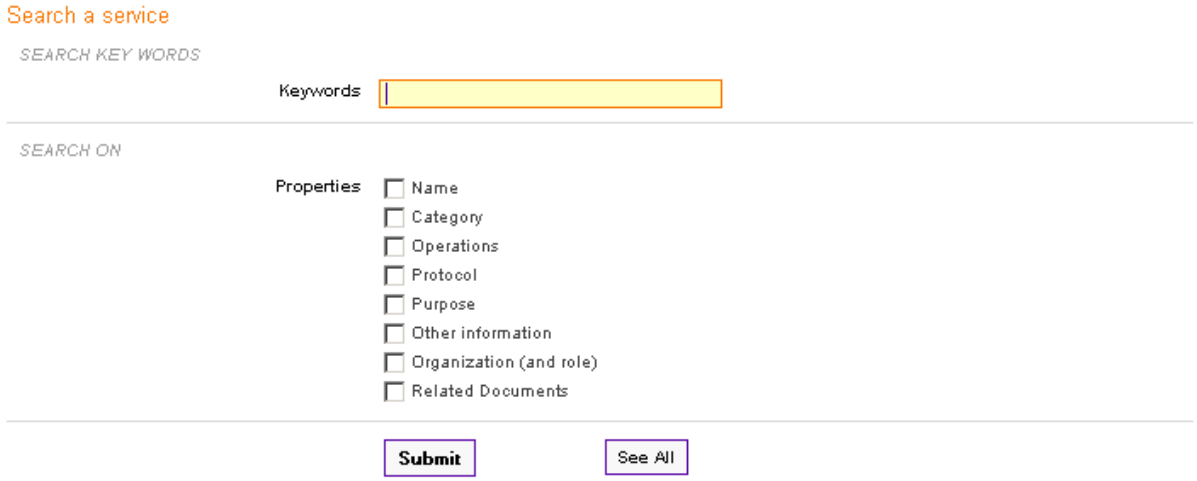

Figure 7: Service Search GUI

Dragon API provides methods to search services by providing a list of properties and a query (containing searched keywords). For example you can search a "Weather" service by simply providing the "Weather" key word. In this case, Dragon searches the key word in all indexed service properties (like service name, service operations, available protocols of the different endpoint of the service, etc.). You can also restrict search by specifying the properties you want to search. For example: you could search the keyword "Weather" on the property "service name" only.

The scope of a Dragon search is larger than the scope of a UDDI search. For example, you could search on names and descriptions of service operations. For service related documents, you could search not only on document description, like for a classical UDDI registry, but also on their whole content.

Like for UDDI registries, you could use wildcards with keywords: the “*” character replace multiple consecutive characters (“test*” could match “tests” or “tester”) and the “?” replace a single character (“te?t” could match “test” or “text”). But, thanks to Lucene integration, Dragon provides more sophisticated search capabilities like fuzzy search, using the “~” on keywords. Fuzzy search allows searching for terms similar in spelling to a given term (“roam~” could match “foam” or “roams”). Fuzzy search uses the Levenshtein Distance algorithm [41] to compare terms.

Dragon API is accessible through the GUI and through a Web Service. For the needs of the SOA4All project, some methods have been added to the original Dragon Web Service API. Originally, Dragon provides methods to register services from WSDL files and to register related document from attached documents. Service descriptions and related documents provided by the crawler engine are packaged as archive files. Some methods have been added to the Dragon API to support service registration from archive files. These methods use an archive reader based on the libraries provided by the Heritrix project [42] to extract WSDL descriptions and service related documents from the archive provided by the seekda crawler and then call the Dragon original API to import them into the service registry. The search method remains the same as the one provided by the original Dragon API. These methods are described below:

- Import operations:
    - importServicesFromArcFile(List<String> arcFileUrls, String services2wsdlsUrl, String services2relatedUrl) : List<String>

This method extract WSDL files and other service related documents from the given archive files, using information provided by index files (services2wsdls and services2related). It

populates registry database with new Services, Endpoints, Interfaces etc. extracted from WSDL files and registers service related documents into the repository. It returns a list of created service identifiers.

- importServicesFromCrawlService(String crawlServiceEPAddress, int start, int count) : List<String>

This method retrieves WSDL files and other service related documents by calling the REST service denoted by the given crawl service endpoint address (crawlServiceEPAddress), starting from the given index (start) and ending at the given index (start + count). It populates registry database with new Services, Endpoints, Interfaces etc. extracted from WSDL files and registers service related documents into the repository. It returns a list of created service identifiers.

- Search operations:
  - searchServices(String searchCriteria, List<String> searchedProperties) : List<String>

This method retrieves a list of services corresponding to the search criteria (a list of searched key words) and list of properties to search on (name, purpose, related documents content etc.). It returns a list of service identifiers.

These methods allow processing documents resulting from the SOA4All crawler component and searching them through a search API based on full text indexing.

5. Semantic Discovery

In the previous section, we have seen how Web services can be discovered by performing full text-based search on their WSDL descriptions and related documents. In this section, we describe the mechanisms for semantic discovery of Web services. These mechanisms allow the users to automatically find the Web services which offer the desired functionality.

The problem raised in Section 1.1 requires a solution for the discovery of services. We address the solution to this problem via three main mechanisms designed to cover semantic discovery.

The first mechanism is a classification-based discovery, which makes use of the classes provided by WSMO-Lite language for service annotation. The second approach does not consider classes anymore, but takes into consideration for discovery the pre-conditions and effects of the service description. We design the algorithm which is founded on these two service properties, that is pre-conditions and effects, in order to provide a discovery mechanism for the services. The last approach within the semantic discovery scope is a hybrid approach that combines both classification mechanism and functionality-based mechanism.

For each of these approaches, a discussion on the performance, complexity and the respective strengths and weaknesses is given, providing a comparative viewpoint on the introduced algorithms.

5.1. Classification-based Discovery

The first Web service discovery approach presented in this deliverable is the most simple and efficient one. In the following, we present the main principle of the classification-based discovery, explain the concept also by an examples and, after addressing its complexity, we discuss the pro and cons of this approach.

5.1.1. Discovery Algorithm

The WSMO-Lite ontology for the description of Web services provides the concept of a functional classification to classify Web services by annotation. WSMO-Lite Web service annotations are assigned to classes of a hierarchy. By this, it becomes possible to quickly query services that promise certain functionality or behavior that is expressed by the assignment to a class.

Classes are identified by meaningful names, such as "*BookSellingService*" or "*TravelService*". A user, who describes a service semantically using WSMO-Lite, assigns this service to these classes that describe the functionality of the service as precise as possible. Once a service is semantically annotated and the semantic service description is published to a service description repository, a user can use class names to express a goal, i.e., to express the desired functionality of a Web service that he or she is looking for. A discovery algorithm that utilizes the functional classification provided by WSMO-Lite service descriptions can easily return services that are annotated with the class names that the user described in the goal.

In the following, we assume that there is a set $\mathcal{W}$ of WSMO-Lite Web service annotations available in a repository. That is, each Web service description $W \in \mathcal{W}$ is described by instances and corresponding relations of the concepts and relation types in the WSMO-Lite ontology. We also assume that the services have been already assigned to classes from a

given classification hierarchy.

Initialization Phase: When a Discovery Engine is started, it will connect to a repository of service annotations, load the annotations and initialize an instance of the reasoner with the annotations.

Discovery Phase: A user sends a goal G that is a set of classes selected from the functional classification hierarchy. Let this set of classes be denoted with C. Each class $c \in C$ of a goal G represents a class to which a Web service should be assigned to in order to be considered a match for the goal G. The discovery engine analyzes the goal G and constructs a query Q for $C = \{c_1, \ldots, c_n\}$ as

$$Q = ?ws\ memberOf\ Service\ and\ ?ws\ modelRef\ ?category\ and$$
$$?category\ subConceptOf\ ?ClassificationRoot\ and$$
$$?category\ subConceptOf\ ?c_1\ and\ \ldots\ and\ ?category\ subConceptOf\ ?c_n$$

The syntax of the query Q corresponds to the WSML query language. Of course, it is possible for the discovery engine to create the query in a different syntax if another reasoner should be used. In the example WSML query above, the property $modelRef$ is inspired by SAWSDL and is used to model the assignment of a Web service to its functional classification.

The discovery engine sends the query Q to the reasoner instance, which then executes the query. The query Q as stated above is interpreted as in the following Table 1.

From all available Web service descriptions $W \in \mathcal{W}$,	$?ws\ memberOf\ Service\ and$
the categories $?category$ specified in the $modelRef$ property,	$?ws\ modelRef\ ?category\ and$
which contain classes for the functional classification,	$?category\ subConceptOf$ $?ClassificationRoot\ and$
must be annotated with the class c_1 or a sub class of c_1, and	$?category\ subConceptOf\ ?c_1\ and$
…, and	$\ldots and$
must be annotated with the class c_n or a sub class of c_n.	$?category\ subConceptOf\ ?c_n$

Table 1: Exemplification of the query Q.

In the result of the query Q, the variable $?ws$ is bound to references to service annotations that fulfill the query Q, namely, annotations of all services that are a member of each of the class selected by the user. For detailed information on the reasoning process, we refer to http://tools.sti-innsbruck.at/wsml2reasoner/ and corresponding deliverables in WP3.

The meaning of a goal that contains a set C of selected classes is that a service $W \in \mathcal{W}$ must be annotated with all of classes in C in order to match the query Q derived from the goal G. The presented classification-based discovery approach can be extended such that a user may also formulate goals with disjunctive and conjunctive selection of classes. Furthermore, the exclusion of classes by the introduction of a negation allows increasing the expressiveness of goals formulated by users. However, the complexity of matching Web service annotations against queries increases with increasing expressiveness.

5.1.2. Example

The following example illustrates the classification-based discovery approach and reveals the benefits of a classification of Web services provided by the WSMO-Lite ontology.

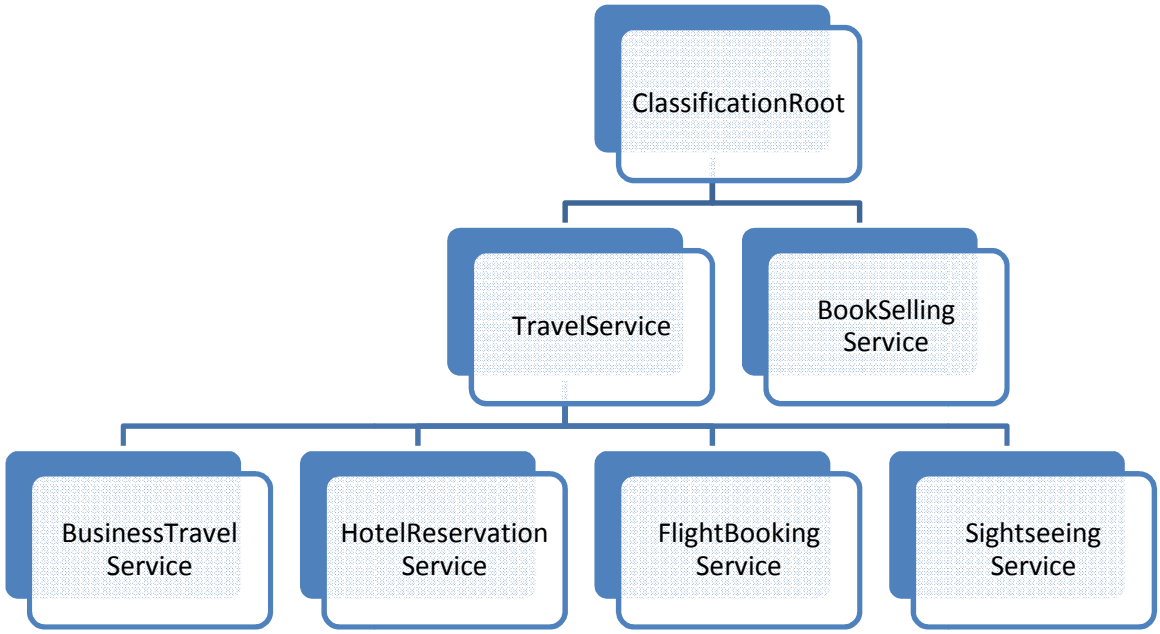

Figure 8: Example classification.

Example Service	Assigned Classes
AcmeFlightUnlimited	BusinessTravelService, FlightBookingService
AcmeHotelsCalifornia	BusinessTravelService, HotelReservationService
AcmeLowAltitudeFlights	SightseeingService, FlightBookingService
AcmeBizOuting	BusinessTravelService, FlightBookingService, HotelReservationService

Table 2: Classification of example Web services.

Consider a user who wants to book a business trip including hotel reservation and flights. The user specifies a goal by the selection of classes “BusinessTravelService”, “HotelReservationService”, and “FlightBookingService”. A given hierarchy on the functional classification, like the one depicted in Figure 8, considers TravelService as super class of BusinessTravelService, HotelReservationService, and FlightBookingService.

The WSML Reasoner provided by WP3 returns those services in the service description repository that match against the query, i.e., service that are annotated with all the classes “BusinessTravelService” and “HotelReservationService” and “FlightBookingService”. This means that the returned services are capable to book a hotel along with a flight. Consequently, the services named AcmeFlightUnlimited and AcmeHotelsCalifornia are not returned as search results since they do not offer both hotel reservation and flight booking. Only the service AcmeBizOuting that was assigned to all three classes specified in the goal

is returned as a result.

As the services have to be annotated with “BusinessTravelService”, they promise to consider issues which might occur in the specific context of business trips. For instance, such classification prevents the discovery and usage of sightseeing service, which might also offer to book a flight and reserve a hotel room at once.

5.1.3. Performance

The proposed classification-based Web service discovery approach is less complex than the subsequent two approaches. We will discuss their complexity in the following.

The classification-based discovery examines each Web service description $W \in \mathcal{W}$ that is available in the repository. Then, the reasoner retrieves each category from the description W, which is a sub concept of the classification root, in constant time. We therefore refer to the query Q shown and explained above. Let the user select the classes $C = \{c_1, \ldots, c_n\}$ to formulate the goal. Then, for each class $c \in C$, the reasoner checks for concept satisfiability[4] with a complexity $\mathcal{O}(x) = \mathcal{O}\big(f(|KB_{\mathcal{C}}|)\big)$ that can not be further specified here since it highly depends on the implementation of the reasoner. The complexity of the reasoner depends on the size $|KB_{\mathcal{C}}|$ of the knowledge base of a classification $\mathcal{C}$ that represents a concept hierarchy, i.e., the terminological box (TBox) of a description logic; the classification does not feature an assertional box. Note, the size of the knowledge base, and thus, the complexity $\mathcal{O}\big(f(|KB_{\mathcal{C}}|)\big)$ does not depend on the number of Web service descriptions in the repository.

The overall complexity of this approach is summarized by $\mathcal{O}(|\mathcal{W}| \cdot |C| \cdot f(|KB_{\mathcal{C}}|))$. Here, in the scope of the present deliverable, we do not provide any evaluation of this service discovery approach. Also, the following two approaches are not evaluated. This is planned to be done in the future of the project.

5.1.4. Discussion

Strengths Classification-based discovery offers a straightforward transformation of the set of classes specified by a user into a query, which can be answered by an ontology reasoner like WSML2Reasoner. A conjunctive query can be answered quite fast, depending on the optimization techniques used within the ontology reasoner.

At the same time, a user interface that allows for selection of classes can be very simple, e.g., a set of keywords, which can interpreted as classes of the classification hierarchy. This approach has the potential of being easily adopted, since common users of the Web are used to such keyword-based search from engines such as Google. It is also possible to let users select the class names, which are arranged in a structure, like a tree, in order to exhibit the hierarchical structure.

Weaknesses The classification hierarchy only consists of class names, which do not any guidance on how to comprehend them. For instance, users may have different understanding of the same term or may use different terms for describing the same fact. Such lack of meaning of the classes leads to the problem of incomprehensibility of the Web service discovery. The user can not know beforehand whether a certain Web service belongs to a

[4] Because of the sub class of relationship between functionality classes, instance checking of the classes pointed at with the “modelRef” element is not sufficient. A class C is sub class of a class D, if and only if $C \wedge \neg D$ is not satisfiable.

specific class or not. In the SOA4All context, it is important to search for Web services at least on the basis of their functionalities.

We summarize the weaknesses of this approach in the main following points:

1. Presenting classes only by names and not attaching a semantic meaning to them causes the problem of ambiguousness in the discovery process.
2. The relation between the classes is not clearly defined, based on formalized semantics (i.e., it is possible to model the class "*TravelService*" as a sub class of the "*BookSellingService*" class). That is, it is not formally clear what an aforementioned "sub class of"-relationship between classes mean. For instance, a sub class may represent more specific or less specific services. However, without a common understanding of the functionality of services, it is not possible to differentiate between them.
3. The classification hierarchy is built and maintained manually as there is no formalism behind the class names that enables automation. Thus, inconsistencies in the hierarchy may occur. The lack of a formal semantics also makes it difficult for users to decide to which classes a specific Web service should belong to. The meaning of the classes can be described in natural language. Apart from the fact that natural language descriptions often lead to ambiguities, the lack of formal meaning of classes makes it impossible to automatically classify Web services correctly and keep the classification hierarchy, as well as the service classification, consistent. For instance, a service can be assigned to "*BookSellingService*", "*TravelService*", and "*WeatherService*", simultaneously, which obviously does not make sense in a real world setting.

5.2. Functionality-based Discovery

In this section, we study the second discovery approach that is based on the functionality of Web services, starting with a definition of Web service functionality. Then, we describe actual as well as the desired functionality of Web services by examples in the second part of this section. It is shown how both functionality descriptions can be matched in order to discover Web services that offer the desired functionality. After a comparison of the complexity with the previous approach, we conclude by discussing the strength and weaknesses of this discovery approach.

Besides the potential of the previous approach to scale with ever increasing number of available Web service descriptions, the main disadvantage is the absence of formal semantics of the classification hierarchy. For example, this lack disallows an automatic assignment of Web service descriptions to appropriate classes. Therefore, the second approach provides a formalism to describe the functionality of Web services and, thus, overcomes the shortcomings of the classification-based discovery approach.

5.2.1. Functionality Description of Web Services

In order to enable automation on the classification of Web services, we need a formal model that allows for automatic comprehension of the classification. I.e., classes and the relationship between need a formal semantics since this features a common understanding. Remember, in case of textual descriptions of class names, the interpretation of classes as well as the relationship between them may vary for different users.

We formally describe the functionality of Web services and let classes represent functionalities. Within such a classification, it is possible to formally describe a “sub class of” relationship between classes and thereby obtain automation for the classification of services.

Definition 1 (Functionality Description) The (actual and desired) functionality of a Web service is defined as a tuple $(\mathrm{I}, \mathrm{O}, \phi, \psi)$, with I the set input variables, O the set output variables, ϕ a logical expression describing the pre-condition, and ψ a logical expression describing the effect.

The offered functionality of Web services is regarded as a sequence of an input activity, a local operation, and an output activity. The input activity receives values for the input parameters from a user or a client and binds them to some variables in I. The local operation may use the values of the input variables and calculates the output by, e.g., querying or updating the knowledge base of the Web service. The output operation binds the result of the local operation to the variables included in O and sends the values of the output variables O to the user. The relationship between input variables, output variables, and further optional variables or constants is described by the effect ψ. The conditions that must hold in the knowledge base, before a service is invoked, are described by the pre-condition ϕ.

Below, we illustrate the description of pre-conditions and effects. The service offering and a user specified goal may vary and, hence, we distinguish between actual and desired functionalities, respectively. The following example considers a book selling Web service, which is described from the perspectives of service providers and prospective consumers. The sets of input and output variable names are omitted.

Example 1. Actual Functionality Description Upon receiving the input values from a user, the book selling service binds the values to the variables o, i, c, and e, respectively. The pre-condition

$$\begin{aligned}\phi_{\text{actual}} = BookOrder(o) \wedge hasISBN(o,i) \wedge ISBN(i) \wedge hasCreditCardDetails(o,c) \\ \wedge CreditCardDetails(c) \\ \wedge (hasCardType(c,\text{"VISA"}) \vee hasCardType(c,\text{"Amex"})) \wedge hasExpiration(c,e) \\ \wedge AtLeast3MonthsInFuture(e)\end{aligned}$$

of the service means that the offered service expects a book order o with an ISBN i of the book to be ordered and the availability of credit card details of the card c used for the book order. Furthermore, the credit card c should be of type "VISA" or “Amex” and have an expiration date e that lies at least 3 months in the future.

The effect ψ_{actual} of such example book ordering service could be described by the following expression.

$$\begin{aligned}\psi_{actual} = OrderConfirmation(oc) \wedge hasOrderedGood(oc,d) \wedge hasDetails(i,d) \\ \wedge hasPrice(i,p) \wedge hasAmount(oc,p) \wedge paymentBy(oc,c)\end{aligned}$$

The effect expresses that the offered service outputs an order confirmation oc with details d about the ordered book i (i is bound to the input value). These details are the same as the details of the book with the ISBN i entered by the user. The service further outputs the total amount p of the order that equals the price p of the ordered book and the payment details that are the same as the payment details of the credit card c entered by the user.

Actual and desired descriptions of Web service functionalities do not differ in their syntactical description. We interpret the desired functionality similar to actual functionality. In the case of desired functionalities denoted by $(\mathrm{I}, \mathrm{O}, \phi, \psi)$, I, O, ϕ, and ψ describe the *desired* inputs, *desired* outputs, *desired* pre-condition, and *desired* effect, respectively.

Example 2. Desired Functionality Description The following desired functionality description of the example book selling service is not explained in detail. It still reveals potential differences between actual and desired service functionality descriptions.

The prospective user of a book ordering Web service may formulate the following desired pre-condition ϕ_{desired} by

$$\begin{aligned}\phi_{\text{desired}} = JournalOrder(jo) \wedge hasISBN(jo,i) \wedge ISBN(i) \wedge CreditCardPayment(jo,c) \\ \wedge hasCardType(c,\text{"VISA"}).\end{aligned}$$

Accordingly, the desired effect ψ_{desired} of the service might be described as follows.

$$\begin{aligned}\psi_{\text{desired}} = OrderConfirmation(oc) \wedge hasDetails(i,d) \wedge hasOderedGood(oc,d) \\ \wedge paymentBy(oc,c)\end{aligned}$$

The differences between actual and desired pre-condition and effect shown by the two examples also justifies the application of semantics for discovery in general. A discovery algorithm must handle different granularities and amounts of input and output variables, pre-conditions, and effects. That is for example the difference between the concepts of $BookOrder$ and $JournalOrder$, the solely one-sided description of the credit card’s expiration date. Therefore, a domain knowledge base is required to model different concepts and the

relations among them. The domain knowledge base should also provide a classification of the input and output types.

5.2.2. Discovery Algorithm

For a given set $\mathcal{W}$ of Web service functionality descriptions and a given desired functionality description G (goal), the task of the discovery algorithm is to find those descriptions in $\mathcal{W}$ that offer the desired functionality expressed by G.

We first establish criteria that need to be evaluated in order to determine whether a desired functionality G matches a Web service functionality description $W \in \mathcal{W}$. For the sake of simplicity, we also let W refer to the Web service instead of its functionality description. Additionally, the indexes of the elements of functionality descriptions reveal the affiliation to a description of a goal G or a Web service W.

- The set of desired inputs I_G of a goal specifies which inputs a user wants to provide, whereas the set of inputs I_W describes which inputs the service W requires. If W requires less inputs than a user expressed in the set of desired inputs I_G, then W should not match the goal G. A Web service $W \in \mathcal{W}$ is a match only if it accepts all the desired inputs I_G a user intends to provide.
  That is, for every desired input $i_G \in I_G$, there should be a distinct actual input variable $i_W \in I_W$, such that the type $T(i_W)$ equals or is a super class of the type $T(i_G)$ of i_G. It is also possible that a service may require more inputs than the user specified in the desired functionality description G.
- The set of desired outputs O_G of a goal specifies which outputs a user wants the Web service to deliver, whereas the set of outputs O_W describe which outputs W actually delivers. A Web service W is a matches the goal G only if the service W delivers all desired outputs O_G. That is, for every desired output $o_G \in O_G$, there must be a distinct output variable $o_W \in O_W$, such that the type $T(o_W)$ of o_W equals or is a sub class of the type $T(o_G)$ of o_G.
  Note, that the goal semantics as provided here, interprets the specified inputs and outputs as compulsory, similar to the keywords provided to state-of-the-art Web search engines. Hence, a matching service may require more inputs and may offer more outputs.
- In order to let a Web service description W match a goal G, the actual pre-condition ϕ_{W}, and effect ψ_{W} of a Web service $W \in \mathcal{W}$ must imply desired pre-condition ϕ_{G} and desired effect ψ_{G}, respectively.

Checking whether a formula $A \Rightarrow B$ holds is equivalent to checking whether $\neg A \vee B$ is satisfiable. Depending on the logic that is chosen to express pre-conditions and effects, the complexity of the satisfiability problem may vary. However, description logics that are expressive enough for practical purposes have exponential complexity in the length of logical expressions.

Require: Goal $G = (\mathrm{I_G}, \mathrm{O_G}, \phi_{\mathrm{G}}, \psi_{\mathrm{G}})$, set of Web services $\mathcal{W}$
Initialize the result set $R = \emptyset$
for all $W \in \mathcal{W}$ **do**
  Compute the set of injective functions Σ such that each $\sigma \in \Sigma$ maps variable names in $\mathrm{I_W}$ to those in $\mathrm{I_G}$ while considering subsumption among their types.
  Compute the set of injective functions Ω such that each $\omega \in \Omega$ maps variable names in $\mathrm{O_W}$ to those in $\mathrm{O_G}$ while considering subsumption among their types.
  for all $\sigma \in \Sigma$ **do**
    for all $\omega \in \Omega$ **do**

if $\sigma(\phi_{\mathrm{W}}) \Rightarrow \phi_{\mathrm{G}}$ and $\omega(\psi_{\mathrm{W}}) \Rightarrow \psi_{\mathrm{G}}$ **then**
set $R = R \cup \{W\}$

return R

Algorithm 1: Pre-Conditions and Effects-based Discovery Algorithm.

Example 3. Matching Actual and Desired Functionality The functionality-base discovery algorithm as show in Algorithm 1 is able to automatically discover Web service descriptions that match a given goal. In this example, the matching of the actual and desired preconditions and effects of Examples 1 and 2 in Section 5.2.1 is demonstrated.

Let $\sigma \in \Sigma$ and $\omega \in \Omega$ denote the proper variable names mapping functions, then $\sigma(\phi_{\mathrm{actual}}) \Rightarrow \phi_{\mathrm{desired}}$ and $\omega(\psi_{\mathrm{actual}}) \Rightarrow \psi_{\mathrm{desired}}$. In this particular instance, a functionality-based discovery checks whether the implication holds.

$$
\begin{aligned}
&\sigma(\phi_{\mathrm{actual}}) \Rightarrow \phi_{\mathrm{desired}} \\
&\quad \Leftrightarrow (BookOrder(\sigma(o)) \wedge hasISBN(\sigma(o), \sigma(i)) \wedge ISBN(\sigma(i)) \\
&\quad \wedge hasCreditCardDetails(\sigma(o), \sigma(c)) \wedge CreditCardDetails(\sigma(c)) \\
&\quad \wedge (hasCardType(\sigma(c), \text{"VISA"}) \vee hasCardType(\sigma(c), \text{"Amex"})) \\
&\quad \wedge hasExpiration(\sigma(c), \sigma(e)) \wedge AtLeast3MonthsInFuture(\sigma(e))) \\
&\quad \Rightarrow \big(JournalOrder(jo) \wedge hasISBN(jo, i) \wedge ISBN(i) \\
&\quad \wedge CreditCardPayment(jo, c) \wedge hasCardType(c, \text{"VISA"})\big)
\end{aligned}
$$

For simplicity, we do not consider the variable name mapping and variable type subsumption in detail. We just briefly want to mention that the mapping function $\sigma \in \Sigma$ should map the book order variable o to the corresponding journal order variable $jo = \sigma(o)$. Further, the type $T(jo)$ must be a sub type of the book order type $T(o)$. A domain knowledge base is required to subsume the types of the variables.
The check for satisfiability of the implications also requires a domain knowledge base. Each axiom of the desired pre-condition must be implied by the actual pre-condition. For instance, $BookOrder(\sigma(o)) \Rightarrow JournalOrder(jo)$ holds since the concept $JournalOrder$ is modeled as a sub concept of $BookOrder$ in the mentioned domain knowledge base. It is worth noting that the desired pre-condition does not express more than the actual service offers. The expiration date is not considered in the desired functionality description. The implication is still satisfiable. By this principle, the algorithm will recognize the satisfiability of the implication. Having said that, the implication would not hold if, for instance,

- the desired inputs contains a login name in addition that cannot be processed by service described by the actual functionality,
- the $JournalOrder$ in the desired precondition would be replaced by $GoodsOrder$, whereas $BookOrder$ is a sub class of $GoodsOrder$ and a super class of $JournalOrder$ in the given domain knowledge base,
- the desired pre-condition specifies a discount on ordered articles for registered costumers.

Considering the actual and desired effects from the example above, the implication $\omega(\psi_{\mathrm{actual}}) \Rightarrow \psi_{\mathrm{desired}}$ is also satisfiable.

5.2.3. Performance

The functionality-based discovery approach computes the sets Σ and Ω of mapping functions for each available Web service description $W \in \mathcal{W}$ in the repository. The numbers $|\Sigma|$ and $|\Omega|$ of injective mapping functions are determined by the respective number of input and output variables of the goal G and the Web service description W.

$$|\Sigma| = \frac{|I_W|}{(|I_W| - |I_G|)!} \qquad |\Omega| = \frac{|O_W|}{(|O_W| - |O_G|)!}$$

Then, for each mapping of input and output variable from the Web service description to the ones specified in the goal, the reasoner also needs to check the types of the variables. As said, the reasoner has to check whether the type $T(i_W)$ of an input variable $i_W \in I_W$ equals or is a super class of the type $T(i_G)$ of a desired input variable $i_G \in I_G$. Similarly, the type $T(o_W)$ of each output variable $o_W \in O_W$ should equal or be a sub class of the type $T(o_G)$ of a desired output variable $o_G \in O_G$. This check involves a domain knowledge base KB_D that contains a model of the variable types and the relationships between them. The complexity of a such a check is in $\mathcal{O}(f(|KB_D|))$ and for all Web service descriptions and mappings of variable names the complexity sums up to $\mathcal{O}(|W| \cdot |\Sigma| \cdot |\Omega| \cdot f(|KB_D|))$.

Additionally, for each Web service description $W \in \mathcal{W}$ and for each combination of mapping functions $\sigma \in \Sigma$ and $\omega \in \Omega$, the algorithm checks for satisfiability of both implications $\sigma(\phi_{\mathrm{W}}) \Rightarrow \phi_{\mathrm{G}}$ and $\omega(\psi_{\mathrm{W}}) \Rightarrow \psi_{\mathrm{G}}$ (cf. Algorithm 1). This check for satisfiability also requires a domain knowledge base that models the concepts and relationships used to express pre-conditions and effects. In contrast to the previous satisfiability check of the variable name mappings, the complexity of checking whether the implications hold for given pre-conditions and effect further depends on the length of the logical expressions.
The reasoner checks for satisfiability of the logical expressions of pre-conditions and effects. The complexity of a satisfiability check depends on the size $|KB_D|$ of the domain knowledge base and the maximum length M_{PE} of the logical expressions that represent actual and desired pre-condition and effect. The length $|\Upsilon|$ of a logical expression Υ is the number of axioms it contains. The maximum formula length

$$M_{PE} = \max\{\max\{|\phi|: \forall \phi \in (\Phi_{\mathcal{W}} \cup \{\phi_{\mathrm{G}}\})\}, \max\{|\psi|: \forall \psi \in (\Psi_{\mathcal{W}} \cup \{\psi_{\mathrm{G}}\})\}\}$$

is the maximum number of axioms of the logical expressions of any pre-condition and effect of Web service descriptions $\mathcal{W}$ and the goal G. $\Phi_{\mathcal{W}}$ and $\Psi_{\mathcal{W}}$ denote the sets of pre-conditions and effects of all Web service descriptions in $\mathcal{W}$, respectively.

Finally, the overall complexity of the functionality-based discovery algorithm can be summarized as

$$\mathcal{O}\big(|\mathcal{W}|(|\Sigma||\Omega| f(|KB_D|) + g(|KB_D|, M_{PE}))\big)$$

The functions f and g are not specified in this document as they hide the complexity of satisfiability checks that is specific to the used reasoner. Nevertheless, the size of the domain knowledge base and the maximum formula length are identified as principal parameters of those functions.

5.2.4. Discussion

Strengths

The introduction of a formal model that allows describing the functionalities of Web services overcomes the shortcomings of the classification-based discovery approach mentioned in Section 5.1.4. Functionality-based discovery is founded on the formal descriptions of inputs, outputs, pre-conditions and effects of Web services. Therefore, this discovery mechanism

allows the users to search for Web services based on their functionality. Furthermore, since is based on the provided formal semantics, the discovery is comprehensible for all users and also enables a higher degree of automation in the further steps of the Web service usage process, e.g., composition.

Weaknesses

On the other side, this approach lacks efficiency and scalability for a large number of Web services. As we have shown above, the complexity is larger than the complexity of the classification-based approach. This approach is less likely to scale for a large number of available Web service descriptions, because the complexity of satisfiability checks is fairly high for each checked match. As we cannot decrease the complexity of the reasoner, the following third discovery approach will overcome this scalability problem by tremendously reducing the number of satisfiability checks to compute.

5.3. Classification and Functionality-based Discovery

In the previous two sections we have seen that on one side classification based discovery has the potential of scaling up to large number of Web service descriptions but does not consider functionalities of Web services, whereas the functionality based discovery is expressive and less likely to scale. In this section, we present an approach that is a combination of both the previous approaches to achieve a scalable functionality based discovery.

The main idea of this approach is the establishment of a hierarchy of classes representing service functionalities. Functionality classes may have super classes and sub classes meaning that they provide more general or more specific functionality, respectively. The subsequent paragraphs introduce the functionality classification of Web services. Such a classification hierarchy is induced by the definition of a sub class relationship over functionality classes. After the creation of a functional classification hierarchy, Web service descriptions can be attached to the most specific functionality class, which matches against the description of the Web service.

5.3.1. Web Service Functionality Classification

Definition 2. Classification Hierarchy *A functionality class is a named tuple representing a functionality description as defined in Definition 1. Given a set C of such classes, each class $c \in C$ is assigned with a set of super-classes $S(c) \subseteq C$.*

In order to achieve automatic procedures that do semantically correct reasoning with the hierarchy of functionality classes, it is necessary to restrict the super classes $S(c)$ of a functinality class c in a way that the hierarchy does not become inconsistent.

For instance, consider a class that is labeled with "$\mathrm{WeatherForecast}$" and represents the functionality of services providing some weather information to a given location. Additionally, consider another class labeled with "$\mathrm{MovieSearch}$" that provides references to movies related to a given topic. A third class "$\mathrm{WeatherForecastPodcast}$" provides weather forecast video podcast and requires a location and the specification of the video encoding format. It is inconsistent to add the latter class as a sub class of "$\mathrm{WeatherForecast}$" and "$\mathrm{MovieSearch}$" in the hierarchy since the input of the movie topic cannot be inherited to the "$\mathrm{WeatherForecastPodcast}$". Inconsistencies may also occur within the set of outputs, pre-conditions, and effects in a similar manner.

We now define formally the meaning of consistent hierarchy.

Definition 3. Overall Input and Output of a Functionality Class *The overall input I_c^o of a functionality class c is*

$$I_c \cup \bigcup_{c' \in S(c)} I_{c'}^o$$

The overall output O_c^o of a class c is

$$O_c \cup \bigcup_{c' \in S(c)} O_{c'}^o$$

Definition 3 states that a functionality description class c inherits inputs and outputs of its super classes $c' \in S(c)$. That is, the overall inputs of a class contain its inputs and the overall

inputs of all the classes it is subsumed by. The overall outputs of a class are defined analogously.

Definition 4. Overall Pre-condition and Effect of a Functionality Class *The overall pre-condition ϕ_c^o of a class c of functionality class is*

$$\phi_c \wedge \bigwedge_{c' \in S(c)} \phi_{c'}^o$$

The overall effect ψ_c^o of a class c is

$$\psi_c \wedge \bigwedge_{c' \in S(c)} \psi_{c'}^o$$

Definition 4 states that the overall pre-condition of a functionality class is equal to its pre-condition together with the overall pre-conditions of all classes it is subsumed by. The overall effect of a class is defined analogously.

Having the definitions of overall inputs, overall outputs, overall pre-conditions and overall effects, we can now define consistency of classification hierarchy formally.

Definition 5. Consistency of Classification Hierarchy *A subsumption relationship of a functionality class c (sub class) with another class c' (super class) is consistent if the pre-condition of the sub class in conjunction with the overall pre-condition of the super class, i.e., $\phi_c \wedge \phi_{c'}^o$, as well as the effect of the subclass in conjunction with the overall effect of the super class, i.e., $\psi_c \wedge \psi_{c'}^o$, are satisfiable formulas. A classification hierarchy is consistent if every subsumption relationship it contains is consistent.*

Defining classes of the classification hierarchy with tuples of the form (I, O, ϕ, ψ) allows automatic consistency checking of the classification hierarchy and, thus, prevent any inconsistencies by definition.

So far, the introduced hierarchy over desired functionalities was not connected to the actual functionalities of Web service descriptions. A Web service $W \in \mathcal{W}$ is member of a class $c \in C$ within the classification hierarchy, if the following four conditions can be satisfied:

1. $I_W \subseteq I_c^o$
2. $O_W \subseteq O_c^o$
3. $\phi_W \Rightarrow \phi_c^o$
4. $\psi_W \Rightarrow \psi_c^o$

A Web service description $W = (I_W, O_W, \phi_W, \psi_W)$ can be member of a class $c = (I_c, O_c, \phi_c, \psi_c)$ if and only if the overall inputs I_c^o and the overall outputs O_c^o of class c are included in the sets I_W and O_W of the Web service description W, respectively. Furthermore, the overall precondition ϕ_c^o and the overall effect ψ_c^o of class c must be implied by the pre-condition ϕ_W and the effect ψ_W of W, respectively. As a consequence, a Web service that is member of a

class c is also member of all super classes $S(c)$ of c. The check for consistency of Web services assigned to classes can be computed automatically.

5.3.2. Discovery Algorithm

The discovery of Web services can be defined again as the attempt to find those descriptions from the given set of descriptions W that match to the query$Q$. In contrast to the discovery algorithm from Section 5.2, functionality classes are now utilized to formulate queries. Therefore, we will specify queries, define matches of queries against functionalities, and show at the end of this section the improvement on the performance of discovery algorithms based on the classification hierarchy.

The description of actual Web service functionalities is not affected by a classification hierarchy and, thus, is expressed by a tuple (I, O, ϕ, ψ) as in Definition 1. However, a query can contain class labels to ease the formulization and reduce the complexity of a query.

Definition 6. Desired Functionality Description of Web Services *Desired functionality of Web services is expressed by a query Q defined as the tuple $(C_Q, I_Q, O_Q, \phi_Q, \psi_Q)$. C_Q denotes a set of functionality classes, where each class is identified by its label. Set of input variables I_Q, set of output variables O_Q, pre-conditions ϕ_Q, and effects ψ_Q are interpreted analogously as specified in Definition 1.*

In order to define matches of desired and actual functionality descriptions of Web services, we first provide semantics of the desired functionality description. As classes of the hierarchy represent functionalities, the set of classes C_Q of a query Q represents functionalities, which are connected with logical conjunction to the overall functionality of classes. Precisely, a query $Q = (C_Q, I_Q, O_Q, \phi_Q, \psi_Q)$ is interpreted as

$$\left(\bigcup_{c \in C_Q} I_c^o \cup I_Q, \bigcup_{c \in C_Q} O_c^o \cup O_Q, \bigwedge_{c \in C_Q} \phi_c^o \wedge \phi_Q, \bigwedge_{c \in C_Q} \psi_c^o \wedge \psi_Q, \right)$$

The overall pre-condition and effect of classes of a query can be derived directly from the classes in a data structure that implements the hierarchy. After the classes of a query are mapped to the form $(I_Q, O_Q, \phi_Q, \psi_Q)$, we can apply the matching Algorithm 2, which is very similar to Algorithm 1. Here we want to emphasize that, although the matching check procedure is not changed, the number of computed matches can be drastically reduced due to the use of a classification hierarchy.

Algorithm 2 Classification-Based Discovery Algorithm

Require: Query $Q = (\mathcal{C}_Q, I_Q, O_Q, \phi_Q, \psi_Q)$ and a set of Web services $\mathcal{V} \subseteq \mathcal{W}$ that are member of all classes $\mathcal{C}_Q$
Initialize the result set R to $\emptyset$
for all $W \in \mathcal{V}$ **do**
Compute the set of injective functions Σ such that each $\sigma \in \Sigma$ maps variables in I_W to those in I_Q while considering subsumption among their types
Compute the set of injective functions Ω such that each $\omega \in \Omega$ maps variables in O_W to those in O_Q while considering subsumption among their types
for all $\sigma \in \Sigma$ **do**
for all $\omega \in \Omega$ **do**
if $\sigma(\phi_W) \Rightarrow \phi_Q$ and $\omega(\psi_W) \Rightarrow \psi_Q$ **then**
set $R = R \cup \{W\}$
return R

We emphasize here that even if it becomes clear from the semantics of a query, using classes in a query is not a limitation on the expressivity of queries. The use of classes allows reducing the length of the formulas representing pre-conditions and effects as well number of inputs and outputs, thus leading to significantly lower run time complexity of query answering.

5.3.3. Performance

The discovery algorithm presented in this section promises to scale much better against an increasing number of available Web service descriptions, while still preserving the expressivity of the algorithm in the previous section. We now compare the complexity of the two algorithms to show that the algorithm presented in this section is more efficient.

The algorithm presented in this section is a combination of the two algorithms presented in the previous two sections. Therefore, its complexity is the sum of the complexities of the previous two algorithms, i.e.

$$\mathcal{O}(|\mathcal{W}| \cdot |C| \cdot f(|KB_C|)) + \mathcal{O}\big(|\mathcal{V}|(|\Sigma||\Omega| f(|KB_D|) + g(|KB_D|, M_{PE}))\big)$$

The gain in efficiency of the hybrid algorithms based discovery is founded in the following points.

- The reduced number of Web services $\mathcal{V} \subseteq \mathcal{W}$ that need to undergo computationally expensive algorithm from the previous section. This is due to the fact, that the based on the classes in the query, the set of Web services can be drastically limited by classification based discovery.
- The reduced numbers $|\Sigma|$ and $|\Omega|$ of variable mapping functions. The number of variable names of both query and Web service description can be drastically decreased by the use of classes because the overall input and overall output variable names of the classes C_Q are do not need to be specified by the user explicitly in I_Q and O_Q, respectively.
- The reduced numbers of terms of a query's pre-condition ϕ_Q and query's effect ψ_Q that are checked for satisfiability. As a result the term M_{PE} decreases. As we have already stated, the check for satisfiability causes exponential costs against the length of the formula of pre-condition and effect if common expressive logics are applied.

The use of classes in the query can decrease the size of the logical expressions ϕ_Q and ψ_Q significantly, because the overall pre-conditions and overall effects of the classes C_Q do not need to be check again.

5.3.4. Discussion

Strengths

From the expressivity point of view, it is same as the functionality based discovery. However, it does not suffer from high complexity. This approach combines the strengths of the pure classification based discovery and pure functionality in order to achieve better performance on one side, and expressivity on the other side. Furthermore, it combines the previous two approaches in such a way that their weaknesses disappear.

Weaknesses

In order to guarantee the correct functioning of this approach, the classification hierarchy as well service classifications need to be kept consistent. That is, every time a new Web service is published, or the description of a published service is updated, the classification for the new Web service description needs to be computed. Whenever the classification hierarchy changes, the classifications of the services need to be updated accordingly. However, even though these computations are time consuming, they are performed beforehand and not during the query answering time.

5.4. Implementation

In this section, we present the details of the prototypical implementation of the semantic discovery component. The third presented approach, the classification and functionality based service discovery from Section 5.3, was implemented. This hybrid approach covers both the classification based discovery and the functionality based discovery. If the user goal only consists of a set of classes, then the hybrid approach acts like the classification based discovery approach (cf. Section 5.1). If the user goal contains a specification of desired inputs, outputs, pre-condition, effect and does not contain any classes, then the hybrid approach acts like the functionality based discovery approach (cf. Section 5.2). Consequently, the implementation of the hybrid approach is not a restriction to one of the presented approaches.

We first describe the use cases of the semantic discovery service within SOA4All. We describe the interactions with those SOA4All components that are required for the discovery of services, secondly. Then we present the architecture of the discovery service and describe the functionality by its provided methods. This section ends with a description of the graphical user interface of the SOA4All discovery component.

Use Cases. The discovery component is used by users of the Process Editor of the SOA4All Studio, by the Semi-Automatic Composer, the Consumption Platform, and the Provisioning Platform. Although their usage of the discovery component is similar, the differences allow us to specify the following four usage scenarios of service discovery (cf. Figure 9). In this section we will only focus on the usage of the semantic service discovery and omit the full text-based service discovery.

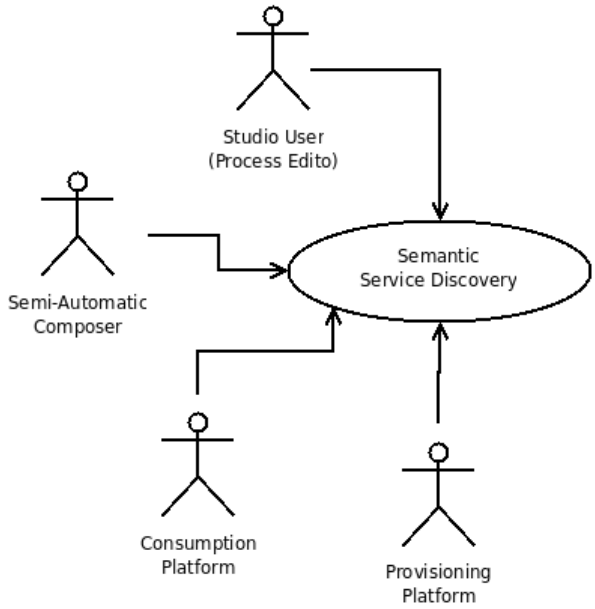

Figure 9: Use cases of the service discovery.

1. A user of the Process Editor composes a process model, either by hand in the free editing mode or guided by the usage of process templates and patterns. The user binds the process activities of the process model to concrete Web services at design time. Therefore the user specifies a goal[5] with the help of the semantic service

[5] Note that the term goal is also used by the LPML (Lightweight Process Modeling Language)

discovery user interface, submits it to the discovery service, and selects a returned Web service operation that will be bound to the process activity. Of course, a user may also use the full text-based service discovery for this task.

2. The composer provided by WP6 also needs to discover Web services. The composer may resolve composite goals and elaborates the contained steps within the goal or optimizes the process against non-functional properties. Technically, this use case is similar to the prior one, but the goal is automatically specified and submitted to the semantic service discovery service. The discovered services are finally used to elaborate the process model by resolving composite goals. In case of optimization, the discovered services may replace the current services bound to process activities if such replacement improves the overall process. (For instance the price of execution is reduced by replacing a service with another one that provides the same functionality.)
3. The Consumption Platform (cf. D2.2.2) invokes and executes processes resolves the LPML-goals at runtime. Therefore, the LPML-goal description is used to specify a discovery goal, which is automatically sent to the discovery service at runtime. The returned services need to be ranked in order to allow an automatic selection of the most appropriate service.
4. The Provisioning Platform as described in D2.1.4 allows annotating semantic information to Web services. If the service to annotate is already available in the documents repository, then the service to annotate can be discovered by using the semantic or the full text-based discovery. Therefore, a user will again use the semantic service discovery GUI to specify the goal. Beyond the scope of SOA4All, it might be also envisioned to automatically annotate services, where the goal for the semantic discovery is generated and committed automatically by a Web service invocation.

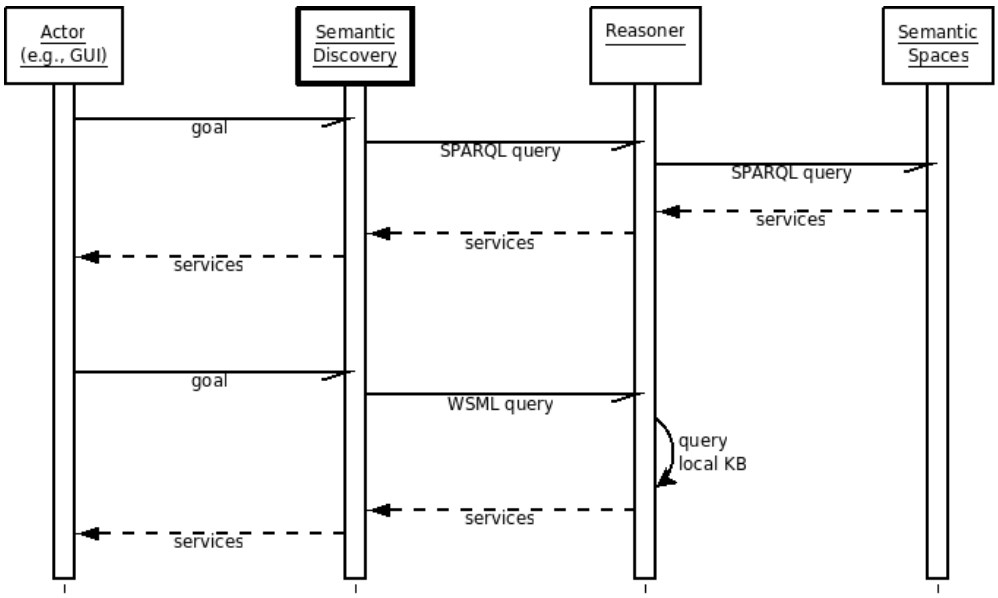

Figure 10: Interactions between involved components to discover services.

of WP6 and is different to the “WP5 goal” used in the present document. In terms of the LPML, a goal is a process element that is not bound to a service at design time.

Interactions. The semantic discovery component relies on reasoning services provided by WP3 for querying the ontologies stored in the semantic spaces, which are provided by WP1. Therefore, it is required that there is a running instance of the reasoning service available. The reasoner has to be connected to the semantic spaces that should be used as the knowledge base. The knowledge base contains the classification hierarchy as well as WSMO-Lite descriptions of the Web services as WSML ontologies.

Figure 10 depicts the interactions between the components involved in the discovery process given a goal. This goal can be specified by a human using the GUI or can be automatically generated by the composer or consumption platforms. In case of processing the first goal in the upper part of Figure 10, a user invokes the semantic service discovery with a specified goal that expresses the desired functionality. In compliance to the definition of the desired functionality on page 39, the goal may include the specification of desired inputs, outputs, pre-condition, effect, and an assignment to classes of functionality. The semantic discovery component creates a WSML query from the goal. In order to let the semantic service discovery component comprehend the goal, it requires to know those ontologies used to model the WSML axioms, which express pre-conditions and effects. The used ontologies are specified by the actor who created the goal.

Furthermore, the semantic service discovery component resolves the selected classes of functionality within the goal. As described in Section 5.3, the overall pre-conditions and effects of the goals have to hold for services which are returned as results. And the specification of these functionality classes causes that expensive satisfiability checks are only performed on those services that are member of the selected classes or its sub classes. The semantic discovery component sends the generated query to the reasoner instance. In case of a SPARQL query, the reasoner forwards the SPARQL query to the semantic spaces. The semantic spaces process SPARQL queries directly and return the result of the query to the reasoner that forwards the results to the discovery module.

In the case that the reasoner instance receives a WSML query (see the lower part of Figure 10), the query can be directly processed by the reasoner. The reasoner incorporates its own local cache of the knowledge base from the semantic spaces and returns the results to the discovery module. The user interface of the discovery module finally displays the results to the user.

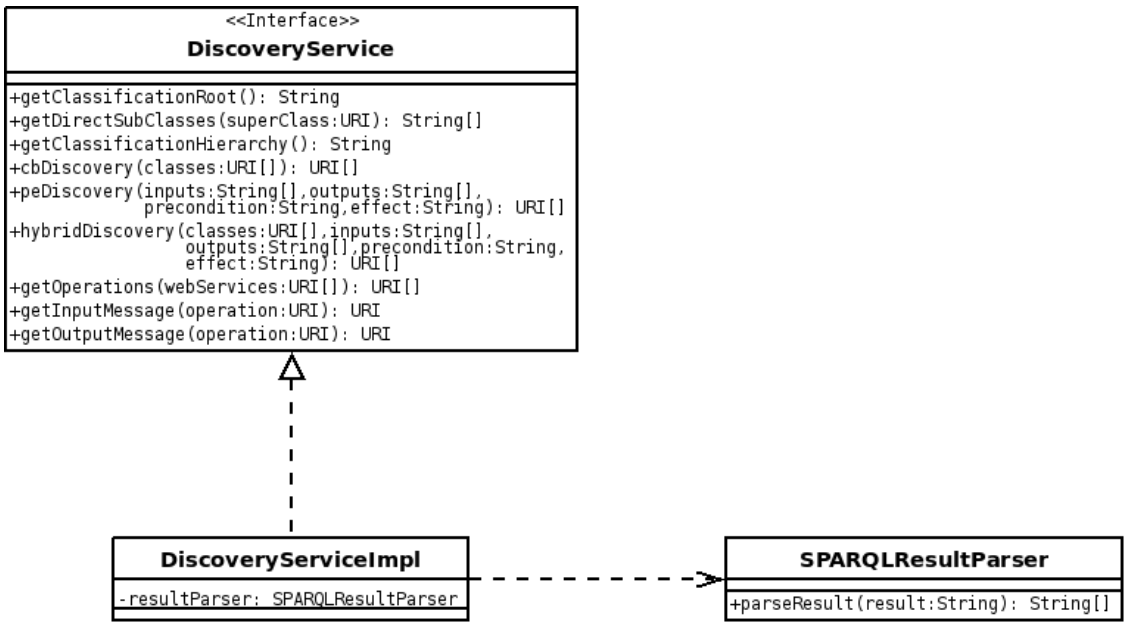

Figure 11: Discovery Service Class Structure.

Structure and Functionality. The semantic discovery is developed as Web service such that it can be used by other SOA4All components with standard Web protocols. The semantic discovery component offers a range of methods for various purposes related to the discovery of Web services. Figure 11 depicts the implementation structure of the discovery service and the following Table 3 explains the signature and functionalities of some the methods the discovery service provides.

Method Name	Input	Output	Functionality
`getClassificationRoot`	`None`	`String`	Returns the URI of the FunctionalClassificationRoot concept
`getDirectSubClasses`	`URI superClass`	`String[] subClasses`	Retrieves direct subclasses of the superClass by sending appropriate SPARQL query to semantic spaces via the ontology reasoner.
`getClassificationHierarchy`	`none`	`String hierarchy`	Retrieves the complete classification hierarchy by sending appropriate SPARQL queries to the reasoned and returns the hierarchy as XML with child nodes of a parent node denoting the subclasses of the class represented by the parent node.
`cbDiscovery`	`URI[] classes`	`URI[] webServices`	Performs classification based discovery. As explained in Section 5.1, the method creates a

			WSML query from the set of class names provided as input and returns the URIs of all Web services that are classified in all the input classes
peDiscovery	String[] inputs, String[] outputs, String pre-condition, String effect	URI[] webServices	Performs pre-conditions and effects based discovery, as explained in Section 5.2. The method checks for each Web service description in semantic spaces whether it has all desired input, all desired outputs and whether its pre-condition implies the desired pre-condition and its effect the desired effect. The method returns the URIs of all Web services that satisfy all of these checks.
hybridDiscovery	URI[] classes, String[] inputs, String[] outputs, String pre-condition, String effect	URI[] webServices	Performs discovery as explained in Section 5.3. The method checks for each Web service description that is match of cbDiscovery with classes as input, whether it has all desired input, all desired outputs and whether its pre-condition implies the desired pre-condition and its effect the desired effect. The method returns the URIs of all Web services that satisfy all of these checks.
getOperations	URI webService	URI[] operations	Returns the set of all operations defined for the Web service. The operations are retrieved by sending an appropriate WSML query to the reasoner.
getInputMessage	URI operation	URI inputMessage	Returns the URI of the input message of the operation by sending the appropriate WSML query to the reasoner.
getOutputMessage	URI operation	URI outputMessage	Returns the URI of the output message of the operation by sending the appropriate WSML query to the reasoner.

Table 3: Methods of Semantic Discovery.

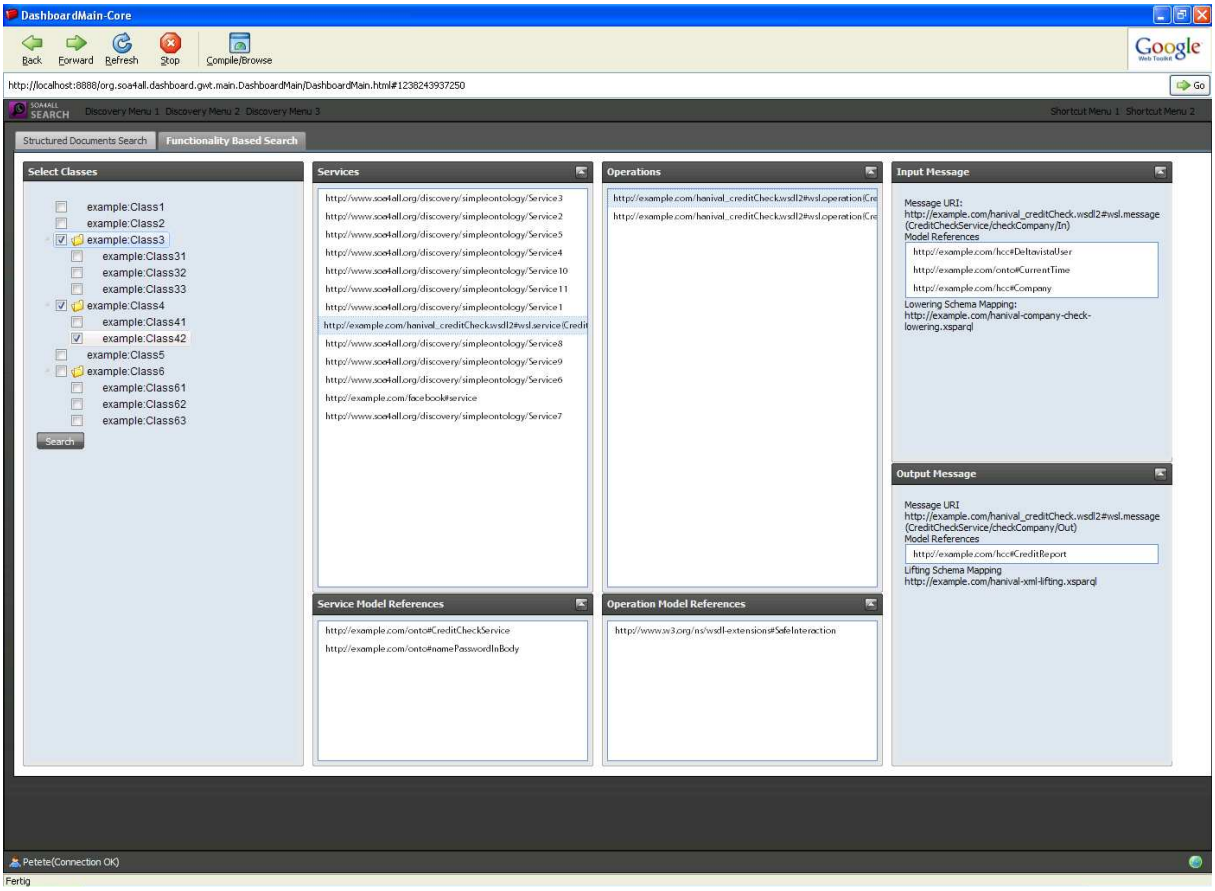

Figure 12: Graphical User Interface for Semantic Discovery.

User Interface. Figure 12 shows the graphical user interface for modeling a goal and searching Web services semantically. The semantic discovery GUI is fully integrated in the SOA4All Studio. The left part of the window serves for modeling the goal, whereas the right part serves for browsing the results, which are the discovered Web service descriptions.

In the left of the goal modeling part, a user can browse through the classification hierarchy. A class is selected by ticking besides it, and this selection appends the class to the list of desired classes of the goal. The classification hierarchy is retrieved by invoking semantic discovery Web services getClassificationHierarchy. Once all the classes that the user wishes a service to be a member of are selected, the user can define the desired pre-conditions and effects if any in a separate window.

The Web service discovery request can be sent by pressing the "Search" button. Upon pressing the button, the goal is sent to the semantic discovery component by invoking the Web service peDiscovery. The outputs of semantic discovery, which are the Web service descriptions that match the goal, are shown in the second column of the window. When a user selects a matched Web service, the operations of the selected Web service are retrieved by invoking the Web service getOperations and presented in the third column. Upon selecting an operation, a user can see details about the input message and the output message in the fourth column. These details are retrieved by invoking the Web services getInputMessage and getOutputMessage, respectively.

6. Related Work

The automatic discovery of services was in the last years and continues to be a very popular research topic. Many solutions have been proposed ranging from pure syntactic to highly logic based approaches. However, they are either syntax based or less expressive or not scalable. The combination of all three aspects is extremely important nowadays given the high number of services available on the Web. In the following, we will discuss some well-known Web service discovery approaches in more detail.

UDDI [21] basically supports keyword based search, which though being fast and scalable has certain limitations. Consider a Web service that sells books and the corresponding WSDL document that contains an output message of type “Book”. The type “Book” is defined in the “types” block of the WSDL document. If a requester searching for a book selling web service, uses keywords, say “Buch”, “hardcover”, or “volume” instead of “Book” the book selling Web service will not be found as a match by the UDDI search algorithm. In other words, the keyword based search detects a Web service as match only if the requester uses the same terminology as the provider. Consider another Web service that sells books but not text books. If a requester searches for a web service that sells text books, keyword based search will detect this Web service as a match. So, keyword based search suffers from two main problems, namely lack of interoperability and lack of semantics.

To overcome such limitation, UDDI discovery was enhanced by semantics in **Error! Reference source not found.**. This approach models services in a hierarchy that is only modeled by the usage of inputs and outputs. In comparison our first classification-based approach presented in Section 5.1, the hierarchy is not related to a description of the actual functionality of services. A sole description of inputs and outputs is not sufficient to describe the functionality, for instance by a descriptive class name as our approach based on WSMO-Lite service descriptions did.

OWL-S Matchmaker [23], [24] uses OWL-S Profile for describing Web services as well as describing the goal. Even if OWL-S Profile is designed for modeling pre- and post conditions in addition to the types of input and output parameters of the Web services, no concrete formalism was fixed for describing the conditions. As a result, the goal specification reduces to the types of input and output parameters. So, it neither allows the specification of relationships between inputs and outputs nor constraints on state changes caused by the Web service execution. [25] proposed a SWRL based solution to fill the gap. The approach presented in [25] however does not make use the power of logic reasoning but looks for a bipartite graph that matches each term in the Web service condition with a distinct term in the query condition. In our work, we have not only shown the semantics of pre-conditions and effects based matching, but also proposed WSML axioms[6] for modeling pre-conditions and effects.

Approaches like [26] model Web service as well queries as description logic classes and base the matchmaking on the intersection of service advertisement and query that computed by a DL reasoner. However, those approaches fail to reason about the dynamics of Web services, since DL reasoners cannot reason about changing knowledge bases [28]. Approaches like [27] can deal with variables, but are limited to Web services that do not change the world and thus can be described by a query. Thus, our approach is more general than the approach presented in [27], as ours works for state changing as well query like services. [29] proposes heuristics to tackle the complexity of matchmaking based on functionality, since the exact matching of pre-conditions and effects is NP-Complete. Such an approach necessarily compromises with the soundness and completeness of the matches,

[6] http://www.wsmo.org/wsml/wsml-syntax

whereas our approach is sound and complete. [30] presents a technique to enhance semantics in UDDI, such that existing UDDI repositories can be used for performing semantic discovery as well. However, [30] does not present new results regarding semantic discovery.

WSMO Discovery presented in [20] is based on modeling both user goals as well as Web services as complex DL concepts. It differentiates between exact-match, subsumption-match, plugin-match and intersection-match depending on the overlapping of the sets described by a goal and a Web service. Furthermore, this approach is not efficient enough, because the algorithm has exponential time consumption and is not optimized at all. Apart from being less likely to scale, it does also not consider the pre-conditions and effects of Web services, although they can be expressed by the language.

Stollberg's thesis on a scalable and goal-based service discovery [31] is related to our third approach as it also relies on a pre-computed indexing structure (SDC trees) and also describes the functionality of services by pre-conditions and effects. However, this approach assumes that pre-conditions imply effects. This assumption does not hold for describing a Web service functionality like one that allows users to unsubscribe from a mailing list, where a pre-condition is that there exists a costumer number and an effect that there exists no costumer number for that particular costumer anymore. Such an implication cannot be true and, thus, this approach fails modeling such services.

7. Conclusion

In this deliverable, we have studied the problem of service discovery. We have motivated the need for a new service discovery approach by analyzing the drawbacks of the existing approaches. Based on the problem description and requirements analysis, we introduced as solution two main approaches for service discovery.

The Service Discovery component enables users to find services that are appropriate for their needs. For this purpose, it provides two types of discovery approaches, *full text-based discovery* and *semantic discovery*. Full text-based discovery uses Web service descriptions (WSDLs) and related documents found by the crawler, allowing users to search for services by entering keywords (much like typical Web search engines).

The semantic discovery component allows users to enter a more structured goal (service classification, pre-conditions and effects). It finds the service that matches the goal using the reasoning facilities provided by WP3. Within the semantic discovery approach, we presented three mechanisms, which are founded on the functionality of the services, classification, or both. We have presented the way they function at the conceptual as well as at the implementation level. The implementation instructions are will be delivered accompanied to the first service discovery prototype source code.

The results presented in this deliverable is be mainly used by service provisioning and service consumption platforms developed in WP2, as well automatic service composition techniques developed in WP6. With ongoing integration of the SOA4All components, the evaluation of the service discovery becomes possible in the future (after M18). The evaluation will provide us the input to develop an adequate and efficient indexing.

8. References

[1] Ioan Toma, Nathalie Steinmetz, Jean Pierre Lorre, Dumitru Roman, Jacek Kopecky, Holger Lausen, John Domingue and Olivier Fabre. *D5.1.1 - State of the Art Report on service description and existing discovery techniques*, SOA4All deliverable, 2008

[2] Shay Banon & Allan Hardy. Compass – Java Search Engine Framework [online]. Available at: http://www.compass-project.org/docs/2.1.1/reference/html/

[3] Apache. Lucene Java Documentation [online]. Available at : http://lucene.apache.org/java/2_4_0/

[4] JBoss. Hibernate Annotations [online]. Available at: http://www.hibernate.org/hib_docs/annotations/reference/en/html/

[5] DWYER Jeff. Pro Web 2.0 Application Development with GWT. New York: Apress, 2008, 450 p. ISBN 978-1-59059-985-3

[6] Manuel Brunner, Nathalie Steinmetz, Olivier Fabre, Marin Dimitrov, Iván Martinez. *D5.1.2 – First Crawler Prototype,* SOA4All deliverable, 2009

[7] eBM Websourcing. Dragon – Open Source SOA Governance Platform [online]. Available at: http://dragon.ow2.org/

[8] JBoss. Hibernate [online]. Available at: http://www.hibernate.org/

[9] Sun. Java Data Object (JDO) [online]. Available at: http://java.sun.com/jdo/index.jsp

[10] Apache. Apache ObjectRelationalBridge (OJB) [online]. Available at: http://db.apache.org/ojb/

[11] Wikipedia. Index (publishing) – Wikipedia, the free encyclopedia [online]. Available at: http://en.wikipedia.org/wiki/Index_(publishing)

[12] Wikipedia. Stemming – Wikipedia, the free encyclopedia [online]. Available at: http://en.wikipedia.org/wiki/Stemming

[13] Wikipedia. Doug Cutting – Wikipedia, the free encyclopedia [online]. Available at: http://en.wikipedia.org/wiki/Doug_Cutting

[14] CBDI. Service Oriented Architecture Practice – CBDI the SOA Portal [online]. Available at: http://www.cbdiforum.com/

[15] UDDI: UDDI Executive White Paper. Technical report, UDDI.org (November 2001)

[16] http://www.seekda.com

[17] R. Chinnici, M. Gudgin, J. Moreau, and S. Weerawarana, Web Services Description Language (WSDL) Version 1.2 Part 1: Core Language, 2003.

[18] R. Chinnici, J. Moreau, A. Ryman, and S. Weerawarana, Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language, 2007.

[19] Tomas Vitvar, and Jacek Kopecky and Dieter Fensel. *WSMO-Lite: Lightweight Semantic Descriptions for Services on the Web*. CMS WG Working Draft D11v0.3. June 2009. Available at http://cms-wg.sti2.org/TR/d11/

[20] Uwe Keller, and Ruben Lara, and Axel Pollers, and Ioan Toma, and Michel Kifer, and Dieter Fensel. *WSMO Web Service Discovery.* November 2004. Available at http://www.wsmo.org/TR/d5/d5.1/v0.

[21] UDDI. *UDDI Executive White Paper*. November 2001. Available at http://uddi.org/pubs/UDDI_Executive_White_Paper.pdf.

[22] N. Srinivasan, M. Paolucci, K. Sycara, *An Efficient Algorithm for OWL-S based Semantic Search in UDDI*, Proceedings of 1st International Workshop on Semantic Web Services and Web Process Composition, San Diego, CA, USA, July 6, 2004.

[23] Paolucci, M. and Kawamura, T. and Payne, T.R. and Sycara, K. *Semantic Matching of Web Services Capabilities*. In Proceeding of The First International Semantic Web Conference (ISWC2002). 2002. Sardinia, Italy.

[24] Katia Sycara and Massimo Paolucci and Anupriya Ankolekar and Naveen Srinivasan. *Automated Discovery, Interaction and Composition of Semantic Web Services*. Journal of Web Semantics (1). December 2003.

[25] Bener, Ayse B. and Ozadali, Volkan and Ilhan, Erdem Savas. *Semantic matchmaker with precondition and effect matching using SWRL*. Expert Syst. Appl. **36**(5). 2009. Pergamon Press, Inc. Tarrytown, NY, USA.
[26] Li, Lei and Horrocks, Ian. *A Software Framework for Matchmaking Based on Semantic Web Technology*. Int. J. Electron. Commerce **8**(4). 2004. M. E. Sharpe, Inc. Armonk, NY, USA.
[27] Duncan Hull and Evgeny Zolin and Andrey Bovykin and Ian Horrocks and Ulrike Sattler and Robert Stevens. *Deciding Semantic Matching of Stateless Services*. In Proceedings of The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference. July 2006. AAAI Press. Boston, Massachusetts, USA.
[28] B. Motik, S. Grimm, and C. Preist. *Variance in e-business service discovery*. In Proc. 1st Intl. Workshop SWS2004 at ISWC 2004, 2004.
[29] Bellur, U. and Vadodaria, H. *On Extending Semantic Matchmaking to Include Preconditions and Effects*. Web Services, 2008. ICWS '08. IEEE International Conference on. September 2008.
[30] Rama Akkiraju and Richard Goodwin and Prashant Doshi and Sascha Roeder. *A Method for Semantically Enhancing the Service Discovery Capabilities of UDDI*. In Proceedings of the IJCAI-03 Workshop on Information Integration on the Web (IIWeb-03). 2003.
[31] M. Stollberg. PhD thesis. *Scalable Semantic Web Service Discovery for Goal-driven Service-Oriented Architectures*, 2008.
[32] MySQL :: The world most popular open database in the world[online]. Available at: http://www.mysql.com/
[33] Wikipedia [online]. Available at: http://www.wikipedia.org/
[34] All News, Video and Images [online]. Available at: http://digg.com/
[35] Apache Tika – Apache Tika [online]. Available at: http://lucene.apache.org/tika/
[36] Hibernate.org – Hibernate Search [online]. Available at: https://www.hibernate.org/410.html
[37] Sphinx – Free open-source SQL full-text search engine [online]. Available at: http://www.sphinxsearch.com/
[38] The Java Persistence API - A Simpler Programming Model for Entity Persistence [online]. Available at: http://java.sun.com/developer/technicalArticles/J2EE/jpa/
[39] North American Industry Classification System [online]. Available at: http://www.census.gov/eos/www/naics/
[40] United Nations Standard Products and Services Code [online]. Available at: http://www.unspsc.org/
[41] Levenshtein Distance [online]. Available at: http://en.wikipedia.org/wiki/Levenshtein_distance
[42] Heritrix [online]. Available at: http://crawler.archive.org/

Appendix A: Research Paper

Accompanied to this deliverable is a research paper “Towards Scalable Discovery of Semantic Web Service Functionality” by Sudhir Agarwal (UKARL) and Martin Junghans (UKARL). We submitted this paper to the 4th Annual Asian Semantic Web Conference (ASWC) 2009. The conference will be held in Shanghai from the 6th to 9th of December 2009. Until the notification of acceptance (10th September 2009) and also in case of acceptance, we do not have the permission to republish the content of the paper with this deliverable. In case of rejection, the paper may be published with this deliverable after notification.