Expert Systems with Applications 38 (2011) 6657–6668

Contents lists available at ScienceDirect

Expert Systems with Applications

journal homepage: www.elsevier.com/locate/eswa

A combinatory framework of Web 2.0 mashup tools, OWL-S and UDDI

G. Meditskos *, N. Bassiliades

Department of Informatics, Aristotle University of Thessaloniki, 54124 Thessaloniki, Greece

ARTICLE INFO

Keywords:
Mashup tools
Semantic mashup discovery
OWL-S
UDDI

ABSTRACT

The increasing number of Web 2.0 applications, such as wikis or social networking sites, indicates a movement to large-scale collaborative and social Web activities. Users can share information, add value to Web applications by using them or aggregate data from different sources creating Web applications (mashups) using specialized tools (mashup tools). However, Web 2.0 is not a new technology, but it rather embraces a new philosophy, treating the Internet as a platform. Several issues related to the Semantic Web vision, such as interoperability or machine understandable data semantics, are not tackled by Web 2.0. In this paper, we present our effort to combine semantic Web services (SWS) discovery frameworks, UDDI repositories and existing mashup tools in order to enhance the procedure of developing mashups with semantic mashup discovery capabilities. Towards this end, we introduce a social-oriented extension of OWL-S advertisements, their mapping algorithm on UDDI repositories and a semantic mashup discovery algorithm. Finally, we elaborate on the way our framework has been realized using the Yahoo Pipes mashup tool.

1. Introduction

The Web 2.0 paradigm is frequently used by those who question the feasibility of Semantic Web as a tangible proof that Web can live without "heavy" semantic technologies and complex frameworks. Web 2.0 offers flexible and widely adopted applications, such as wikis and social bookmarking sites, with tagging being one of the main means for specifying metadata information. One of the most exciting Web 2.0 applications are the *mashup tools* that are used for creating *mashups*, applications that combine several external data sources, such as Web services or RSS feeds. Mashup tools give the opportunity to users to select and combine services, creating a Web application with added value. However, although these tools hide execution aspects from users, there is still an open issue of determining which service is suitable to users' needs, since the selection is performed manually based on keyword-based searches. We argue that semantic Web technologies, such as ontologies and the research that has been done on semantic Web services (SWS) can help users to define better their requirements.

In this work we reuse the existing infrastructure of mashup tools for combining and executing services on the Web and we enhance it with SWS discovery capabilities in UDDI[1] repositories using OWL-S advertisements. The general idea is to use a *lightweight* and *social-oriented* version of OWL-S advertisements that we call SOOWL-S, in order to describe conceptually and abstractly mashups in the Web 2.0 domain. In that way, we can exploit the research that has been done on SWS frameworks and discovery algorithms in the mashup tool domain, enabling mashup tool users to semantically filter out and publish mashups.

The contributions of our work can be summarized in the following:

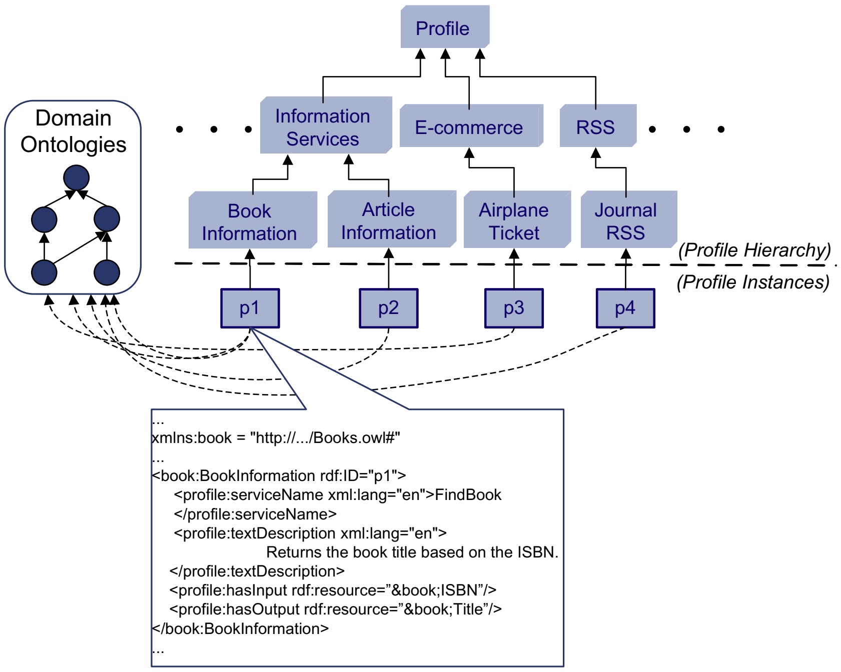
- We introduce the notion of SOOWL-S advertisements, a social-oriented version of OWL-S advertisements that are used as the ontology-based means for providing semantic descriptions of mashups, viewing them abstractly as black boxes.
- We define an algorithm for mapping SOOWL-S advertisements on UDDI databases, enabling the standard-based, collaborative and distributed management of mashup semantic descriptions.
- We define a mashup matchmaking algorithm based on SOOWL-S advertisements, allowing the semantic retrieval of mashup descriptions from UDDI repositories.

The rest of the paper is structured as follows: in Section 2 we describe the basic background related to SWS frameworks, UDDI registries and mashup tools, and we state our motivation. In Section 3 we present the architecture of our framework, showing the way the current mashup tool architecture can be enhanced with semantic mashup discovery capabilities. In Section 4 we describe the SOOWL-S advertisements and their mapping algorithms to UDDI constructs. In Section 5 we introduce the notion of the MashUDDI registry and we elaborate on the mashup matchmaking algorithm that is employed. Section 6 presents a prototype

* Corresponding author.
E-mail addresses: gmeditsk@csd.auth.gr (G. Meditskos), nbassili@csd.auth.gr (N. Bassiliades).

[1] http://www.oasis-open.org/committees/uddi-spec/.

0957-4174/$ - see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.eswa.2010.11.072



```
...
xmlns:book = "http://.../Books.owl#"
...
<book:BookInformation rdf:ID="p1">
    <profile:serviceName xml:lang="en">FindBook
    </profile:serviceName>
    <profile:textDescription xml:lang="en">
                    Returns the book title based on the ISBN.
   </profile:textDescription>
   <profile:hasInput rdf:resource="&book;ISBN"/>
   <profile:hasOutput rdf:resource="&book;Title"/>
</book:BookInformation>
...
```

Fig. 1. An example of an OWL-S Profile hierarchy with four advertisements.

implementation of our framework in the Yahoo Pipes mashup tool. Finally, in Sections 7 and 8, we review related work and we conclude, respectively.

2. Background and motivation

In this section we present the basic background related to SWS frameworks, the UDDI standard and mashup tools, and we state our motivation for combining them in a single framework.

2.1. Semantic web service frameworks

Web services have given new potentials in the domain of Business-to-Business collaborations and Consumer-to-Business interactions, enabling easy and standardized communication among the involving parties (Tsalgatidou & Pilioura, 2002). However, as it happens in every dynamic and heterogeneous environment where many parties are involved, the selection of a Web service is a difficult task, mainly due to the large number of services.

The SWS paradigm aims at making Web services machine-understandable and use-apparent, utilizing Semantic Web technologies for service annotation and processing (Burstein et al., 2005; Medjahed & Bouguettaya, 2005). The main idea is to apply reasoning techniques (Baader, 2003; Sirin, Parsia, Grau, Kalyanpur, & Katz, 2007; Tsarkov & Horrocks, 2006) over ontology-based Web service descriptions in order to perform automated Web service discovery (semantic Web service discovery), composition (semantic Web service composition) and invocation. Towards this goal, many languages and frameworks have been proposed, such as OWL-S (Martin et al., 2007), WSMO (et al., 2006), SAWSDL (Kopecký, Vitvar, Bournez, & Farrell, 2007) and WSDL-S (Akkiraju et al., 2005). The first two define a conceptual, ontology-based framework under which Web services can be described and manipulated. More specifically, OWL-S provides four upper OWL ontologies, namely Service, Service Profile, Service Process and Service Grounding. In brief, the Profile concept of the Service Profile ontology, known also as the *advertisement* of a Web service, is used as input to SWS discovery engines and provides the information needed for an agent to discover a service based on semantic annotations regarding functional, e.g. inputs and outputs, and non-functional properties, e.g. price (O'Sullivan, Edmond, & ter Hofstede, 2002). The Service Process and Service Grounding provide information for an agent to make use of a service. Fig. 1 depicts a Profile-based annotation example using a custom subclass hierarchy of the Profile concept (OWL-S 1.1 release, 2004).

WSMO defines four top level elements, namely Ontologies, Goals, Web Services and Mediators. Ontologies provide the formal terminology definitions, Goals are formal specifications of objectives that a client aims to achieve by using Web services, Web Services are formal descriptions needed for automated service handling and usage and Mediators are the top level elements for handling heterogeneity.

Finally, the SAWSDL and WSDL-S approaches are based on the WSDL descriptions of Web services that annotate semantically with ontological knowledge.

2.2. UDDI registries

Universal Description, Discovery and Integration (UDDI) (UDDI.org, 2000) constitutes a Web service standard for publishing Web services. It is a platform-independent, XML-based registry that allows businesses to publish their Web services and to locate others via keyword search over various attributes, such as names or tModels. It can be viewed actually as a *meta-service* for locating Web services by enabling robust queries against rich metadata.

The importance of a standard-based registry (Dogac et al., 2005; Dustdar & Treiber, 2005) able to store the semantic descriptions of Web services has been quickly identified, leading to mapping approaches of SWS frameworks on the UDDI standard. In Srinivasan, Paolucci, and Sycara, 2004; Herzog, Zugmann, Stollberg, and Roman, 2007 the mapping of OWL-S advertisements and WSMO elements on UDDI tModels is discussed, respectively. Similar mappings have been defined for SAWSDL[2] and WSDL-S (Sivashanmugam, Verma, Sheth, & Miller, 2003). In that way, algorithms for SWS discovery

[2] http://lsdis.cs.uga.edu/projects/meteor-s/downloads/Lumina/.

and composition can operate over the semantic service descriptions that are stored in UDDI registries, allowing the integration of Web services from heterogeneous environments.

2.3. Web 2.0 mashups and mashup tools

The development of mashup applications requires a lot of effort: the desirable services should be located and the APIs should be implemented in order to realize the communication, especially in the case of RESTful Web services that do not follow any standard API, in contrast to SOAP Web services (Zur Muehlen, Nickerson, & Swenson, 2005). The Representational State Transfer (REST) design considers Web services as resources that are identified by their URLs. Clients communicate with the service using remote calls that describe the action that is to be performed. Thus, programmers need to manually implement all the necessary steps to consume the Web service. On the other hand, a SOAP Web service is easier to be consumed, using existing tools that automate the procedure. However, the REST approach is considered as a more lightweight approach than SOAP.

Several tools have emerged, called *mashup tools* that assist the end users to create mashups, such as the *Dapper*[3] and the *IBM Lotus Mashups*.[4] Some other tools offer a graphical, component-based way of wiring services together, such as the *Yahoo Pipes*[5] and the *Microsoft Popfly*.[6] In each mashup tool there is a set of preregistered services, whose invocation has been implemented at design-time, and users select and wire services together, creating complex Web applications (mashups). Furthermore, there is the possibility of creating advanced mashups by calling dynamically external Web services. For example, Yahoo Pipes provides a Web service module that acts as a meta-service, allowing the invocation of external Web services following the JSON[7] data format.

2.4. Motivation

We are motivated by the fact that mashup tools and SWS frameworks can serve complementary goals. Mashup tools offer the infrastructure for combining and executing real-world Web services. On the other hand, the SWS frameworks offer the standards and algorithms for semantically annotating, discovering and composing Web services. Therefore, *mashup tools can serve as a platform for applying and evaluating SWS frameworks and algorithms on real-world Web services*.

In this work, we target at the utilization of SWS frameworks for semantic mashup discovery and publish in mashup tools. The main idea is to enhance the procedure of selecting services in mashup tools with semantic capabilities that are offered by the SWS frameworks (service discovery). More specifically, there are three capabilities for service selection in mashup tools.

2.4.1. Predefined services

Mashup tools provide a predefined set of simple services (modules) that are used by users during the development of mashups. These services are listed in a taxonomy tree according to their functionality. For example, there are RSS feed categories or services that perform string transformations, such as string concatenations. Users browse the categories and select or combine the services that satisfy their requirements.

[3] http://www.dapper.net.
[4] http://www.ibm.com/software/lotus/mashups.
[5] http://pipes.yahoo.com/.
[6] http://www.popfly.com/.
[7] http://www.json.org/.

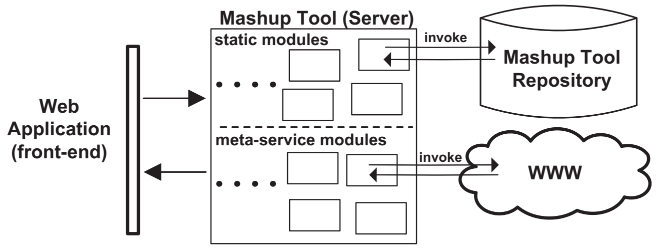


Fig. 2. The typical architecture of mashup tools.

2.4.2. Search

Mashup tools manage a directory where users publish their mashups, providing a set of keywords that characterize them. The search in such directories is performed with keywords, such as the *search for pipes* field of Yahoo Pipes, returning relevant mashups that can be reused in new applications. Note that there is no interoperability among the mashups of different mashup tools. Each mashup directory can be used only by the mashup tool that have been created with.

2.4.3. Meta-services

Mashup tools are able to call external Web services that follow a specific API using a meta-service module. For example, Yahoo Pipes provides a meta-service for calling JSON Web services. The URL of the Web service is provided as a parameter by the user who is responsible for finding the suitable Web service from available repositories.[8]

Our work targets at the introduction of semantic capabilities during the last two service/mashup selection procedures, using semantic service descriptions and discovery algorithms. More specifically, we describe a framework that uses the meta-service infrastructure of mashup tools in order to help users to semantically search for services. The meta-services communicate with special UDDI repositories that we call MashUDDI's that contain SOOWL-S advertisements, a variation of the typical OWL-S model that allows the collaborative semantic annotation of services in the Web 2.0 environment. The meta-services can be also used to semantically publish created mashups in a MashUDDI registry by providing semantic annotations, as if they were Web services. Furthermore, by organizing the services/mashups of mashup tools into a UDDI registry we give great potentials, both for the easier retrieval and management of services, and for the potential cooperation with other frameworks.

3. Semantically enhanced mashup tool architecture

A typical mashup tool architecture is depicted in Fig. 2 and involves the mashup tool server and the Web application through which the tool exposes its functionality. The mashup tool provides a set of *modules* that users may select in order to develop their mashups in a component-based way. In general, these modules can be categorized into *static* and *meta-service* modules. The static modules are bound to pre-registered services in the mashup tool repository and the communication with the services has been implemented at design-time. The meta-service modules are parameterized modules, e.g. they are not bound to specific services, that can communicate through a specific API with external data sources, e.g. RSS feeds or Web services.

Our vision of introducing semantic mashup discovery capabilities in mashup tools is depicted in Fig. 3. The mashup tool

[8] http://www.programmableweb.com/mashuplist.

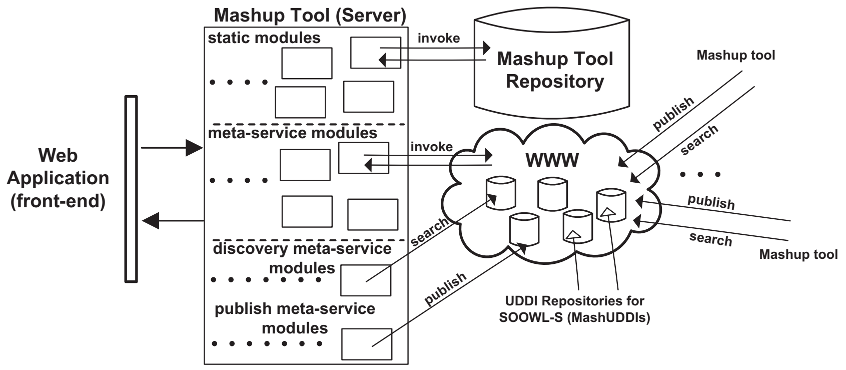


Fig. 3. Semantically enhanced mashup tool architecture.

preserves the existing functionality of communicating with local and external data sources, and it is enhanced with special *discovery* and *publish* meta-service modules that communicate with semantically extended UDDI repositories (MashUDDIs). These repositories, which are described in detail in Section 5, are distributed UDDI databases that store SOOWL-S advertisements and they can be used for the following activities.

- *Publish SOOWL-S advertisements.* Users should be able to semantically publish their mashups in a MashUDDI registry. Thus, the mashup tool should support a set of domain ontologies to play the role of the controlled annotation vocabulary and it should offer, through the Web interface, the necessary means to users for selecting the annotation constructs. The annotation activity follows the idea that each mashup can be considered as a black box that performs a specific task, for which we can define semantic annotations, as if it were a Web service.
- *Search for SOOWL-S advertisements.* Users should be able to query a MashUDDI in order to retrieve service/mashup descriptions following exactly the same procedure to publishing. For that reason, the MashUDDI registry should employ a *service matchmaking mechanism* in order to semantically retrieve the advertisements that match users' queries.

Note that we do not propose a new mashup tool architecture, but a general framework under which existing mashup tools can be enhanced with semantic capabilities. The advantage of such an approach is that it does not require any change to the architecture of mashup tools. Instead, it reuses the functionality that is already provided by mashup tools through the meta-service modules. Note also that the proposed framework enhances the interoperability among different mashup tools, as far as mashup publishing and discovery are concerned (Fig. 3). This is due to the UDDI-oriented nature of MashUDDIs that allows the distributed and standard-based management of semantic mashup descriptions.

3.1. Abstract semantic service/mashup annotations

The mashup tool paradigm has some unique characteristics that hamper the direct and unconditional utilization of SWS frameworks:

3.1.1. Service implementation

Mashup tools do not impose any restriction on the Web services they use: they can be either RESTful, which have their capabilities described in Web pages, or SOAP-based, providing a WSDL document. Moreover, few RESTful Web services have WSDL because the main objective with REST is simplicity (WSDL facilitates significant tooling support) (Sheth, Gomadam, & Lathem, 2007). Therefore, it is not so trivial to consider RESTful services in SWS frameworks, since there is not always a standard API to be considered during service annotation.

3.1.2. Service representation

The SWS standards target at the automation of several activities, such as invocation and composition and thus, the semantic descriptions are tightly-coupled with implementation aspects of the services. However, in mashup tools there is the need only for a conceptual description of services, since the invocation infrastructure is pre-implemented and the "composition" is performed manually by users.

3.1.3. Collaborative environment

In SWS frameworks the providers are responsible for the appropriate semantic annotations of their services. However, in the mashup tool paradigm there is a more collaborative (social) nature where many users can participate in the semantic annotation of services. This is an important feature that should be taken into consideration during the application of any service/mashup matchmaking algorithm.

We follow the assumption that in order to make practical the introduction of semantic service descriptions in mashup tools, a *conceptual* model should be followed (Preist, 2004). By the word "conceptual" we mean that, since the development of mashups is based on human intervention, the descriptions of services (a) can be disengaged from any implementation aspect, and (b) should be convenient, targeting at the average mashup tool user. These requirements can be satisfied by following a conceptual approach for service/mashup annotation, using only simple input, output and non-functional properties. In that way, both RESTful and SOAP-based (Fielding & Taylor, 2000) services can be annotated, independently of the API that they follow. For example, an RSS feed can be conceptually described in terms of an OWL-S advertisement without any input and with an output ontological concept characterizing the type of the information that it provides (Fig. 4). In that way, we are free to annotate the feed according to our perception, without being interested in the detailed mapping of the RSS/XML syntax to the Profile instance. Bear in mind that our goal is to assist mashup tool users by semantically filtering out services, preserving the existing mashup tool architecture, rather than introducing an automated mashup development procedure that would require a strong dependence of mashup tools on SWS frameworks.

In the rest of the discussion we describe the various aspects involved in a semantically enhanced mashup tool architecture, such as the need to incorporate SOOWL-S descriptions, the mapping of service/mashup SOOWL-S descriptions on MashUDDI registries and the matchmaking procedure that should be followed.

```
xmlns:jour = "http:///Journals.owl#"
xmlns:rss = "http:///RSS.owl#"
...
<jour:RSS rdf:ID="profile_1">
  <profile:serviceName xml:lang="en">CallForPapers_RSS_CS_Els
  </profile:serviceName>
  <profile:textDescription xml:lang="en">
    Call for papers RSS feed in Computer Science (Elsevier).
  </profile:textDescription>
  <profile:hasOutput rdf:resource="#_CallForPapersCompScEls"/>
  <rss:url>http://../CallForPapersComputerScienceElsevier</rss:url>
</jour:RSS>
<process:Output rdf:ID="_CallForPapersCompScEls">
  <process:parameterType rdf:datatype="&xsd;anyURI">&jour;CS_ELS_CFP
  </process:parameterType>
</process:Output>
```

Fig. 4. An example of an OWL-S advertisement for an RSS feed.

4. Social-oriented OWL-S advertisements

The typical OWL-S annotation paradigm using advertisements, that is, using the Service Profile ontology we mentioned in Section 2.1, is performed by defining annotations for the input/output parameters and the non-functional properties of Web services. These advertisements can be then mapped on UDDI constructs (Srinivasan et al., 2004), using a set of predefined tModels, such as input and output tModels that are used in a single UDDI category bag. As far as functional parameters are concerned, the approach assumes that *there is only a single annotation concept for each input/output parameter*, such as in the OWL-S advertisement of Fig. 4 that is mapped on the UDDI `businessService` construct of Fig. 5.

However, in the Web 2.0 domain, it would be more appropriate to allow the definition of more than one annotation concepts for a specific input or output parameter, creating *concept groups* that would allow many users to participate in the annotation activity of a Web service or, in our case, of a mashup. We refer to such kind of advertisements as *social-oriented OWL-S advertisements (SOOWL-S)*. For example, the output parameter of the profile instance in Fig. 4 could be annotated with one more concept, which might belong to a different ontology, as it is depicted in Fig. 6. Such a Profile definition is a consistent OWL-S instance since the input/output parameters of OWL-S are allowed to contain more than one `parameterType` definition. In that way, the annotated feed would have the {& `jour;CS_ELS_CFP`, & `jour2;ELS_CFP`} output concept group for a single output parameter.

This SOOWL-S advertisement, however, cannot be mapped by adding one more tModel entry in the mapping procedure of Srinivasan et al. (2004), since the service/mashup would then be semantically incoherent, considered to contain one more output, as it is depicted in Fig. 7.

4.1. Mapping SOOWL-S advertisements on UDDI

The `uddiMapping` procedure depicts the overall mapping algorithm of an SOOWL-S advertisement on a UDDI `businessService` construct. More specifically, for each advertisement, a UDDI `businessService` instance is created (line 1) along with a `categoryBag` instance (line 2).

Procedure `uddiMapping`(adv)

```
   Input: An SOOWL-S advertisement adv
   Result: The UDDI BusinessService mapping
1  bs ← BusinessService("serviceKey","businessKey")
2  cb ← CategoryBag()
3  taxonomyMapping(adv, cb)
4  nonFunctionalMapping(adv, cb)
5  bs.addCategoryBag(cb)
6  bts ← BindingTemplates()
7  bt ← BindingTemplate("serviceKey","businessKey")
8  bts.addBindingTemplate(bt)
9  tmid ← TModelInstanceDetails()
10 bt.addTModelInstanceDetails(tmid)
11 inputMapping(adv, tmid)
12 outputMapping(adv, tmid)
13 bs.addBindingTemplates(bts)
```

These instances are used by four sub-procedures relevant to the mapping of the *taxonomy* (line 3), *non-functional* (line 4), *input* (line 11) and *output* (line 12) annotations that we describe in the following sections. The mapping of the default Service Profile properties, such as service *name*, *description*, etc., is trivial and we omit it.

4.1.1. Taxonomy mapping

The advertisements are usually defined as direct instances of the `Profile` concept of the OWL-S Service Profile ontology. However, there are cases where we want to categorize an advertisement in terms of an existing profile taxonomy, which is actually a subclass hierarchy of the `Profile` concept (OWL-S 1.1 release, 2004). For example, the advertisement of Fig. 4 is defined as an instance of the class `RSS` that is assumed to be a subclass of the

```
<businessService ...>
  <categoryBag>
    <keyedReference KeyName="Output" KeyValue="&jour;CS_ELS_CFP"
      TModelKey="Output tModel uuid" />
  </categoryBag>
</businessService>
```

Fig. 5. The typical UDDI mapping of an output parameter.

```
<process:Output rdf:ID="_CallForPapersCompScEls">
  <process:parameterType rdf:datatype="&xsd;anyURI">&jour;CS_ELS_CFP
  </process:parameterType>
  <process:parameterType rdf:datatype="&xsd;anyURI">&jour2;ELS_CFP
  </process:parameterType>
</process:Output>
```

Fig. 6. Multiple annotations for a single output parameter.

```
<businessService ...>
  <categoryBag>
    <keyedReference KeyName="Output" KeyValue="&jour;CS_ELS_CFP"
      TModelKey="Output tModel uuid" />
    <keyedReference KeyName= "Output" KeyValue= "&jour2;ELS_CFP"
      TModelKey= "Output tModel uuid" />
  </categoryBag>
</businessService>
```

Fig. 7. The typical UDDI mapping of multiple output annotations.

`Profile` class, that is, `RSS` $\sqsubseteq$ `Profile`. Note that an SOOWL-S advertisement is actually an OWL instance and therefore, it might be defined in more than one taxonomy concept, a common ontology modeling technique that is known as *multiple instance class membership*. In that way we are able to classify advertisements in terms of their domain, incorporating different user perspectives in the Web 2.0 domain.

```
Procedure taxonomyMapping(adv, cb)
    Input: An SOOWL-S advertisement adv and a category bag cb
    Result: Adds the taxonomy mapping of adv to the given category bag
1   foreach ($t \in directType(adv)$) do
2       $kr \leftarrow KeyedReference()$
3       $kr.tModelKey$= TUUID
4       $kr.keyName$= “Taxonomy concept”
5       $kr.heyValue = t$
6       $cb.add(kr)$
7   end
```

By allowing many users to participate in the annotation process of an SOOWL-S advertisement, it is possible to result in more than one taxonomy concept for the same advertisement. The `mappingTaxonomy` procedure adds an appropriate reference (line 2) for each taxonomy concept to the category bag that is provided as parameter (line 6), using the predefined tModel key “TUUID”. Note that we map only the direct instance class membership relationships of an advertisement (line 1). The complete instance class membership relationships are retrieved from the subsumption hierarchy that is computed with reasoning (e.g. (Sirin et al., 2007)) over the advertisement ontology. More specifically, the *directType* method is defined as

$$directType(i) = \{C | i \in CEXT(C) \wedge \nexists D | D \sqsubseteq C : i \in CEXT(D)\}, \qquad (1)$$

where i is an ontology instance (advertisement) and $CEXT(C)$ denotes the class extension of the concept C, that is, the set of all of its instances.

4.1.2. Non-functional property mapping

The non-functional properties are accessible by advertisements through profile taxonomies and they can be used to annotate an advertisement beyond its functional parameters. For example, the `profile_1` instance of Fig. 4 is defined in the `RSS` taxonomy class and therefore, it inherits all its properties, such as the `url` property which denotes the URL of the Web feed. For each non-functional property of an SOOWL-S advertisement, the `nonFunctionalMapping` procedure adds an appropriate reference (line 2) to the category bag (line 6) that is provided as parameter, using the predefined tModel key “NFPUUID”.

```
Procedure nonFunctionalMapping(adv, cb)
    Input: An SOOWL-S advertisement adv and a category bag cb
    Result: Adds the non-functional property mappings of adv to the given category bag
1   foreach ($nf \in properties(adv)$) do
2       $kr \leftarrow KeyedReference()$
3       $kr.tModelKey$= NFPUUID
4       $kr.keyName = nf.name()$
5       $kr.heyValue = nf.value()$
6       $cb.add(kr)$
7   end
```

4.1.3. Input/output parameter mapping

In the case of functional parameters, there is the need to group together the concepts that annotate a particular input or output parameter. Otherwise, it is impossible to identify the concepts that annotate the same parameter, as we have explained for the example in Fig. 7.

The mapping of the functional parameters of an SOOWL-S advertisement on UDDI is depicted in the `inputMapping` and `outputMapping` procedures. For each `process:Input` and `process:Output` definition we create dynamically a complex tModel that consists of one or more input (IUUID) and output (OUUID) tModel entries, respectively, that are attached to the business service instance using binding templates. In that way, the matchmaking algorithm that we describe in Section 5.2 is able to perform based on more than one annotation concepts per each functional parameter.

Procedure `inputMapping`(*adv*, *tmid*)

```
Input: An SOOWL-S advertisement adv and a category bag cb
Result: Adds the input mappings of the adv to the given TModelInstanceDetail instance
foreach ($pi \in process{:}Input(adv)$) do
    $tm \leftarrow TModel($"uuid:xxx..."$)$
    $cb \leftarrow CategoryBag()$
    foreach ($papamType \in pi$) do
        $kr \leftarrow KeyedReference()$
        $kr.tModelKey$= IUUID
        $kr.keyName$= "Input"
        $kr.KeyValue = paramType.value()$
        $cb.add(kr)$
    $tm.addCategoryBag(cb)$
    $tmii \leftarrow TModelInstanceInfo()$
    $tmii.tModelKey = tm.key()$
    $tmid.addTModelInstanceInfo(tmii)$
```

Fig. 8 presents the final `businessService` UDDI mapping of the SOOWL-S RSS advertisement of Fig. 4 that has been extended with more than one output annotation, as it is depicted in Fig. 6. The taxonomy concepts and the non-functional properties are mapped as references on a single category bag and they are identified by their tmodel keys. However, the multiple annotated output parameter of the example is mapped on a dynamically generated tModel with key "`uuid:ccc...`" that is referenced by a `tModelInstanceInfo` construct. In that way, we are able to annotate each service input and output parameter with many ontological concepts that are grouped in a single tModel in order to be specially treated by the mashup matchmaking algorithm.

5. MashUDDI: Mashup-oriented UDDI repositories

The MashUDDI registries play a key role in the framework of Fig. 3. They are actually UDDI registries that have been augmented with two capabilities:

Procedure `outputMapping` (*adv*, *tmid*)

```
Input: An SOOWL-S advertisement adv and a category bag cb
Result: Adds the output mappings of the adv to the given TModelInstanceDetail instance
foreach ($po \in process{:}Output(adv)$) do
    $tm \leftarrow TModel($"uuid:yyy..."$)$
    $cb \leftarrow CategoryBag()$
    foreach($papamType \in po$) do
        $kr \leftarrow KeyedReference()$
        $kr.tModelKey$ = OUUID
        $kr.keyName$= "Output"
        $kr.KeyValue = paramType.value()$
        $cb.add(kr)$
    $tm.addCategoryBag(cb)$
    $tmii \leftarrow TModelInstanceInfo()$
    $tmii.tModelKey = tm.key()$
    $tmid.addTModelInstanceInfo(tmii)$
```

1. They can store SOOWL-S advertisements by applying the SOOWL-S mapping algorithm we have described in Section 4.1
2. They are able to semantically search for relevant SOOWL-S advertisements to a specific query, applying a mashup matchmaking algorithm.

A MashUDDI is actually a semantically extended UDDI repository that provides the necessary infrastructure for storing (publishing) and retrieving (discovering) SOOWL-S advertisements. The

```
<businessService serviceKey="serviceKey" businessKey="bussinesKey">
  <categoryBag>
    <keyedReference tModelKey="TUUID" keyName="Taxonomy Concept"
      keyValue="jour:RSS" />
    <keyedReference tModelKey="NFPUUID" keyName="rss:url"
      keyValue="http://../CallForPapersComputerScienceElsevier" />
  </categoryBag>
  <bindingTemplates>
    <bindingTemplate bindingKey="bindingKey" serviceKey="serviceKey">
      <tModelInstanceDetails>
        <tModelInstanceInfo tModelKey="uuid:ccc..."/>
      </tModelInstanceDetails>
    </bindingTemplate>
  </bindingTemplates>
</businessService>
<tModel tModelKey="uuid:ccc..." >
  <categoryBag>
    <keyedReference tModelKey="OUUID" keyName="Output"
      keyValue="&jour;CS_ELS_CFP" />
    <keyedReference
      tModelKey="OUUID" keyName="Output" keyValue="&jour2;ELS_CFP" />
  </categoryBag>
</tModel>
```

Fig. 8. The UDDI mapping of the SOOWL-S RSS advertisement.

discovery can be performed based on keywords, exploiting the native UDDI interface, and on user requirements expressed in terms of SOOWL-S queries. Furthermore, as the UDDI specification imposes, a UDDI registry exposes its interface via SOAP messages, i.e. it is a Web service itself. Therefore, a MashUDDI can be viewed *as a Web service that can be consumed by a client*, which, in our case, could be a mashup discovery or publish meta-service module (Fig. 3).

5.1. Publishing SOOWL-S in mashUDDI repositories

The publishing procedure of mashup advertisements involves the submission of an SOOWL-S advertisement to a MashUDDI registry or modifying an existing advertisement. The former takes place when the user submits the advertisement of a newly created mashup. The latter is relevant to the social-oriented environment of mashups and involves the participation of many users in the annotation process of mashups. In both cases, an SOOWL-S advertisement needs to be submitted to the MashUDDI registry in order to be processed, transformed and stored in the form of UDDI constructs. We do not elaborate further on the publishing procedure for the following two reasons:

- Since an SOOWL-S advertisement is actually an OWL-S ontology, it can be easily created using existing APIs and tools, such as the OWL-S API.[9]
- The mapping of an SOOWL-S advertisement on UDDI constructs has been discussed thoroughly in Section 4.1.

In the following section we present in detail the mashup matchmaking algorithm that a MashUDDI employs for the semantic discovery of mashup advertisements.

5.2. The mashup matchmaking algorithm

In the typical Profile-based OWL-S Web service matchmaking scenario, both queries and services are represented as OWL-S advertisements and the goal is to find the set of the relevant advertisements to the query. Following the same rationale, the goal of the mashup matchmaking algorithm is to retrieve the set of SOOWL-S mashup advertisements relevant to a query SOOWL-S advertisement.

The mashup matchmaking algorithm, which is depicted in the `mashupMatchmaking` procedure, is defined over the taxonomy, input, output and non-functional annotations. The algorithm makes use of two semantic operators, namely the *semantic intersection of two concept sets* ($\tilde{\cap}$) and the *semantic intersection of two instance sets* ($\overset{\circ}{\cap}$) that are defined as follows:

$$A\tilde{\cap}B = \min_{\forall x \in A, \forall y \in B}[d(x,y)], \tag{2}$$

$$A\overset{\circ}{\cap}B = \begin{cases} \text{true} & \text{if } \exists x \in A, \exists y \in B : x = y \vee owl:sameAs(x,y), \\ \mathit{false} & \text{otherwise.} \end{cases} \tag{3}$$

More specifically, the semantic concept intersection returns the minimum concept distance $d(x,y)$ found among the concepts of two sets A and B. Any concept distance measure can be used, provided that

- it is able to treat differently plugin and subsume matches (Paolucci, Kawamura, Payne, & Sycara, 2002), and
- $d(x,y) \in [0..1]$, with *1* denoting absolute mismatch.

The instance intersection between two instance sets A and B returns true if there are at least two instances $x \in A$ and $y \in B$, such that x and y are identical (=), in terms of identical URIs, or the same (*owl:sameAs*). In the following we describe in detail the mashup matchmaking algorithm that returns the similarity of a query (*query*) and a mashup (*adv*) SOOWL-S advertisement as the weighted mean value of their taxonomy and functional similarities (line 26). The similarity is computed based on the operators (2) and (3), on a concept similarity threshold a, and on three weights that denote the user's preferences regarding the importance of the taxonomy (w_t) and functional similarities (w_i and w_o) in the weighted mean sum.

Procedure `mashupMatchmaking`$(query, adv, a, w_t, w_i, w_o)$

```
   Input: A query and an adv SOOWL-S descriptions, a
   concept similarity threshold $a \in (0..1]$ and three weights
   $w_t, w_i, w_o \in [0..1]$
   Result: The similarity of the two descriptions normalized
   in [0..1]
1  $taxSim \leftarrow 1 - directType(query)\tilde{\cap}directType(adv)$
2  if ($taxSim < a$) then return 0
3  foreach ($in_{adv} \in adv.inputs()$) do
4      $ICG_{adv} \leftarrow in_{adv}.conceptGroup()$
5      $found \leftarrow false$
6      foreach ($in_{query} \in query.inputs()$) do
7          $ICG_{query} \leftarrow in_{query}.conceptGroup()$
8          $inputSim \leftarrow 1 - ICG_{adv}\tilde{\cap}ICG_{query}$
9          if ($inputSim \geqslant a$) then
10             $found \leftarrow true$
11             break
12     if (not $found$) then return 0
13 foreach $out_{query} \in query.outputs()$ do
14     $OCG_{query} \leftarrow out_{query}.conceptGroup()$
15     $found \leftarrow false$
16     foreach ($out_{adv} \in adv.outputs()$) do
17         $OCG_{adv} \leftarrow out_{adv}.conceptGroup()$
18         $outputSim \leftarrow 1 - OCG_{query}\tilde{\cap}OCG_{adv}$
19         if ($outputDist \geqslant a$) then
20             $found \leftarrow true$
21             break
22     if (not $found$) then return 0
23 $commonNonFunc \leftarrow query.nonFunc() \cap adv.nonFunc()$
24 foreach ($nf \in commonNonFunc$) do
25     if (not $query.nf \overset{\circ}{\cap} adv.nf$) then return 0
26 return $\frac{w_t \cdot taxSim + w_i \cdot inputSim + w_o \cdot outputSim}{w_t + w_i + w_o}$
```

5.2.1. Taxonomy matching

The taxonomy matching is defined over the sets with the direct types of the two mashup descriptions (see Eq. (1)). For example, the profile instance of Fig. 4 has $directType$(`profile_1`) = {`RSS`}, even if `profile_1` $\in CEXT$(`Profile`), since `RSS` $\sqsubseteq$ `Profile` and therefore, `RSS` is the direct type of `profile_1`.

The taxonomy algorithm stores in the *taxSim* variable the similarity, that is, (1 − *distance*), computed by the concept intersection operator over the direct type sets of the two advertisements (line 1). If the computed similarity is lower than the provided thershold a (line 2), then the procedure terminates returning 0 that denotes an absolute mismatch. Note that the `mashupMatchmaking` procedure returns the similarity of two SOOWL-S advertisements and not their distance.

5.2.2. Functional matching

The functional matching algorithm (lines 3 to 22) follows the idea that each mashup input concept group should semantically

9 http://projects.semwebcentral.org/projects/owl-s-api/.

intersect at least one query input concept group (lines 3 to 12). Similarly, each query output concept group should semantically intersect at least one mashup output concept group (lines 13 to 22). In that way, the mashup matchmaking algorithm ensures that (a) all the input concept groups of a matched SOOWL-S mashup description (ICG_{adv}) are satisfied by at least one query input concept group (ICG_{query}), and (b) all the query output concept groups (OCG_{query}) are satisfied by at least one output concept group (OCG_{adv}) of a matched SOOWL-S mashup description. Note that the concept similarity threshold is also considered during the functional matching (lines 9 and 19).

5.2.3. Non-functional matching

Finally, the algorithm checks the value sets of the common non-functional properties of the two SOOWL-S advertisements (lines 23 to 25). More specifically, for each non-functional query property value set, it checks the instance intersection with the corresponding property value set of the mashup description. Therefore, a mashup advertisement matches a query if the former satisfies, in terms of the instance intersection operator, all the non-functional query requirements. Note that:

- The non-functional matching algorithm ignores the values that do not match between two advertisements.
- The instance intersection operator can be applied also in datatype property value sets, performing exact match among the datatype values, e.g. strings.

5.2.4. A simple example

In order to demonstrate the rationale behind the SOOWL-S matchmaking algorithm, we present a simple example related to the functional parameters of a mashup that returns the author (output) of a book (input). Suppose that this mashup has been created by combining an RSS feed of an online bookstore and a search function over the RSS text. The annotation process is not necessary to comply with the implementation aspects of the mashup. Therefore, the mashup can be considered as a single service and the annotation activity of more than one users could result in the input concept group {`uri1:Title`, `uri2:NovelTitle`,...} and the output concept group {`uri1:Author`, `uri2:Writer`,...}. In that way, a query defined using the {`uri2:BookTitle`,...} input concept group and the {`uri1:Author`,...} output concept group would match the service, even if the `Title`, `NovelTitle` and the `BookTitle` concepts (as well as the `Author` and `Writer` concepts) do not lexicographically match. Assuming that (a) $\texttt{uri2:NovelTitle} \sqsubseteq \texttt{uri2:BookTitle}$, (b) the query and the mashup have the same taxonomy concepts, and (c) we allow all the hierarchically related concepts to be matched ($a \simeq 0$), we have from the `mashupMatchmaking` procedure (lines 8 and 18) that

$$\{\texttt{uri2:BookTitle},\ldots\} \,\widetilde{\cap}\, \{\texttt{uri1:Title},\texttt{uri2:NovelTitle},\ldots\} \neq 0, \quad \text{and} \quad \{\texttt{uri1:Author},\texttt{uri2:Writer},\ldots\} \,\widetilde{\cap}\, \{\texttt{uri1:Author},\ldots\} \neq 0$$

for the inputs and outputs, respectively. In that way, we are able to match the two mashup descriptions, capturing different users' perspectives during the annotation activity.

6. A prototype mashUDDI repository in Yahoo Pipes

We have implemented the framework of Fig. 3 in Yahoo Pipes, defining a discovery meta-service module that communicates with a MashUDDI repository. We used the JUDDI[10] and UDDI4J[11] libraries in order to implement the MashUDDI registry and the Pellet DL reasoner (Sirin et al., 2007) as the MashUDDI's reasoning infrastructure. Since there is not any real-world SOOWL-S dataset to consider in our implementation, we stored a modified version of the OWLS-TC version 2.2 advertisement collection.[12] This collection follows the conventional SWS OWL-S profile model, using only single concept annotations for each I/O parameter. For that reason, we preprocessed the dataset, adding in each I/O parameter an arbitrary number of annotation concepts up to 20, in the same rationale to Fig. 6. The concepts were selected randomly from the set of the domain ontologies of the collection. In that way, we resulted in a synthetic dataset of 1007 SOOWL-S profiles. The same procedure we followed also for the 29 OWL-S query profiles of the collection.

6.1. Abstract mashup advertisements

The modified OWL-S advertisements of the OWLS-TC dataset are compliant with the abstract nature of mashup advertisements. More specifically, we assume that each mashup is semantically described by an SOOWL-S advertisement that contains concept group annotations, i.e. ontological concepts, about service inputs and outputs. For example, the input parameter of a service that returns the price of books can be annotated with an ontology concept `Book` and the output can be annotated with an ontology concept `Price`. These advertisements are stored in our MashUDDI, following the algorithm of we have presented in Section 4.1. Similarly, a mashup request (query) is formed by defining ontological concepts for the inputs and outputs of the desirable mashup. In that way, the mashup discovery procedure applies the matchmaking algorithm of Section 5.2 on mashups' and queries' input and output annotations in order to determine "similar" advertisements.

6.2. Realizing the discovery meta-service module

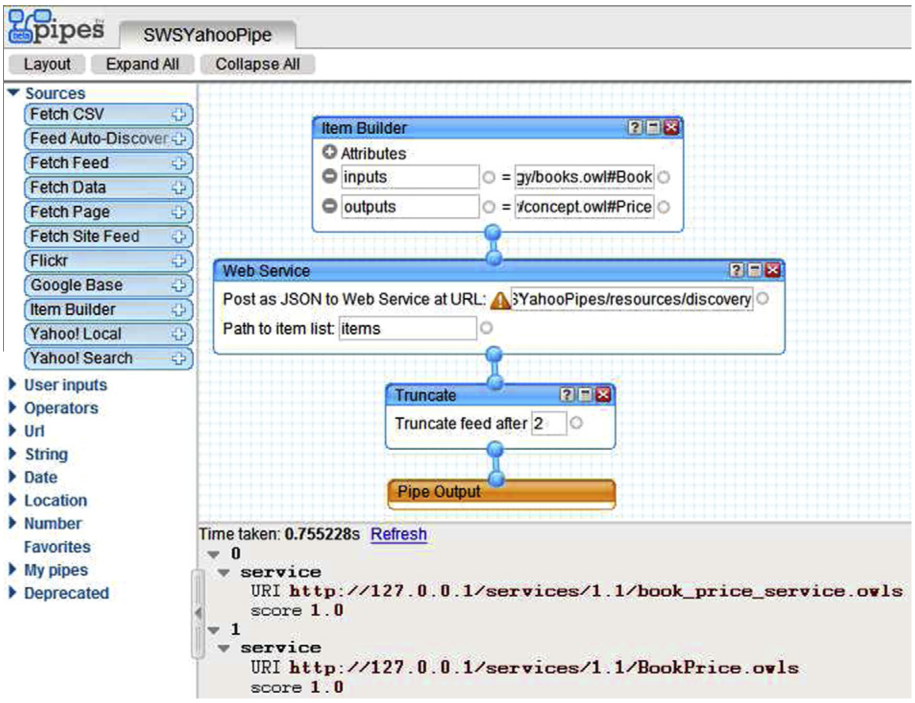
We used the JSON-based meta-service module of Yahoo Pipes in order to communicate with our MashUDDI. Fig. 9 depicts the workflow of the overall pipe. The *Item Builder* module is used in order to create a single item data source by assigning values to the inputs and outputs attributes. The values are the URIs of the ontological concepts that we use to annotate the input and output parameters of the desirable service. The *Web Service* module is the provided meta-service module of Yahoo Pipes. It takes as parameter the URL of our RESTful JSON-enabled Web service and the path to the item list. Finally, we used the *Truncate* module in order to obtain only the first two matched services.

In this simple example, we searched for services in our MashUDDI that consume information annotated semantically with the `Book` concept and produce information annotated semantically with the `Price` concept. The pipe returned two results with the URI and the matching score of each service. In that way, a user can select a returned URI and use it as input to a normal meta-service module in order to invoke the service.

The absolute match (1.0) denotes that the returned services have semantically the same input/output annotations to the required ones: the annotation concepts have either the same concept URIs or they are equivalent in terms of OWL concept equivalence (Eq. (2)). As a concept distance measure d we have implemented the simple *edge-based distance* that computes the distance of two concepts based on the number of edges found on the shortest path between them in the ontology, and we set $a = 1$ in order to retrieve only exact matched mashups.

Bear in mind that the absolute match does not necessarily mean that the returned services satisfy the desirable functionality: two

[10] http://ws.apache.org/juddi/.
[11] http://uddi4j.sourceforge.net/
[12] http://projects.semwebcentral.org/projects/owls-tc/.



inputs = gy/books.owl#Book
outputs = /concept.owl#Price

Post as JSON to Web Service at URL: SYahooPipes/resources/discovery
Path to item list: items

Truncate feed after 2

```
Time taken: 0.755228s Refresh
0
  service
    URI http://127.0.0.1/services/1.1/book_price_service.owls
    score 1.0
1
  service
    URI http://127.0.0.1/services/1.1/BookPrice.owls
    score 1.0
```

Fig. 9. A Yahoo Pipe workflow based on a discovery meta-service module.

services may have the same input/output annotations (*signature* match) but they may have a totally different functionality (*specification* match) (Zaremski & Wing, 1995). However, our intention is to offer the functionality to users to search semantically for mashups in mashup tools, making a lightweight use of SWS frameworks. In native SWS frameworks, automation is at first place and a more sophisticated semantic service representation is needed, e.g. *preconditions*, *effects* or even further *negotiation* among the involved parties in order to determine the actual service functionality. However, such sophisticated approaches would affect negatively the usability of mashup tools for the average users.

Similarly to the discovery meta-service module, a publish meta-service module can be defined in order to create annotations about mashups and to store them in the MashUDDI. For example, the pipe of Fig. 9 is actually a mashup itself that can also be semantically annotated. Based on an ontology relevant to matchmaking algorithms, we could use two concepts, e.g. `ServiceInput` and `ServiceOutput`, to annotate the pipe input, and one concept, e.g. `MatchedService`, to annotate the pipe output. This is an example of the advantage of a conceptual approach, since we are able to create abstract annotations without following the implementation aspects of the created mashups.

We want to mention here that a keyword-based search for mashups is still possible in a MashUDDI, using the native UDDI API. For example, we could define a query using the "book" and "price" keywords. However, we argue that with the semantic approach we are able to define the service requirements more accurately, describing the desirable services at a conceptual input/output level and allowing the incorporation of semantics, defining, for example, that we are interested in "books" and not in some general "booking" service.

6.3. Experimental results

Fig. 10 depicts the query response time for each one of the 29 SOOWL-S query profiles on a desktop PC with 3.2 GHz processor, 1 GB RAM and JAVA EE 5 SDK. As we expected, the query response time depends on the number of I/O parameters (`process:Input`, `process:Output`) and the size of the I/O concept groups of the query and mashup advertisements, that is, the number of the `process:parameterType` definitions. For example, query 15 defines two concept groups (it has one input and one output parameter), each containing a single concept. On the other hand, query 11 defines four concept groups (it has three input and one output parameter), each containing 3 or 4 concepts. For that reason, the response time of query 11 is greater than of query 15. However, the time overhead that is introduced by the extra number of annotation concepts does not affect heavily the performance.

Note that due to the modifications we have made in the initial OWLS-TC profile collection, introducing additional concepts, the query relevance sets that are provided by the collection do not fit in our experiments. Bear in mind that due to the lack of any SOOWL-S collection suitable for our mashup matchmaking algorithm and UDDI mapping procedure, we generated a synthetic one which is far away from a real-world collection. Therefore, the collection can only be used for an algorithmic evaluation of our framework.

7. Related work

Web 2.0 mashups have already attracted the interest of Semantic Web researchers. SA-REST (Sheth et al., 2007) is an effort to develop a standard for adding semantic annotations into Web pages where RESTful services are described. It is a developer-oriented, low-level approach, dealing mostly with data mediation issues, whereas we follow a user-oriented direction, supporting mashup tool users with semantic capabilities, based on existing standards, independently of implementation issues.

SWAF (Oren et al., 2007) extends the Ruby on Rails framework for dynamic Web application development using the Ruby dynamic language. It is based only on RDF metadata and uses the ActiveRDF library in order to create Ruby classes, objects and attributes. Like SA-REST, SWAF focuses mainly on experienced developers, helping them to build their Web applications, whereas we

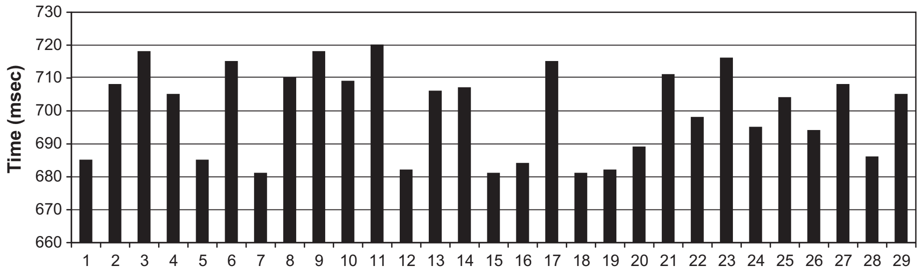


Fig. 10. Query response times for the 29 SOOWL-S queries.

target at the average mashup tool user and we provide a simple framework for the semantic discovery of mashups in mashup tools.

SBWS (Battle & Benson, 2008) is a tool for integrating Web services and Semantic Web. The goal is to provide a SPARQL endpoint to Web services based on the WSDL or WADL services' documents, using, however, only RDF metadata. In the case of SOAP services, it follows the paradigm of the SWS matchmaking algorithms, incorporating single concept annotations. In the case of RESTful services, a semantically enhanced WADL document is needed, as a substitute of WSDL. Our work follows a single, conceptual approach for any Web service API based on SOOWL-S advertisements and on the UDDI standard.

Semantic Web Pipes (Morbidoni, Phuoc, Polleres, Samwald, & Tummarello, 2008) is a tool for building RDF-based mashups. It works by fetching RDF models on the Web, operating on them, and producing an output which is itself accessible via a stable URL. Therefore, it is a mashup tool built from scratch able to process RDF annotations. With our framework we want to reuse the existing infrastructure of mashup tools and it mainly targets at the mashup discovery domain.

Apart from service-related approaches, there are general approaches for embedding semantics into Web pages. Microformats (Khare, 2006) and RDFa[13] add lightweight semantic markup into HTML tags. GRDDL[14] is a technique for declaring and applying transformations to XML in order to extract RDF. hGRDDL (Adida, 2008) is a GRDDL-oriented method, preserving the correlation between the semantic annotations and the structure of the HTML Web page. Actually, (Sheth et al., 2007) is based on RDFa and GRDDL in order to annotate the Web pages of Web services.

A great advantage of our framework is that, since it is based on the OWL-S ontology, already existing OWL-S matchmaking algorithms, such as Klusch, Fries, and Sycara (2006), Kourtesis and Paraskakis (2008), Meditskos and Bassiliades (2007), Srinivasan et al. (2004), can be also used for mashup discovery, provided that they implement the modifications described in Sections 4.1 and 5.2 for the UDDI mapping and matchmaking procedures for SOOWL-S. In that way, we can take full advantage of already tested algorithms.

To the best of our knowledge, we consider our framework as the first approach of combining existing SWS and Web service standards, such as the OWL-S ontology and the UDDI, with the existing infrastructure of mashup tools. This attempt has led to the introduction of the SOOWL-S mashup advertisement that allows the conceptual and collaborative mashup annotation, regardless of the implementation API that is followed.

8. Conclusions

In this paper we presented a combination of existing frameworks for introducing semantic mashup discovery capabilities in mashup tools. Our approach is based on the definition of SOOWL-S advertisements for the conceptual description of mashups, using special mapping and matchmaking algorithms for UDDI registries, i.e. MashUDDI repositories. Our framework is defined upon the existing infrastructure of mashup tools without needing to alter any implemented technology.

We exemplified on our framework using SOOWL-S advertisements for discovering mashups in the Yahoo Pipes mashup tool. We followed a rather conceptual/abstract direction for service annotation, using only the intuitive OWL-S functional parameters that characterize a Web service, i.e. inputs/outputs, targeting at the average mashup tool user. In fact, any framework for semantic service annotation can be used that follows a conceptual approach, such as the *WSMO Discovery Framework* (Keller, Lara, Lausen, & Fensel, 2006; Li, Du, & Tian, 2007).

A limitation of the ontology-based approach is that the user must be familiar with the domain ontologies that are used by a specific mashup tool in order to formulate semantic service requests. For example, we were familiar with the domain ontologies of the OWLS-TC service collection and we used directly the `Book` and `Price` concepts to semantically search for book-price mashups in Fig. 9. Otherwise, we should have firstly identified the domain ontology concepts that fit to our requirements. To overcome such a limitation, the framework can be extended with the ability of mapping text to ontology concepts (Bhagdev, Chapman, Ciravegna, Lanfranchi, & Petrelli, 2008) in order to enable the use of simple keywords to formulate semantic requests. In that way, our framework can enhance the traditional keyword-based searching of mashup tools with semantic results.

In the future we want to evaluate our framework on different SWS standards in order to investigate the factors that might affect the usability of a semantically-enabled mashup tool for the average mashup tool user. Furthermore, we plan to make extensive experiments with different SWS discovery algorithms in order to determine the actual advantage of a semantically-enabled mashup tool in terms of recall and precision, compared to the traditional keyword-based search. Such an evaluation is not possible with the current version of OWLS-TC, since it lacks sufficient textual descriptions.

References

Adida, B. (2008). hGRDDL: Bridging microformats and RDFa. *Web Semant.*, *6*(1), 54–60.

Akkiraju, R., Farrell, J., Miller, J., Nagarajan, M., Schmidt, M.T., Sheth, A., Verma, K. (2005). Web service semantics-WSDL-S. Available online http://www.w3.org/Submission/WSDL-S/.

Baader, F. (2003). *The description logic handbook: Theory*. *Implementation and applications*. Cambridge University Press.

Battle, R., & Benson, E. (2008). Bridging the semantic Web and Web 2.0 with representational state transfer (REST). *Web Semant.*, *6*(1), 61–69.

Bhagdev, R., Chapman, S., Ciravegna, F., Lanfranchi, V., Petrelli, D. (2008). Hybrid search: Effectively combining keywords and ontology-based searches. In: Hauswirth, M., Koubarakis, M., Bechhofer, S. (Eds.), Proceedings of the 5th European semantic web conference, LNCS. Berlin, Heidelberg.

[13] http://www.w3.org/TR/xhtml-rdfa-primer/.

[14] http://www.w3.org/TR/grddl/.

Burstein, M., Bussler, C., Finin, T., Huhns, M., Paolucci, M., Sheth, A., et al. (2005). A semantic Web services architecture. *IEEE Internet Comput., 9*(5), 72–81.

Dogac, A., Kabak, Y., Laleci, G. B., Mattocks, C., Najmi, F., & Pollock, J. (2005). Enhancing ebXML registries to make them OWL aware. *Distributed and Parallel Databases, 18*(1), 9–36.

Dustdar, S., & Treiber, M. (2005). A view based analysis on Web service registries. *Distributed and Parallel Databases, 18*(2), 147–171.

Fensel, D., Lausen, H., Polleres, A., Bruijn, J. D., Stollberg, M., & Roman, D., et al. (Eds.). . *Enabling semantic web services: The web service modeling ontology*. Heidelberg: Springer-Verlag.

Fielding, R. T., & Taylor, R. N. (2000). Principled design of the modern Web architecture. In *ICSE '00: Proceedings of the 22nd international conference on software engineering* (pp. 07–416). New York, NY, USA: ACM.

Herzog, R., Zugmann, P., Stollberg, M., Roman, D.: D10 v0.1 WSMO registry (2007). Available online http://http://www.wsmo.org/2004/d10/v0.1/.

Keller, U., Lara, R., Lausen, H., & Fensel, D. (2006). *Semantic web service discovery in the WSMO framework*. Idea Publishing Group.

Khare, R. (2006). Microformats: The next (small) thing on the semantic Web? *IEEE Internet Comput., 10*(1), 68–75.

Klusch, M., Fries, B., & Sycara, K. (2006). Automated semantic Web service discovery with OWLS-MX. In *AAMAS '06: Proceedings of the fifth international joint conference on autonomous agents and multiagent systems* (pp. 915–922). New York, NY, USA: ACM.

Kopecký, J., Vitvar, T., Bournez, C., & Farrell, J. (2007). SAWSDL: Semantic annotations for WSDL and XML schema. *IEEE Internet Comput., 11*(6), 60–67.

Kourtesis, D., & Paraskakis, I. (2008). Combining SAWSDL, OWL-DL and UDDI for semantically enhanced Web service discovery. In S. Bechhofer, M. Hauswirth, J. Hoffmann, & M. Koubarakis (Eds.), *5th European semantic Web conference (ESWC), lecture notes in computer science* (pp. 614–628). Springer. Vol. 5021.

Li, H., Du, X., Tian, X. (2007). A WSMO-based semantic Web services discovery framework in heterogeneous ontologies environment. In KSEM, pp. 617–622.

Martin, D., Burstein, M., Mcdermott, D., Mcilraith, S., Paolucci, M., Sycara, K., et al. (2007). Bringing semantics to Web services with OWL-S. *World Wide Web, 10*(3), 243–277.

Meditskos, G., & Bassiliades, N. (2007). Object-oriented similarity measures for semantic Web service matchmaking. In *ECOWS '07: Proceedings of the fifth European conference on Web services* (pp. 57–66). Washington, DC, USA: IEEE Computer Society.

Medjahed, B., & Bouguettaya, A. (2005). A dynamic foundational architecture for semantic Web services. *Distributed and Parallel Databases, 17*(2), 179–206.

Morbidoni, C., Phuoc, D. L., Polleres, A., Samwald, M., & Tummarello, G. (2008). Previewing semantic Web pipes. In S. Bechhofer, M. Hauswirth, J. Hoffmann, & M. Koubarakis (Eds.), *ESWC, Lecture notes in computer science* (pp. 843–848). Springer. Vol. 5021.

Oren, E., Haller, A., Mesnage, C., Hauswirth, M., Heitmann, B., & Decker, S. (2007). A flexible integration framework for semantic Web 2.0 applications. *IEEE Softw., 24*(5), 64–71.

O'Sullivan, J., Edmond, D., & ter Hofstede, A. H. M. (2002). What's in a service? towards accurate description of non-functional services properties. *Distributed and Parallel Databases, 12*(2/3), 117–133.

OWL-S 1.1 release: Examples (2004). Available online http://http://www.daml.org/services/owl-s/1.1/examples.html.

Paolucci, M., Kawamura, T., Payne, T. R., & Sycara, K. P. (2002). Semantic matching of Web services capabilities. In *ISWC '02: Proceedings of the First international semantic Web conference on the semantic Web* (pp. 333–347). London, UK: Springer-Verlag.

Preist, C. (2004). A conceptual architecture for semantic Web services. In International Semantic Web Conference (ISWC), pp. 395–409.

Sheth, A., Gomadam, K., & Lathem, J. (2007). SA-REST: Semantically interoperable and easier-to-use services and mashups. *IEEE Internet Comput., 11*(6), 91–94.

Sirin, E., Parsia, B., Grau, B. C., Kalyanpur, A., & Katz, Y. (2007). Pellet: A practical OWL-DL reasoner. *Web Semant., 5*(2), 51–53.

Sivashanmugam, K., Verma, K., Sheth, A. P., & Miller, J. A. (2003). Adding semantics to Web services standards. In L. J. Zhang (Ed.), *ICWS* (pp. 395–401). CSREA Press.

Srinivasan, N., Paolucci, M., & Sycara, K. P. (2004). An efficient algorithm for OWL-S based semantic search in UDDI. In J. Cardoso & A. P. Sheth (Eds.), *SWSWPC, Lecture notes in computer science* (pp. 96–110). Springer. Vol. 3387.

Tsalgatidou, A., & Pilioura, T. (2002). An overview of standards and related technology in Web services. *Distributed and Parallel Databases, 12*(2–3), 135–162.

Tsarkov, D., & Horrocks, I. (2006). FaCT++ description logic reasoner: System description. In *Proceedings of the international joint conference on automated reasoning (IJCAR 2006), Lecture Notes in Artificial Intelligence* (pp. 292–297). Springer. Vol. 4130.

UDDI.org: UDDI technical white paper.(2000). Technical Report.

Zaremski, A. M., & Wing, J. M. (1995). Specification matching of software components. In *SIGSOFT '95: Proceedings of the 3rd ACM SIGSOFT symposium on foundations of software engineering* (pp. 6–17). New York, NY, USA: ACM.

Zur Muehlen, M., Nickerson, J. V., & Swenson, K. D. (2005). Developing Web services choreography standards: The case of REST vs. SOAP. *Decision Support System, 40*(1), 9–29.