Project Number:	**215219**
Project Acronym:	**SOA4ALL**
Project Title:	**Service Oriented Architectures for All**
Instrument:	**Integrated Project**
Thematic Priority:	**Information and Communication Technologies**

D5.1.3 Second Crawler Prototype

Activity:	Activity 2 - Core Research and Development	
Work Package:	WP5 – Service Location	
Due Date:	M18	
Submission Date:	31/08/2008	
Start Date of Project:	01/03/2008	
Duration of Project:	36 Months	
Organisation Responsible of Deliverable:	SEEKDA	
Revision:	1.0	
Author(s):	Nathalie Steinmetz	UIBK
	Holger Lausen	SEEKDA
	Manuel Brunner	SEEKDA
	Iván Martinez	ISOCO
	Alex Simov	ONTOTEXT
Reviewer(s):	Maria Maleshkova	OU
	Bernhard Schreder	HANIVAL

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)

	Dissemination Level	
PU	Public	**X**

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	02.07.2009	Initial Version	Nathalie Steinmetz (UIBK)
0.2	30.07.2009	First Draft Section 4	Alex Simov (ONTO)
0.3	31.07.2009	First Draft Sections 2.3 and 3.1.3	Iván Martinez (ISOCO)
0.4	06.08.2009	Second Draft Sections 2.3 and 3.1.3	Iván Martinez (ISOCO)
0.5	12.08.2009	Third Draft Sections 2.3 and 3.1.3	Iván Martinez (ISOCO)
0.6	24.08.2009	Editing draft Sections 1, 2 and 3	Nathalie Steinmetz (UIBK)
0.7	25.08.2009	Finalizing Document	Nathalie Steinmetz (UIBK)
0.8	02.09.2009	Integrating reviewers' comments	Nathalie Steinmetz (UIBK)
0.9	04.09.2009	Integrating reviewers' comments	Iván Martinez (ISOCO)
1.0	04.09.2009	Finalizing	Nathalie Steinmetz (UIBK)
1.0	14.09.2009	Final Editing	Malena Donato (ATOS)

Table of Contents

List of Figures

List of Listings

Glossary of Acronyms

Acronym	Definition
API	Application Programming Interface
D	Deliverable
EC	European Commission
hRESTS	HTML for RESTful Services
OWL	Web Ontology Language
REST	Representational State Transfer
SA-REST	Semantic Annotations for RESTful services
SAWSDL	Semantic Annotations for WSDL
SWS	Semantic Web Service
UDDI	Universal Description, Discovery and Integration
WP	Work Package
WSDL	Web Services Description Language
WSML	Web Service Modeling Language
WSMO	Web Service Modeling Ontology

Executive summary

Within this deliverable the second SOA4All crawling prototype is described. First, we present a follow-up of the WSDL crawling described in the first crawling prototype and provide an overview of the Web API and Semantic Web Services crawling. Then we explain the structure and content of the different formats that we crawl and describe what semantic meta-data we extract from them. We also provide an updated overview of the ontologies that we use to structure and store the metadata and explain how we produce unique unified service representations for both WSDL services and Web APIs. Finally we provide a new version of the crawler API that has been developed in the scope of the first crawling prototype to allow an easy, comprehensive and consistent way of programmatic access to the crawled data.

1. Introduction

The main concept of SOA4All is the support for billions of services, be it in the domain of service annotation, discovery, composition, etc. If we want to be able to discover services on such a large scale, we cannot rely on the concept of manual service registration, as it is promoted by UDDI or by most of the current Web Service portals (e.g. ProgrammableWeb, StrikeIron, etc.). Our approach is to crawl the Web for services and this way automatically gather as many as possible publicly available services.

While in the first crawler prototype we have focused our attention on WSDL services, we have enlarged our focus for the second crawler prototype with RESTful services (a.k.a. Web APIs) and Semantic Web Services (SWS). This is an important step considering the fact that on the one side RESTful services currently show the greatest growth rate and, on the other side, more and more SWS, i.e. semantically annotated services, are published on the Web. We do not only want to provide users means to annotate services within the SOA4All Studio, but also want to provide them access to the already available semantic services.

Figure 1 shows how the Service Crawler is embedded into the whole SOA4All architecture, with focus on the single discovery components. The crawler is the component that gathers the services and related meta-data from the Web and thus provides service data to the other discovery components, e.g., Semantic Discovery and Ranking. None of the other components – be it WP5 components or others – does directly access the crawler component: the Crawl API provides access to the crawl data via a RESTful service. The data is then stored either in the Documents Repository (WSDLs, related documents, Web APIs, SWS) or in the Semantic Spaces (RDF service meta-data).

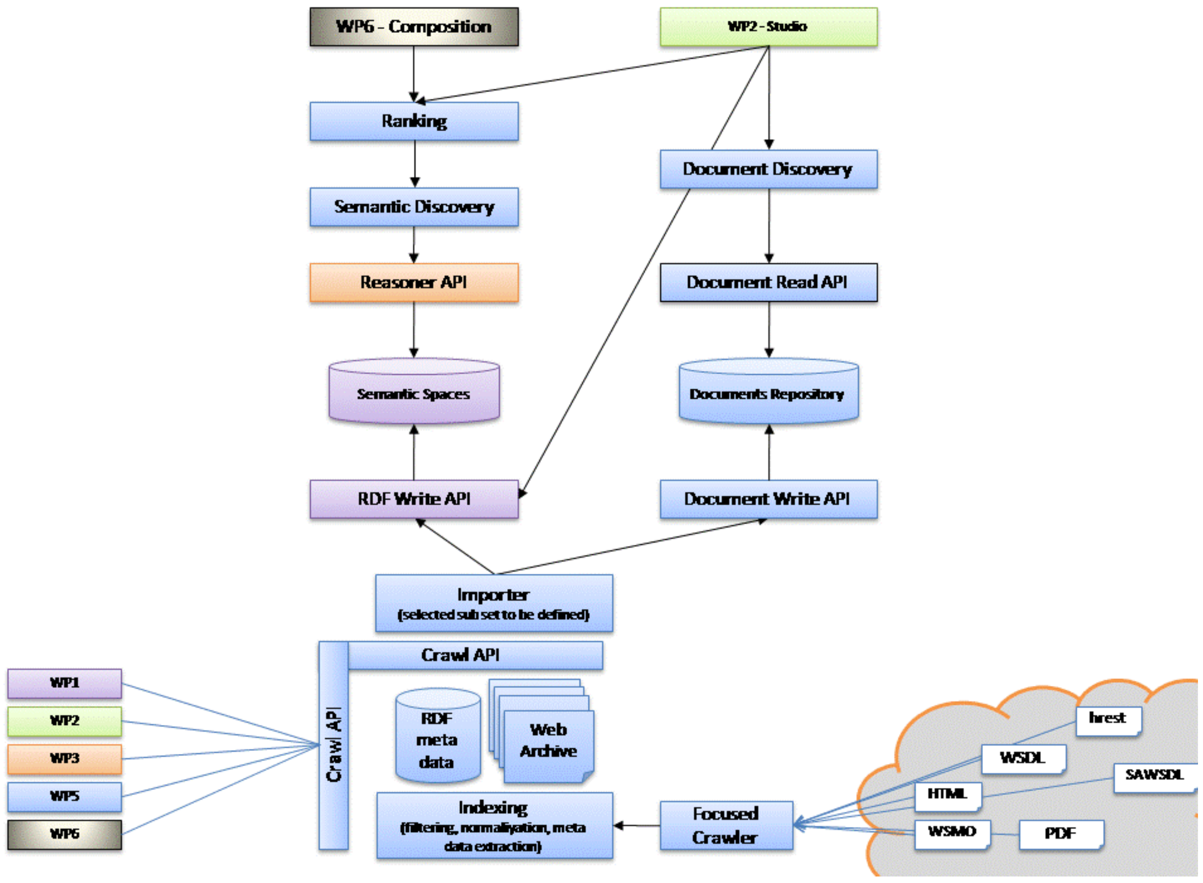

Figure 1: WP5 Architecture with Focused Crawler

2. Web Service Crawler

The SOA4All Web Service Crawler performs a focused crawl of the Web, concentrating on those parts of the Web that contain service relevant parts. This is a different approach to traditional service discovery approaches: currently people either search by using standard search engines like Google with keyword search, or they rely on the manual registration of services on specialized portals (like ProgrammableWeb[1] or XMethods[2]). Both of these methods come together with problems like outdated or missing data or turn out to have efficiency deficiencies [14][15]. The need for the crawling approach, and the success of the first results becomes obvious by looking at the quantitative analysis of the two described search methods provided in [16]. While the work with the WSDL crawling got already a bit mature over the last year, the approach with crawling for Web APIs is very new and experimental. So far Web APIs are mostly only published on the Web by providing their Web API Homepage, but there is no possibility to do a large scale search on the Web. This is mainly because they don't follow any standards and are mostly simple HTML pages that don't even allow a user to do a filtered search on a search engine like Google.

A very important aspect when developing a crawling strategy for Web Services is the focus, as we do not want to crawl blindly through the World Wide. We thus need to focus our crawl only on WSDL descriptions, pages that are related to the latter, Web API homepages or semantic service descriptions. A good focus and service identification methodology allows us to gather as many as possible publicly available services from the Web and to not rely on a manual registration of services.

While – in the scope of the first crawler prototype – the Service Crawler concentrated its search on WSDL services and related information, as well as on Web APIs, we extended its scope to SWS for the second crawler prototype and put more focus on the Web API crawling than in the first prototype. Considering the fact that the number of available Web APIs is constantly growing, we estimate that the support for these services will significantly increase the number of services made available through the SOA4All infrastructure.

In D5.1.2 First Crawler Prototype [1] we have provided an overview of the Service Crawler and its architecture. We have introduced new processors that we added to the crawler in order to focus our crawls on Web Services and in order to write some meta-data to RDF. We have also described how we can configure the crawler to do a focused crawl of WSDL Web Services and related documents and have described the crawl output. In the following, we will provide an overview of the WSDL and the Web API crawling approaches and furthermore describe how we crawl the Web for Semantic Web Services (SWS). As the work on WSDL crawling has been started in the European project Service-Finder[3], we only describe that part shortly (the main SOA4All extensions in terms of WSDL crawling have happened on the meta-data extraction and storage and will be described in Section 3). In this section, we concentrate mainly on the crawling novelties within SOA4All, the Web API and SWS crawling. [2] provides more details on the WSDL crawling strategies developed within the Service-Finder project and their implementation.

2.1 Focused Web Service Crawling Techniques

Two very important means to focus a crawl are queue and URL scheduling (for an introduction on crawl scheduling refer to [1]), as without proper scheduling it is not possible to focus a crawl. Prioritizing URLs is absolutely necessary, as due to (a) the size of the Web,

[1] http://www.programmableweb.com/

[2] http://www.xmethods.net/

[3] http://www.service-finder.eu/

(b) restricted hardware resources and (c) time constraints, it is unrealistic to provide a complete coverage of the Web. We want to only fetch those documents from the Web that are relevant for the domain of Web Services, that is we want to scope our crawl to the relevant resources.

The crawler creates new queues per top-level domain, i.e. each new host gets its own queue. In this queue all the URLs that belong to that host are scheduled. Influencing the URL and queue scheduling means (in the case of our Service Crawler) allocating costs to URLs or setting precedences of URLs and/or crawl queues: URLs with a low cost (or a high precedence) are put more upfront in the queue, such that they are processed earlier than URLs with a higher cost (or lower precedence) that are put more at the back of a queue. The concept of scheduling for queues is similar to the one for URLs: queues with a high precedence are processed earlier than queues with a lower precedence.

To assign costs for a Web Service focused crawl we (a) check the single URLs for certain criteria and (b) look at the content of its provenance page. We, e.g., prioritize URLs that promise to be WSDL service descriptions (URLs containing "?wsdl") or that seem to be related to services (URLs containing, e.g., "ws", "api", "service", etc.). Furthermore we calculate a score for the provenance page, i.e. for the 'from-link' URL whose outlinks we are currently looking at. The score provides us a probability related to whether the page content is related to Web Services or not (based on the number and position of Web Service related term occurrences in the page's content)

2.2 WSDL Crawling – Follow Up

The first important aspect in Web Service crawling is, as mentioned above, the right focus of the crawler. To focus a crawl on WSDL services and on related information we not only need to apply intelligent URL and queue scheduling methods, but we also need to choose the right set of seed URLs. We collect the seed URLs in a semi-automatic process, involving, e.g., screening of well-known sites that contain information or registries of Web Services. We then start a focused crawl, beginning with the seed URLs and domains that we know that host Web Services or that talk about Web Services, and further extending these during the crawl.

2.2.1 WSDL Identification Techniques

The huge advantage of WSDL crawling, as opposed to, e.g., Web API crawling, is that WSDL descriptions are conformant to a standard – a fact that makes it easier for us to identify the service descriptions. First of all we concentrate our search on textual files, as both the WSDLs and the related information are most often stored in text files. That said, we by default reject a lot of content in our crawls, like images, audio or video files. We specifically look at pages like HTML, XML, PDF, other text documents, i.e. all types of files that could either contain a service description or related information. During the crawl process we check whether fetched pages are valid WSDL descriptions: we parse the content of a page with a SAX XML parser and, if the content is XML, check whether it is a valid WSDL 1.1[4] or WSDL 2.0[5] description. We therefore extract the XML start element; if the name of the element is *definitions* we are looking at a WSDL 1.1 description - if the name is *description* we are looking at a WSDL 2.0 description. Next we check for elements like *service*, *portType*, *operation* and *endpoint*.

2.2.2 Related Information Identification

Identifying documents on the Web that are related to WSDL services is harder than detecting the WSDLs themselves. They do not all conform to a standard description, but consist mainly

[4] http://www.w3.org/TR/wsdl

[5] http://www.w3.org/TR/2003/WD-wsdl20-20031110

of normal HTML, PDF or text files. Content-wise the related information may consist of provider documentation of the service functionality, provider Web pages, Wikis, Blogs, FAQs, user ratings and many more. This sort of information allows us to know more about an offered service than only its technical description: this might be important for a potential user of a service who wants to estimate how reliable a service is, whether it is free or not, etc.

To identify the related information, we need to know how the corresponding Web resources are tied to the service descriptions: the WSDL may be linking to the document, the document may be pointing to the service or the service provider's service definition or the document may also not at all be directly linked to the service. In a first step, we consider the inlinks and outlinks of the WSDL documents, i.e. those resources that include links pointing to the service description and vice versa. This information can be gathered from the link graphs that are being written by the crawler during crawl run-time.

Unfortunately it is not sufficient to collect related information only by relating outlinks and inlinks to the service descriptions. While on the one hand, not all outlinks or inlinks lead to useful information related to the services, we, on the other hand, miss information that is relevant but that stays hidden to us when we only concentrate on outlinks and inlinks. Another way that we explored to detect information related to services is looking at term vector similarities. We assume that by looking at the term vectors of pages we are able to assess the similarity between documents and services and can thus conclude that they are related. To do so, we calculate at crawl run-time the term vectors of all fetched pages and store them. Afterwards, in the crawl post-processing process we perform the actual term vector similarity comparison. Clearly we cannot apply this approach blindly on all fetched documents, as this would require far too much computing power. We restrict our approach to checking the similarity of the term vectors of services to the term vectors of documents fetched from their respective provider domains.

2.2.3 Integration Into Crawler

As already outlined in [1], we have extended the crawler with new processors that handle the WSDL identification, the writing of link graphs, the URL cost assignment, the Queue Precedence Policy (Queue Scheduling), as well as the writing of RDF meta-data regarding the detected WSDL services. We will not further detail the implementation of these processors here.

The Related Documents Identification is happening in a post-processing step after a crawl has been terminated. This process is implemented as a small Java program that goes (a) through the link graph and (b) queries the RDF service meta-data to get the unique services and their WSDLs (stored in RDF during the crawl). The information about related documents is then as well stored as RDF triples and allows each further post-processing analysis to only rely on RDF queries (e.g., SPARQL).

2.3 Web API Crawling

Same as for WSDL services and their related documents, we do not want to crawl unnecessary data when we are looking for Web APIs on the Web. But similar to the problem of identifying related documents it is unlike harder to detect Web APIs than WSDL descriptions. They again do not conform to any standardised description. Web APIs are in the end HTML documents, same as normal Web pages, differentiated only by the fact that they expose a functionality that can be invoked by (in most cases) adding a specific query string to the URL that then calls a specific method in the background (e.g. https://api.linode.com/api/?api_key= cakeisgood&action=domainGet&DomainId=45F33). We call Web APIs documents that provide a set of functions that are accessible over the Internet using basic HTTP request methods (GET, POST, PUT, DELETE), including SOAP (thus in general also including WSDL services). Such a Web API has to be well-described and needs to be intended for automatic, programmatic usage. And while such an API may represent a RESTful service, it can as well represent a service that is not strictly RESTful (following the definition in [3]).

Usually Web APIs are a lot easier to create than WSDL services and quite understandable for humans (which explains their growth rate as mentioned above). Often they are published on specific portals on the Web (as, e.g., ProgrammableWeb[6]), which is thus where we start collecting seed URLs for the Web API focused crawl.

2.3.1 Web API Identification Techniques

To tackle the challenge of identifying the Web APIs we have developed two initial approaches, which are described here. While one of the approaches addresses the problem from an automatic classification side, using supervised machine learning, the other one follows a more 'manual' term vector similarity approach, relying on hand-crafted term vectors. Both approaches are still in a rather experimental phase and not yet as well matured and evaluated as the WSDL crawling approach. We will though provide an initial evaluation of the two in Section **Error! Reference source not found.**.

Automatic Classification Approach

The first approach follows a traditional data mining approach: text classification. Automated classification (also called categorisation) of texts has become quite important as in recent years huge amounts of digital documents have become available [4]. Text classification is applied in many different contexts like, e.g., automated meta-data generation, document filtering (e.g., spam filtering). We can distinguish between two major types of text classification: supervised and unsupervised learning, as described in [5]. In short, supervised learning works such that the user first provides an example set (a.k.a. positive set), i.e., a set of already classified documents. The learning function then takes this set as input and produces a class label prediction (the so-called classification) from this. Afterwards other documents are 'labeled' (classified) based on the results from the example set (which makes it very important that the example set really contains relevant documents). Unsupervised learning functions are used when there is no training set available for the machine-learning tool.

In our approach we chose to use a supervised learning algorithm, concretely the Support Vector Machine (SVM) model [5]. We used Web API documents as example set that we collected from a site that is known to provide Web APIs: ProgrammableWeb. This is our so-called *positive set*. Our *negative set* consists of randomly crawled Web sites, i.e. with no specific Web Services focus. When a Web site is matched as positive it gets a score that indicates that this page is marked positively by the classifier. In a final step we then need human interaction to approve the results of the automatic classification.

Term Frequency Approach

Our second approach is based on term vector similarities and on term frequencies and tries to tackle the weaker aspect of the automatic SVM approach: the fact that it is only based on pure text and does not take into account any structural elements of the text. In the Web domain we predominantly deal with (semi-)structured documents in HTML, i.e. not pure text documents but text intermixed with mark-up. Here we would like to handle words differently depending on their mark-up or depending on where they occur in the document. This way we could, e.g., favor keywords that appear in the title of a page or that are marked in bold or italic. We might as well want to take into account the URL of a Web document, which often contains words describing the topic of the page. Another relevant aspect covers the syntactical properties of the language used in Web API homepages. Most times they contain a higher amount of camel-cased words than random pages (e.g., getDocument) and often they contain fewer external links than usual (links that point to pages on other domains).

[6] http://www.programmableweb.com/

Often Web API homepages also contain internal links that target to the same domain, e.g., example calls for the described API.

To address the shortcomings of the automatic classification approach we have developed the term frequency approach that provides an analysis of HTML pages taking into account the following aspects:

- hand-crafted term vectors (build of terms from the Web API domain) are compared to the HTML pages' term vectors, differing between text in headings, titles or normal text extracts and counting as well the occurence of terms (i.e. their frequency)
- the number of occurences of camel-case tokens, external and internal links within the HTML pages is counted
- the pages' URLs are checked for certain keywords (e.g., api)

We have developed three different indicators that are supposed to tell us what type of document a page is: Web-related, API or Documentation. The indicators are based on a manual feature analysis of actual Web APIs (chosen from ProgrammableWeb). In a next step these indicators are used to calculate the score of a page. The score gives us a probability value of whether the page is a Web API or not. The following listings provide an overview on how the single indicators are configured (all terms having been de-capitalized before the term comparison).

Strong indicator: keyword(s) in URL: rest

Medium/weak indicator: keyword(s) in content: rest, web (service(s)) api, url

Strong indicator: high amount of internal links (i.e. links pointing to the same domain)

Listing 1: Web API "Web-Related" Indicator

Medium indicator: keyword(s) in URL: api(s), dev(eloper)(s), lib, code, service, sdk

Strong indicator: above keyword(s) appearing as separate tokens in the URL

Medium Indicator: keyword(s) in content: api(s), dev(eloper)(s), lib, code, service, sdk

Medium indicator: above keyword(s) appearing in heading- or title-tags

Weak indicator: above keyword(s) appearing in normal text

Strong indicator: high amount of camel-cased words

Listing 2: Web API "API" Indicator

Normal indicator: keyword(s) in URL: dev, doc, help, code, wiki, blog, lib, support, partner, lab, explain, affiliate

Strong indicator: keyword(s) appearing as separate tokens in the URL

Strong indicator: few external links (i.e. links pointing to different domains)

Strong indicator: high amount of camel-cased words

Listing 3: Web API "Documentation" Indicator

We differ between normal indicators and weak, medium or strong indicators: the occurrence of strong indicators in a page is weighted higher in the calculation of the score of a page than the occurrence of weak indicators, etc. This 'manual' term frequency approach to identify Web APIs on the Web requires a frequent tuning of the parameters and the scores of the individual indicators in order to find out the best configuration to detect Web APIs with a high probability. In this sense it is an advantage to have three separate indicators that are all individually configurable.

In our current configuration we classify a Web resource as a Web API when the three individual scores are above 0.4 (all scores being between 0 and 1). It is not enough if only two of the indicators are above our threshold, as shown by the following example: http://java.sun.com/j2se/1.4.2/docs/api/index.htm (Java API Documentation) fulfills both the “Documentation” and the “API” indicators, but it is nevertheless not a Web API. In Section **Error! Reference source not found.** we present an initial evaluation of our approach, which we will need to further analyze, extend and improve in the future.

2.3.2 Integration Into Crawler

We have extended the Service Crawler by two new processors that analyze whether fetched HTML pages can be Web APIs or not: the SVMClassifier processor that implements the Automatic Classification Approach (using the open-source Data Mining software RapidMiner[7] for the classification) and the WebAPIEvaluator processor that implements the Term Frequency Approach.

Each HTML page that is fetched by the crawler is evaluated by these two processors (at crawl run-time) and both the HTML meta-data (e.g., number of camel-case tokens) and the resulting scores are stored as RDF meta-data (Section 3 provides more details on the resulting RDF meta-data).

2.4 Semantic Web Services Crawling

This section introduces our approach to crawl and identify Semantic Web Services. Instead of only enabling users to produce semantic annotations to services in the SOA4All Studio, we actively crawl the Web for Semantic Web Services that are already published on the Web.

The crawler currently crawls for two different types of services, WSDLs and Web APIs, which results in two separated crawl outputs:

- Web APIs: archives containing the documents identified up to a given probability as Web APIs. Together with the archives comes a batch of RDF triple data that contains the metadata stored with the Web APIs.
- WSDLs: the crawler fetches both WSDLs and related information from the Web. Two types of delivered archives: archives containing all the WSDL service descriptions and archives containing all the related documents.

In the crawler Web APIs files we can find annotations with microformats. We will focus on microformats which include semantic information in service descriptions. We will work with a microformat called HTML for RESTful Services, in short hRESTS [7], for machine-readable descriptions of Web APIs, backed by a simple service model in RDF. The hRESTS microformat captures machine-processable service descriptions, building on the HTML service documentation aimed at developers. We will consider MicroWSMO, an extension of hRESTS that adds means for Semantic Web Service automation. hRESTS can support other extensions as well, such as SA-REST [8], intended for enabling tool support, especially faceted browsing and discovery of services by client developers.

On the other side, in the crawled WSDL files, we can include semantic annotations by using SAWSDL[8]. The Semantic Annotations for WSDL and XML Schema (SAWSDL) defines mechanisms for adding semantic annotations to WSDL components. SAWSDL does not specify a language for representing the semantic models, e.g. ontologies. Instead, it provides

[7] http://rapid- i.com/content/blogcategory/38/69/

[8] http://www.w3.org/TR/sawsdl/

mechanisms, by which concepts from the semantic models that are defined either within or outside the WSDL document, can be referenced from within WSDL components as annotations. These semantics when expressed in formal languages can help disambiguate the description of Web services during automatic discovery and composition of the Web services.

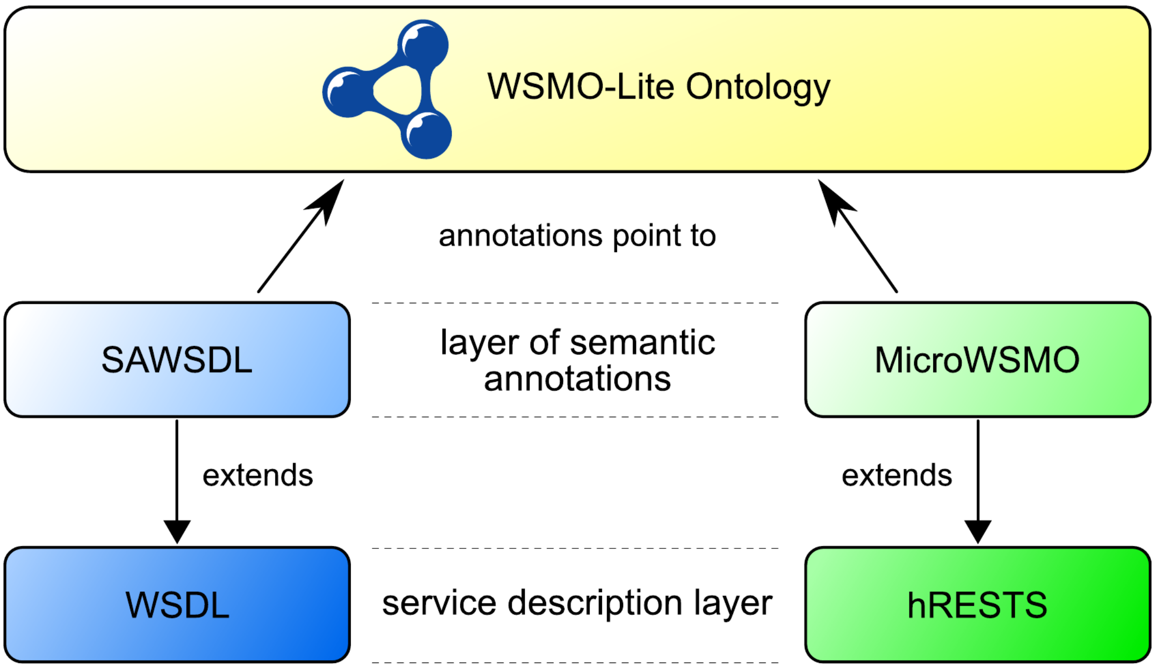

Figure 2-1: Service Semantic Layers

Figure 2-1 illustrates the relation of MicroWSMO to SAWSDL, along with their positioning among the various service description specifications. MicroWSMO is a SAWSDL-like layer on top of hRESTS. WSMO-Lite [9] specifies an ontology for the content of SAWSDL annotations in WSDL; MicroWSMO annotations in hRESTS also point to instances of the WSMO-Lite ontology, since it captures service semantics independently of the underlying Web service technology (WSDL/SOAP or REST/HTTP).

In effect, MicroWSMO is on a layer below WSMO-Lite, even though both use the acronym "WSMO" in their names.

2.4.1 Semantic Documents Identification Techniques

In the Semantic Web Service crawling we need to identify the type of the files crawled and focus on Semantic Web Service descriptions. The work here is to extract the RDF from SAWSDL, microWSMO, WSMO-Lite, etc. descriptions which potentially might be found on the Web. In order to automatically populate that information to semantic spaces two problems need to be solved:

- given a resource (byte stream), automatically determine which kind of semantic service description is contained
- given a document and its type (e.g. SAWSDL) extract the relevant service data as RDF triples

To solve the outlined problems two components have been developed: the Semantic File Identifier and the Semantic Annotator Extractor.

The developed Semantic File Identifier is file-structure oriented. We work with the idea that the more complex file types have a rigorous format specification associated with it. By examining a file's structure and comparing it with known format specifications, it should be possible to determine a file's type. Media files typically have in common that they have at least some kind of header section, containing certain information – or metadata – about the file, such as its type, its creation date, followed by a series of data "chunks" containing the actual data specific to the particular content. There are currently several file type identification tools that look at the header section to determine its type. In our case, we use the FileExt[9] tool in order to extract the "identifying characters"[10] (in WSDL files this is: Hex: 3C, ASCII: .) or the "MIME Type" to identify in the current version of the prototype, HTML and WSDL files associated with REST and WSDL services respectively.

In the case of HTML identified files, we will use an openly available XSLT style sheet[11] that parses HTML documents with hRESTS and MicroWSMO microformat mark-up to produce the RDF form of the service description data. To carry out the parsing task, first we use JTidy[12] to transform the files to XTHML format and in a second step we apply to the XHTML file the style sheet using Xalan[13] as engine.

In the case of WSDL files we use the WSDL Parser included in the WSDL4Java library for the parsing process.

Two more formats that we will identify using the Semantic File Identifier are WSML and OWL-S[14]. To parse the files to be able to further treat the files we will use wsmo4j[15] and the OWL-S API[16]

In all cases after the parsing process we obtain the semantic annotations in RDF format and these annotations are stored in the semantic spaces in order to provide semantic search from discover point of view. All these subcomponents mentioned above have been represented in Figure 2-2.

[9] http://filext.com/

[10] unique characters at the beginning of a file

[11] http://cms-wg.sti2.org/TR/d12/v0.1/20081202/xslt/hrests.xslt

[12] http://jtidy.sourceforge.net/

[13] http://xml.apache.org/xalan-j/

[14] http://www.w3.org/Submission/OWL-S/

[15] http://wsmo4j.sourceforge.net/index.html

[16] http://www.daml.ri.cmu.edu/owlsapi/

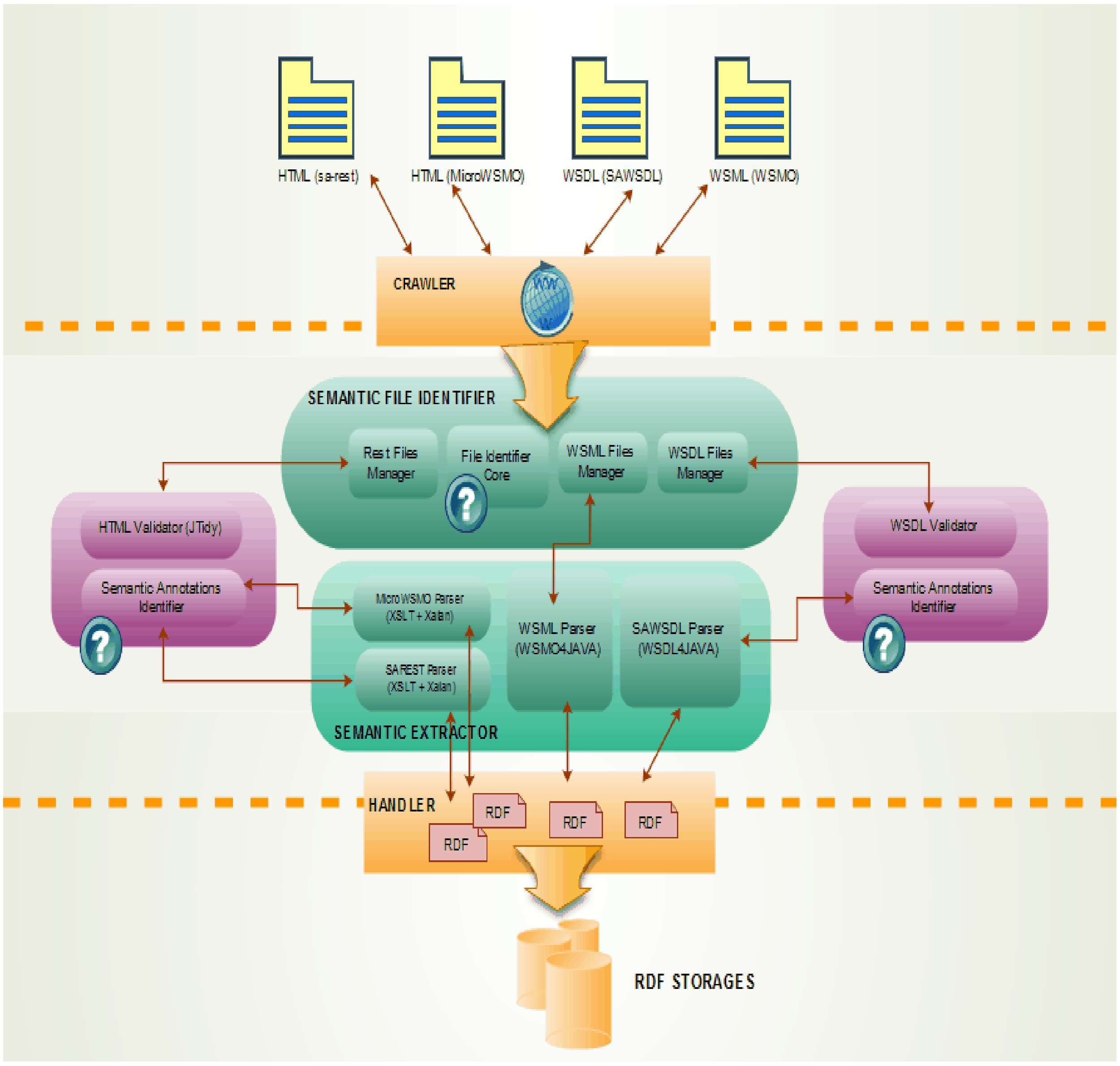

Figure 2-2: SA Identification and Extraction Components

2.4.2 Integration into Crawler

The implementation of the semantic service identification will be integrated into the service crawler while the property extraction will be implemented as one of the crawl post-processing steps. The Semantic File Identifier is thus integrated as single processor into the crawler, similar to the new processors developed for the WSDL and the Web API crawling processes.

3. Extracting Semantic Services From Web Data

The Service Crawler gathers different types of Web Services from the Web, as described in Section 2: WSDL services and related documents, Web APIs and Semantic Web Services (supporting different formats). We do not forward the raw crawled data to the other discovery components but first analyze the fetched documents, build unique service objects and store service meta-data as RDF triples. This process distinguishes the SOA4All Service Crawler from regular search engines (as, e.g., Google) that only deliver lists of raw documents as search results to the user (e.g. WSDL files, HTML pages, etc.).

In the following we will first provide a follow-up of the ontologies that we use within SOA4All to store the crawl meta-data (Section 3.1) and will then describe what meta-data we extract from the different service types (Section 3.2). Section 3.3 will conclude with an explanation on where the different data, i.e. the documents and the meta-data, is stored.

3.1 WSMO-Lite and Its Extensions

The main format used within SOA4All to semantically annotate services is WSMO-Lite. Within the crawling component we declare each detected WSDL service or Web API as WSMO-Lite service. This service is equivalent to the Service as defined in the Service-Finder Service Ontology[17]. The latter is one of the two ontologies that we use to describe the crawl data (and meta-data); the second ontology that we use is the seekda Crawl Ontology[18]. Figure 3 provides an overview of those parts of the two ontologies that we use within SOA4All and also outlines the relation to WSMO-Lite.

It is important to note that, on the one side the WSMO-Lite ontology, and on the other side the Service-Finder Service Ontology and the seekda Crawl Ontology are complementary, i.e. their schema does not overlap as they describe different aspects and parts of the services. The Service-Finder Service Ontology is mainly used to build unique service objects and relate them to their provider and to any related documents. As can be seen in **Error! Reference source not found.**, all related documents are tied to the services via annotations. This is as well true for the WSDL specifications, which are seen as another related document to the abstract service object.

The seekda Crawl Ontology has been developed to structure the Web API related meta-data. In this case each Web API becomes a unique service that is then annotated with document classifications. Furthermore we store some information that is used for the calculation of the three Web API indicators (e.g. number of camel-cased tokens, etc.). The Agent from the Service-Finder Service Ontology is populated by two new components: the SVMClassifier and the WebAPIEvaluator. The agents are used to store where the single annotations come from (i.e. from which component).

[17] http://www.service-finder.eu/ontologies/ServiceOntology

[18] http://seekda.com/ontologies/CrawlOntology.rdfs

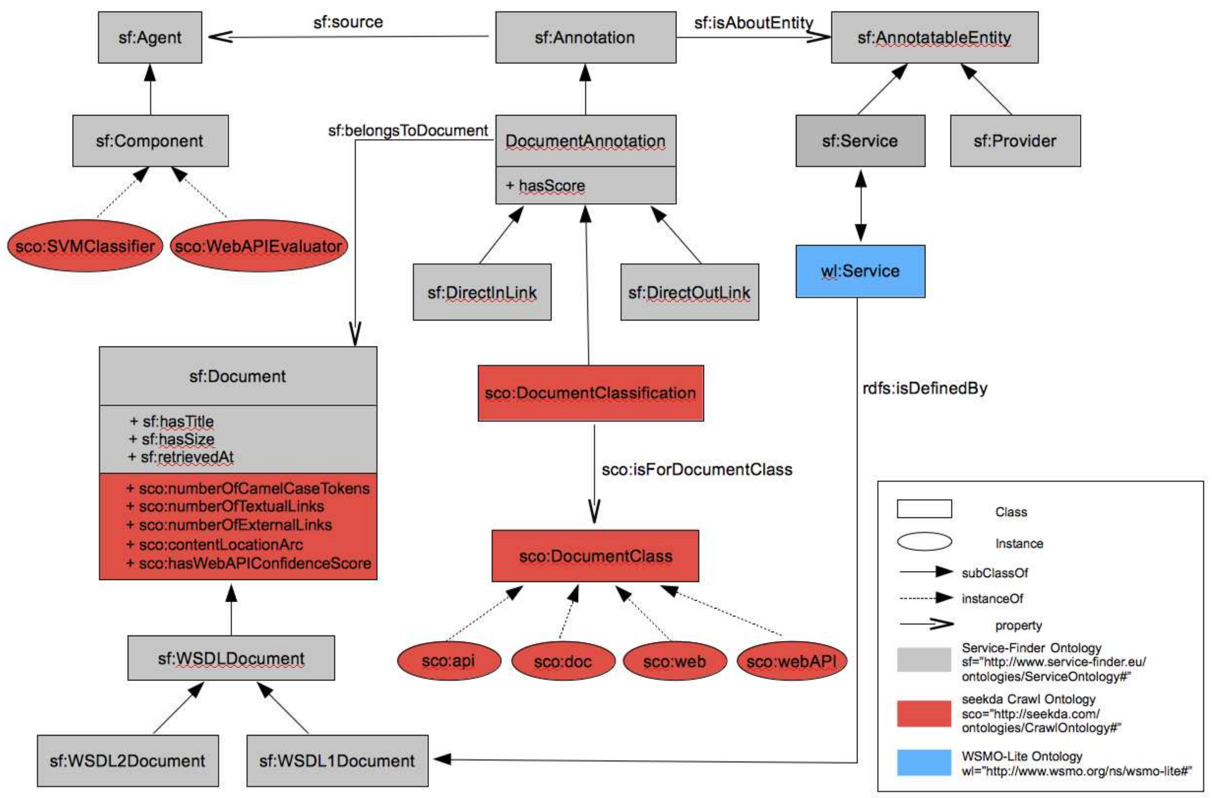

Figure 3: Ontologies Used for Crawl Data

3.2 Structure and Content of Web Resources and Their Meta-data

Depending on the type of service, i.e. WSDL service, Web API or SWS, we extract and store partly different meta-data, as will be described in the following. We will use the following namespaces and prefixes in the subsections:

- *Service-Finder Service Ontology:* `sf` – "http://www.service-finder.eu/ontologies/ServiceOntology#"
- *seekda Crawl Ontology:* `sco` – http://seekda.com/ontologies/CrawlOntology#
- *XML Schema:* `xsd` – http://www.w3.org/2001/XMLSchema#

3.2.1 WSDL descriptions

Regarding WSDL-based services we store meta-data concerning the service, its provider and its related documents. In a first step we create a unique service object: as one service might contain several WSDLs (e.g., hosted on different servers, testing WSDLs, etc.), we choose the best one of these (and only export this one in the crawl data output). In our definition the best WSDL is the one who's URL contains the provider domain and/or is the shortest one of all WSDLs belonging to this service.

To get the provider of a service we apply an approximation: we choose the effective top level domain[19] of the endpoint (not of the domain where the WSDL is found, i.e. hosted). If two

[19] https://wiki.mozilla.org/Gecko:Effective_TLD_List

providers implement a particular interface, we consider this as two services, as prices and Service Level Agreements might be different. The unique service object is identified by a newly constructed identifier that contains the domain of the provider and the local name of the portType/interface of the WSDL.

The WSDL-based service meta-data is stored using elements of the Service-Finder Service Ontology and of the seekda Crawl Ontology, as shown in Listing 4. There are four types of annotations that can be applied to WSDL-based service objects: WSDL annotations, outlink annotations (i.e. an annotation to a document to which one of the services' WSDL files links), inlink annotations (i.e. an annotation to a document that itself links to one of the services' WSDL files) or term vector similarity annotations (i.e. an annotation to a document that is not related by a direct link to or from the WSDL files, but that is classified as related by the term vector similarity analysis). Finally we use the seekda Crawl Ontology to store the location of the single documents within the crawl archives.

```
<serviceURI> <rdf:type> <sf:Service>

<serviceURI> <sf:hasProvider> <providerURI>
<providerURI> <rdf:type> <sf:Provider>
<providerURI> <sf:hasDomain> <providerDomain>
<providerDomain> <rdf:type> <sf:Domain>

<wsdlURI> <rdf:type> <sf:WSDLDocument>
<wsdlAnnotationID> <rdf:type> <sf:DocumentAnnotation>
<wsdlAnnotationID > <sf:belongsToDocument> <wsdlURI>
<wsdlAnnotationID > <sf:isAboutEntity> <serviceURI>

<outlinkURI> <rdf:type> <sf:Document>
<outlinkAnnotation> <rdf:type> <sf:DirectOutLink>
<outlinkAnnotation> <sf:belongsToDocument> <outlinkURI>
<outlinkAnnotation> <sf:isAboutEntity> <serviceURI>

<inlinkURI> <rdf:type> <sf:Document>
<inlinkAnnotation> <rdf:type> <sf:DirectInLink>
<inlinkAnnotation> <sf:belongsToDocument> <inlinkURI>
<inlinkAnnotation> <sf:isAboutEntity> <serviceURI>

<tvURI> <rdf:type> <sf:Document>
<tvAnnotation> <rdf:type> <sf:TermVectorSimilarityAssociation>
<tvAnnotation> <sf:belongsToDocument> <tvURI>
<tvAnnotation> <sf:isAboutEntity> <serviceURI>

<wsdlURI> <sco:contentLocationArc> <xsd:string>
<outlinkURI> <sco:contentLocationArc> <xsd:string>
<inlinkURI> <sco:contentLocationArc> <xsd:string>
<tvURI> <sco:contentLocationArc> <xsd:string>
```

Listing 4: WSDL-based service meta-data

3.2.2 Web API Homepages

Similar as for WSDL-based services we start building a unique service object from the discovered Web APIs. Again its identifier is containing the provider domain, but as we cannot know the service name we attach the hash value of the Web API URL to the identifier. As other meta-data we store criteria that we use for the single classification indicators (corresponding to the HTML features), as well as the Web API scores that are calculated by the two crawler classifiers (as described in Section 2.3): SVM classifier and Web API Evaluator. For convenience reasons we also provide one general Web API confidence score that is built from the single scores of the two classifiers. Listing 5 shows the Web API related meta-data.

```
<serviceURI> <rdf:type> <sf:Service>

<serviceURI> <sf:hasProvider> <providerURI>
<providerURI> <rdf:type> <sf:Provider>
<providerURI> <sf:hasDomain> <providerDomain>
<providerDomain> <rdf:type> <sf:Domain>

<documentURI> <rdf:type> <sf:Document>
<documentURI> <sf:hasSize> <xsd:number>
<documentURI> <sf:retrievedAt> <xsd:date>
<documentURI> <sf:hasTitle> <xsd:string>
<documentURI> <sco:numberOfCamelCaseTokens> <xsd:number>
<documentURI> <sco:numberOfTextualLinks> <xsd:number>
<documentURI> <sco:numberOfExternalLinks> <xsd:number>

<documentClassification> <rdf:type> <sco:DocumentClassification>
<documentClassification> <sf:belongsToDocument> <documentURI>
<documentClassification> <sco:isForDocumentClass> <WebAPI>
<documentClassification> <sf:hasScore> <xsd:number>
<documentClassification> <sf:source> <sco:SVMClassifier>

<documentClassification> <rdf:type> <sco:DocumentClassification>
<documentClassification> <sf:belongsToDocument> <documentURI>
<documentClassification> <sco:isForDocumentClass> <api>
<documentClassification> <sf:hasScore> <xsd:number>
<documentClassification> <sf:source> <sco:WebAPIEvaluator>

<documentClassification> <rdf:type> <sco:DocumentClassification>
<documentClassification> <sf:belongsToDocument> <documentURI>
<documentClassification> <sco:isForDocumentClass> <doc>
<documentClassification> <sf:hasScore> <xsd:number>
<documentClassification> <sf:source> <sco:WebAPIEvaluator>

<documentClassification> <rdf:type> <sco:DocumentClassification>
<documentClassification> <sf:belongsToDocument> <documentURI>
<documentClassification> <sco:isForDocumentClass> <web>
<documentClassification> <sf:hasScore> <xsd:number>
<documentClassification> <sf:source> <sco:WebAPIEvaluator>

<documentURI> <sco:hasWebAPIConfidenceScore> <xsd:number>
```

Listing 5: Web API service metadata

3.2.3 Semantic Web Services

Next, we describe the different semantic Web services approaches, and analyze in a nutshell the followings kind of semantic service descriptions: SAWSDL, hRESTS, MicroWSMO, WSMO-Lite, OWL-S and WSML.

We consider WSMO-Lite and MicroWSMO, two related lightweight approaches to semantic Web service description, evolved from the WSMO framework. WSMO-Lite uses SAWSDL to annotate WSDL-based services, whereas MicroWSMO uses the hRESTS microformat to annotate RESTful APIs and services.

In RESTful services, a service is a grouping of related Web resources, each of which provides a part of the overall service functionality. While a SOAP service has a single location address, each operation of a RESTful service must have an address of the concrete resource that provides this operation. Therefore, hRESTS also defines classes for marking the resource address and the HTTP method used on that resource (the classes address and method), which together uniquely identify a single operation. hRESTS is an approach with wider goals than SWS automation, and it does not contain links for semantic annotations. Consequently, MicroWSMO extends hRESTS with SAWSDL-like annotations: the HTML class “mref” identifies the model reference annotations, and the link relations *lifting* and *lowering* identify the data grounding transformations. The concrete semantics are added analogously to how WSMO-Lite annotates WSDL documents: functional and nonfunctional semantics are model references on the service, behavioral semantics are captured using functional descriptions of operations, and information model links go on the input and output messages.

From the point of view of WSDL-based services we can distinguish between a concrete service and its abstract interface that defines the operations. This structure is annotated using SAWSDL annotations with different kinds of semantics. The following paragraphs describe how the various types of semantics are attached in the WSDL structure:

- Functional semantics can be attached as a model reference either on the WSDL service construct, concretely for the given service, or on the WSDL interface construct, in which case the functional semantics apply to any service that implements the given interface. Nonfunctional semantics, by definition specific to a given service, are attached as model references directly to the WSDL service component.
- Information semantics are expressed in two ways. First, pointers to the semantic counterparts of the XML data are attached as model references on XML Schema element declarations and type definitions that are used to describe the operation messages. Second, lifting and lowering transformations need to be attached to the appropriate XML schema components: input messages (going into the service) need lowering annotations to map the semantic client data into the XML messages, and output messages need lifting annotations so the semantic client can interpret the response data.
- Finally, behavioral semantics of a service are expressed by annotating the service's operations (within the WSDL interface component) with functional descriptions, so the client can then choose the appropriate operation to invoke at the appropriate time during its interaction with the service.

A WSDL document with WSMO-Lite annotations can be validated for consistency and completeness. When mapping such a WSDL document into our simplified service model, which does not represent a separate service interface, we combine the interface annotations with the service annotations. Similarly, annotations from the appropriate XML Schema components are then mapped to annotations of the messages in our service model. Otherwise, the mapping of WSDL to our service model is straightforward.

In the post-processing step we extract the above described semantic annotations as RDF triples, relate them to the corresponding services and store them in the semantic spaces using the storage API.

Following with the WSDL services, we consider those OWL-S constructs, which are appropriate for use with the various SAWSDL annotations.

As noted in [11], the principal high-level objectives of OWL-S are (i) to provide a general-purpose representational framework in which to describe Web Services; (ii) to support automation of service management and use by software agents; (iii) to build, in an integral fashion, on existing Web Service standards and existing Semantic Web standards; and (iv) to be comprehensive enough to support the entire life cycle of service tasks.

OWL-S (formerly known as DAML-S) is an OWL ontology [12] that includes three primary subontologies: the service profile, process model, and grounding. The service profile is used to describe what the service does; the process model is used to describe how the service is used; and the grounding is used to describe how to interact with the service. The service profile and process model are thought of as abstract characterizations of a service, whereas the grounding makes it possible to interact with a service by providing the necessary concrete details related to message format, transport protocol, and so on.

We are concerned with the use of constructs of the profile and process model as referents of SAWSDL annotations. Because we adopt a perspective centered around WSDL and SAWSDL, there is no need to employ the OWL-S grounding.

OWL-S's grounding reflects an OWL-S perspective; that is, it is motivated by use cases in which service processing, tools, and reasoning of various kinds are organized around OWL-S.

Finally, we consider WSML services. In this context, the Web Service Modeling Ontology WSMO [10] provides a conceptual model for the description of various aspects of Services towards such SemanticWeb Services (SWS). In particular, WSMO distinguishes four top-level elements:

- Ontologies: Ontologies provide formal and explicit specifications of the vocabularies

used by the other modeling elements. Such formal specifications enable automated processing of WSMO descriptions and provide background knowledge for Goal and Web Service descriptions.

- Goals: Goals describe the functionality and interaction style from the requester perspective.
- Web Service descriptions: Web Service descriptions specify the functionality and the means of interaction provided by the Web Service.
- Mediators: Mediators connect different WSMO elements and resolve heterogeneity in data representation, interaction style and business processes.

WSML makes a clear distinction between the modeling of the different conceptual elements on the one hand and the specification of complex logical definitions on the other. To this end, the WSML syntax is split into two parts: the conceptual syntax and logical expression syntax. The conceptual syntax was developed from the user perspective, and is independent from the particular underlying logic; it shields the user from the peculiarities of the underlying logic. Having such a conceptual syntax allows for easy adoption of the language, since it allows for an intuitive understanding of the language for people not familiar with logical languages. In case the full power of the underlying logic is required, the logical expression syntax can be used. There are several entry points for logical expressions in the conceptual syntax, namely, axioms in ontologies and capability descriptions in Goals and Web Services.

3.3 Data Storage

As result of each crawl iteration we obtain (a) the fetched documents and (b) meta-data corresponding to the services (as described in Section 3.2). The data is stored in two different ways (as can as well be seen in **Error! Reference source not found.** within Section 1):

- documents: the relevant fetched documents (i.e. WSDLs, related documents, Web APIs and SWS descriptions) are stored within the Document Repository, i.e. the DRAGON Service Registry. More information on the Document Repository can be found in [13].
- meta-data: all the meta-data provided by the crawling component is stored as RDF triples and will be added to the SOA4All Semantic Spaces.

4. API For Accessing Crawled Data

To store the fetched documents and the RDF meta-data resulting from the different crawls we need to be able to access this data in an easy and comprehensive way. This is why we have developed a Crawl API that is implemented as RESTful SOA4All platform service and is thus plugged to the SOA4All Distributed Service Bus.

The Crawl API is used from both WP5 components and other SOA4All components to access the crawl data. It supports access to different versions of crawl data (i.e. data resulting from different crawl iterations) and allows the extraction of single services (WSDLs or RESTful services) and their related information. Furthermore, it provides a simple SPARQL endpoint that allows each component to freely query the RDF crawl meta-data.

In the following we provide an updated overview of the Crawl API that had initially be presented in [1]:

1. list crawler sets (IDs)
   - method: GET
   - url: CRAWLER_API_URL/sets?start=xxx&count=yyy
   - result: XML (crawler set IDs)

2. list services
   - method: GET
   - url: CRAWLER_API_URL/sets/<set-id>/webservices?start=xxx&count=yyyy
   - result: XML (service IDs)

3. get service descriptions:
   - GET
   - CRAWLER_API_URL/sets/<set-id>/webservices/<service-id>
   - ZIP archive with all description files for the service

4. get ALL related documents
   - GET
   - CRAWLER_API_URL/sets/<set-id>/webservices/<service-id>/related
   - ZIP archive with all the related documents

5. get service metadata
   - GET
   - CRAWLER_API_URL/sets/<set-id>/webservices/<service-id>/metadata
   - XML (RDF)

6. execute a SPARQL query
   - GET
   - CRAWLER_API_URL/sets/<set-id>/query?q=<sparql-expr>
   - XML (RDF)

5. Conclusions

In this deliverable we have presented the second crawler prototype. We have provided an overview of the three different types of services that we crawl the Web for: WSDL services and any related information, Web APIs and Semantic Web Service descriptions. We have explained the identification techniques that are applied within the crawler to detect the services and have shortly outlined their implementation-wise integration into the Service Crawler.

Both during the crawls and in a post-processing process we extract service meta-data corresponding to the discovered services. We have provided a detailed overview of our process of building unique service objects and of the meta-data that we store for the single types of services. We have outlined what ontologies we use for storing the meta-data and have explained the relation among the different ontologies that we use in the crawling component and the main SOA4All service ontology, WSMO-Lite.

In a last step we have explained where the crawled data is stored and how it is accessed by the different SOA4All components (including the discovery components).

6. References

[1] Manuel Brunner, Nathalie Steinmetz, Oliver Fabre, Marin Dimitrov and Iván Martinez. D5.1.2 First Crawler Prototype. SOA4All deliverable, March 2009.

[2] Nathalie Steinmetz, Holger Lausen and Martin Kammerlander. D2.1 – Crawling Research Report – version 1. Service-Finder deliverable, 2008. Available from: http://www.service-finder.eu/attachments/D2.1.pdf.

[3] Roy Thomas Fielding. Architectural Styles and the Design of Network-based Software Architectures. PhD Thesis, 2000, University of California, Irvine.

[4] Fabrizio Sebastiani. Machine Learning in Automated Text Categorisation. 1999, Consiglio Nazionale delle Ricerche. Available from: http://dienst.isti.cnr.it/Dienst/UI/2.0/Describe/ercim.cnr.iei/1999-B4-31-12

[5] Marie-Francine Moens. Information Extraction: Algorithms and Prospects in a Retrieval Context. Springer, 2006. ISBN: 978-1-4020-4987-3.

[6] Saartje Brockmans, Irene Celino, Dario Cerizza, Emanuele Della Valle, Michael Erdmann, Adam Funk, Holger Lausen, Wolfgang Schoch, Nathalie Steinmetz and Andrea Turati. D7.1 Testing Scenarios and Evaluation Criteria Preparation for Alpha Release. Service-Finder deliverable, 2008. Available from: http://www.service-finder.eu/attachements/D7.1.pdf.

[7] Jacek Kopecky, Tomas Vitvar and Dieter Fensel. MicroWSMO and hRESTS. D3.4.3 SOA4All deliverable, March 2009.

[8] Amit P. Sheth, Karthik Gomadam, and Jon Lathem. SA-REST: Semantically Interoperable and Easier-to-Use Services and Mashups. IEEE Internet Computing,11(6):91–94, 2007.

[9] Tomas Vitvar, Jacek Kopeck´y, and Dieter Fensel. WSMO-Lite: Lightweight Semantic Descriptions for Services on theWeb. CMS WGWorking Draft, February 2009. Available at http://cms-wg.sti2.org/TR/d11/.

[10] D. Roman, U. Keller, H. Lausen, R. L. Jos de Bruijn, M. Stollberg, A. Polleres, C. Feier, C. Bussler, and D. Fensel. Web service modeling ontology. *Applied Ontology*, 1(1):77.106, 2005.

[11] D. Martin, M. Burstein, D. McDermott, S. McIlraith, M. Paolucci, K. Sycara, D. McGuinness, E. Sirin, and N. Srinivasan, "Bringing Semantics to Web Services with OWL-S", to appear in the World Wide Web Journal, Vol. 10, No. 3, Sept. 2007.

[12] D. L. McGuinness and F. van Harmelen, "OWL Web Ontology Language Overview", W3C Recommendation, 10 February 2004; http://www.w3.org/TR/2004/REC-owl-features-20040210/.

[13] Sudhir Agarwal, Martin Junghans, Olivier Fabre and Ioan Toma. D5.3.1 First Service Discovery Prototype. SOA4All deliverable, August 2009.

[14] Daniel Bachlechner, Katharina Siorpaes, Holger Lausen and Dieter Fensel. Web service discovery – a reality check. In 3rd European Semantic Web Conference, 2006.

[15] Holger Lausen and Thomas Haselwanter. Finding Web Services. In 1st European Semantic Technology Conference, 2007.

[16] Holger Lausen and Nathalie Steinmetz. Survey of current means to discover Web Services. Technical report, STI Innsbruck, 2007.

[17] Nathalie Steinmetz and Ioan Toma (Eds). D16.1 v1.0 WSML Language Reference. August 2008. Available at: http://www.wsmo.org/TR/d16/d16.1/v1.0/

Annex A.

Paper title: **“Web Service Search on Large Scale”.**

The following paper describing our WSDL and Web API crawling approaches has been accepted at the International Conference on Service Oriented Computing (ICSOC) 2009, November 2009, Stockholm, Sweden:

Authors: Nathalie Steinmetz (UIBK), Holger Lausen (SEEKDA) and Manuel Brunner (SEEKDA).

Link: http://www.sti-innsbruck.at/results/browse/conference-papers/details/?uid=395