Chapter 1
Semantic annotations and Web service retrieval: the URBE approach

Pierluigi Plebani and Barbara Pernici

Abstract The goal of this chapter is to discuss how annotating the Web service interfaces can improve the precision of a Web service matchmaking algorithm. To this aim, we adopt URBE (UDDI Registry By Example) as a matchmaking algorithm for calculating the similarities between two Web service interfaces described using the SAWSDL or WSDL. The approach adopted in URBE takes into account both the structural and semantic analysis of the interfaces: the former takes into account the number of operations, inputs, and outputs as well as the data types involved; the latter considers the concepts related to the names given to the service, the operations, and the parameters. In case the Web services are described with WSDL, WordNet is used to find the relationships between names. In case of SAWSDL-based descriptions, the analysis is based on the ontologies referred by the annotations.

1.1 Introduction

In the area of Autonomic Computing [8] methods and tools are required for making the execution of business processes as reliable as possible. When considering service-based processes, the reliability of a process strongly depends on the reliability of the composing services. As a consequence, in case of a service failure, it is fundamental to figure out how to find an alternative solution, i.e., a similar service. According to this scenario, inside the PAWS (Processes with Adaptive Web Services) framework [2], we have developed URBE (Uddi Registry By Example) a UDDI compliant service registry that also performs content-based retrieval. Gen-

Pierluigi Plebani
Dip. Elettronica ed Informazione - Politecnico di Milano, Via Ponzio 34/5 - 20133 Milan, Italy, e-mail: `plebani@elet.polimi.it`

Barbara Pernici
Dip. Elettronica ed Informazione - Politecnico di Milano, Via Ponzio 34/5 - 20133 Milan, Italy e-mail: `pernici@elet.polimi.it`

erally speaking, URBE is a tool for supporting process design. For specifying a process definition or for substitution purposes, given a service interface description, with URBE a designer, following a *query by example* approach, can find the services published in a repository that expose the interfaces as similar as possible to the requested one.

The goal of this chapter is to give an overview of the matchmaker which URBE is based on. A detailed discussion of this matchmaker is given in [13]. In particular, since URBE can analyze both WSDL and SAWSDL Web service interface descriptions, in this chapter we focus on how the annotations defined in a SAWSDL file can be useful to improve the accuracy of the matchmaking algorithm in terms of precision and recall.

The chapter is structured as follows. Section 1.2 introduces the ideas underlying the URBE approach. Section 1.3 enters into the details of the algorithm that calculates the similarity between two Web service description interfaces. Section 1.4 analyzes the matchmaking results, especially considering the influences on the annotations included in SAWSDL documents. A summary of the content of this chapter is given in Section 1.5.

1.2 Approach: Web service substitution

As discussed in the introduction, Web service substitution might be required when the execution of a service-based business process fails due to a failure of one of the composing services. Regardless of when the selection of the substituting Web service occurs (i.e., at design or run-time), one of the main goals is to minimize the engineering effort for the Web service substitution required on the client-side to re-implement the Web service functionalities invocations. URBE aims at supporting this situation by providing a similarity function:

$$fSim(ws_a, ws_b) \rightarrow [0..1] \tag{1.1}$$

where ws_a and ws_b are two Web service interfaces described with WSDL (or SAWSDL). The higher the result of $fSim$, the higher the similarity between the two interfaces is. Since our goal is the Web service substitution, a higher value of $fSim$ also means less burden with the Web service substitution. So, we assume that:

Definition 1. Given two Web service interfaces ws_a and ws_b, then $fSim(ws_a, ws_b) = 1$ if the two Web services expose the same interface, whereas $fSim(ws_a, ws_b) = 0$ in case the interfaces are completely different.

Figure 1.1 shows the interfaces of two similar Web services. In this case, even if they fulfill the same goal, i.e., currency exchange, the number of available operations, as well as the way in which the input and output parameters are named, is different. Since one of our goals is to evaluate the similarity for substitutability, we need to consider that substituting `CurrencyWS` with `CurrencyExchangeService` is different to the opposite activity. Thus:

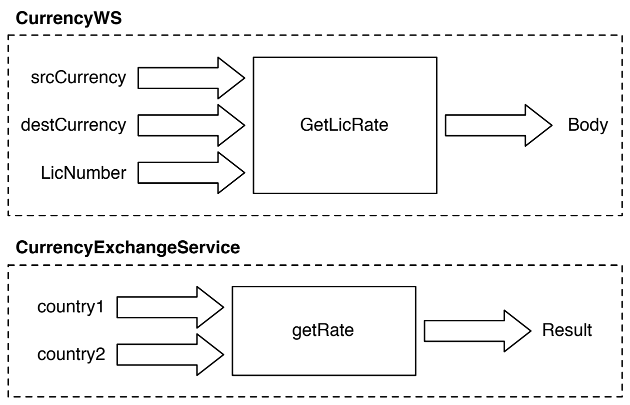


Fig. 1.1 Example of similar Web services.

Definition 2. The function $fSim$ is not reflective. So, $fSim(ws_a, ws_b)$ could differ from $fSim(ws_b, ws_a)$ as the similarity depends on which Web service holds the role of subtituting and which the role of substituted.

Indeed, if we usually invoke the `CurrencyWS`, in case of a failure we can start invoking the corresponding operation of the `CurrencyExchangeService` after implementing a mediator able to transform the messages: skipping the value of the license number, i.e. `LicNumber`, obtaining the country name from the related currency symbol, and modifying the name of the invoked operation and the parameters. On the contrary, if we are using `CurrencyExchangeService` and we have to switch to `CurrencyWS`, the new Web service needs an additional parameter that could be obtained by payment of registration fees, i.e., `LicNumber`.

Generally speaking, URBE relies on the assumption that two Web services could be defined as functionally equivalent, and thus $fSim$ returns 1, if their interfaces expose the same operations with the same inputs and outputs. Thus, the two interfaces must use the same names to define the operations and the parameters and, with respect to the latter, the same data types are used. In case URBE realizes that these constraints are only partially satisfied, then the value returned by $fSim$ decreases accordingly. This results in a pair-wise comparison of the elements composing the service description. This approach has been inspired by the literature on reusable software components [20] and by work done in the Web service community [7]. With respect to the existing approaches, our algorithm combines both semantic and syntactic aspects of the Web service that can be derived from a WSDL description. The semantic aspects derives from the analysis of the relation between the names used for the whole service, the operations and the parameters and the concepts stored in WordNet. In particular,, we are not interested on the meaning of the names, but on their distance. In case, the service description includes explicit references to domain ontologies, the distance of these annotations are considered. Instead, the syntactic aspects can state the compliance between the input and output structures and the adopted data types. The algorithm assumes that, as usually occurs,

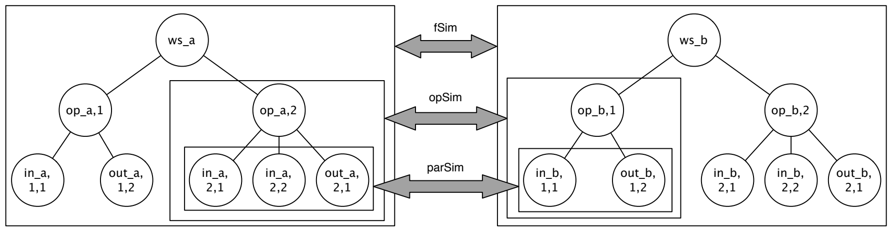


Fig. 1.2 Structure of the service similarity function *fSim* [13]

the WSDL specification of the Web service interface is (semi-)automatically generated by a tool starting from a software module, such as a Java class. URBE exploits this engineering practice during the semantic analysis since it implies that the resulting description will probably reflect the naming conventions usually adopted by developers [17]. In addition, when they exist, annotation of Web service description elements may improve the accuracy of the semantic analysis. Indeed, annotating the Web service elements with concepts in an ontology can help to better manage possible ambiguities in the meaning of terms in the Web service descriptions.

1.3 Solution: URBE matchmaker

The URBE matchmaker implements the algorithm that computes the similarity function *fSim*. Generally speaking, the hierarchical structure of a WSDL/SAWSDL description affects the structure of *fSim*. For this reason, *fSim* is defined in terms of *opSim*, which in turn, is defined in terms of *inParSim* and *outParSim* (see Figure 1.2). This approach results in the algorithm shown in Listing 1.1.

Regardless of the function considered, it is worth noting that at each level the algorithm is based on a function *maxSim* that implements the maximum weighted assignment in a bipartite graph problem [3]. Given a graph $G = (V,E)$, a *matching* is defined as $M \subseteq E$ so that no two edges in M share a common end vertex. If the edges of the graph have an associated weight, then a *maximum weighted assignment* is a matching such that the sum of the weights of the edges is maximum. Let us suppose that the set of vertices is partitioned in two sets Q and P, each edge of the graph connects a vertex from Q with a vertex from P, and that the edges of the graph have an associated weight given by a function $f : (Q,P) \rightarrow [0..1]$. The function $maxSim : (f,Q,P) \rightarrow [0..1]$ returns the maximum weighted assignment.

Applying the assignment in bipartite graphs problem to our context, the set Q represents a query, whereas P is what we compare with the query to evaluate the similarity. Let us assume, for instance, that Q and P are composed of the operations in ws_a and ws_b. $|Q| < |P|$ means that the number of operations in Q is lower than the number of operations available in P; so, for each operation in Q we may find a corresponding operation in P. On the contrary, $|Q| > |P|$ means that we are asking for more operations than are actually available. Since our approach aims to state if ws_a

```
function parSim(par_a, par_b) : double

  parSim = wParNameSim * nameSim(par_a.name, par_b.name);
  parSim = parSim + (1-wParNameSim) *
                    dataTypeSim(par_a.type, par_b.type);

end function

function opSim(op_a, op_b) : double

  opSim = wOpNameSim * nameSim(op_a.name, op_b.name);
  opSim = opSim + (1-wOpNameSim) *
         (0.5 * maxSim(parSim, op_a.inputs, op_b.inputs)  +
          0.5 * maxSim(parSim, op_b.outputs, op_a.outputs));

end function

function fSim(ws_a, ws_b) : double

  fSim = nameSim(ws_a.portType_name, ws_b.portType_name);
  fSim = fSim + maxSim(opSim, ws_a.operations, ws_b.operations)

end function
```

Listing 1.1 URBE Algorithm

can be replaced with ws_b, then the situation in which $|Q| \leq |P|$ is, in general, better than the case $|Q| > |P|$. For this reason, we divide the result of the maximization by the cardinality of $|Q|$. So, if $|Q| \leq |P|$ then $maxSim : (f, Q, P) \to [0..1]$, whereas if $|Q| > |P|$ then $maxSim : (f, Q, P) \to [0..\frac{|P|}{|Q|}]$. In this way, the function *maxSim* is asymmetric, i.e., $maxSim(f, Q, P) \neq maxSim(f, P, Q)$, and this justifies the property of *fSim* introduced the Definition 2.

The URBE matchmaker also includes a set of weights to assign the relevance of the similarity at the different levels during the computation of the overall similarity value. More specifically, the weight $wPTNameSim \in [0..1]$ defines how much the similarity of the names of the `portType`s has more importance than the similarity between the operations that these `portType`s contain in computing the overall similarity. In the same way, at operation level, the weight *wOpNameSim* weights the importance between the similarity of the operation names and the similarity of the related parameters in computing the operation similarity. Finally, *wParNameSim* weights the importance between the similarity of the parameter names and the similarity of the data types.

To conclude the URBE matchmaker analysis, the last two elements to be analyzed are the functions $dataTypeSim(dt_a, dt_b) \to [0..1]$ and $nameSim(n_a, n_b) \to [0..1]$. These functions return, respectively, the similarity among two data types, and the similarity among two names.

Focusing on the data type analyses, a data type in an SAWSDL/WSDL can be *built-in* or *complex* [5]. In the former case, simple data types (e.g., *xsd:string*, *xsd:decimal*, *xsd:dateTime*) as well as derived data types (e.g., *xsd:integer*, *xsd:short*, *xsd:byte*) are included. When comparing two built-in data types, following the approach defined in [16], their similarity is inversely proportional to the information loss that will occur if we apply a casting from dt_a to dt_b. For instance, if we move from an integer to a real, we do not have any information loss since all the integers can be represented with a real variable, i.e., $dataTypeSim(integer, real) = 1$. On the other way round, when casting a real to an integer, we lost the decimal part so we assume that $dataTypeSim(integer, real) = 0.5$. In the case of complex data types, data type is expressed according to an XSD schema which is included, or imported, in the WSDL specification as a *complexType*: a data type which includes other data types (either built-in or complex). To reduce the complexity of the overall algorithm, the URBE matchmaker only considers the name of the data types. The analysis of the data type structure is now under investigation by considering work on service compatibility based on subtyping theory [1]. As a consequence, in case of comparison of complex data types, $dataTypeSim(dt_a, dt_b) = nameSim(dt_a.name, dt_b.name)$. Due to its importance, in the next section we discuss in detail the *nameSim* function and the effects of its parametrization for the evaluation of the Web service interfaces similarity.

1.4 Lessons Learned

1.4.1 Evaluation Results

Generally speaking, in URBE the similarity among two names relies on a *domain specific knowledge base* and a *general purpose ontology*. The *domain specific knowledge base* includes terms related to a given application domain. We assume that this ontology can be built by a domain expert also analyzing the terms included in the Web services published in the registry. The *general purpose knowledge base* includes all the possible terms.

The effect of considering these two different ontologies is emphasized when using the SAWSDL Test Collection[1] as the benchmark for evaluating the accuracy of the matchmaker. The Web services in this collection include annotations only for some of the elements composing the interfaces. Indeed, in such a benchmark the operation names are usually not annotated, whereas the data types of the input and output parameters are annotated. Moreover, the annotations refer to concepts stored in a set of ontologies that, in some cases, are related each other. This scenario affects the way in which $fSim$ is computed: the function *nameSim* compares the names used to define the elements of the Web service description if the annota-

[1] The test collection is available at `http://projects.semwebcentral.org/projects/sawsdl-tc/`

tion is not available, and WordNet[2] is adopted as general purpose knowledge base. Otherwise, *nameSim* will invoke the function $annSim(a_q, a_p) \rightarrow [0..1]$ to compare the annotations of these elements, if they exist. In this case, the ontologies specified in the annotations hold the role of domain specific knowledge bases. For instance, giving the example in Listing 1.2, at operation level, URBE has to consider the name `get_PRICE` since no annotation is available. On the contrary, at the parameter level, the *parSim* function can exploit the annotations for the data types `PriceType` and `BookType`: namely, `concept.owl#Price` and `books.owl#Book`.

```
<wsdl:definitions ...>
  <wsdl:types>
      <xsd:element name="Book" type="BookType" .../>
      <xsd:element name="Price" type="PriceType" .../>
      <xsd:complexType name="PriceType"
                       sawsdl:modelReference="concept.owl#Price">
        ...
      </xsd:complexType>
      <xsd:complexType name="BookType"
                       sawsdl:modelReference="books.owl#Book">
        ...
      </xsd:complexType>
      <xsd:simpleType name="Currency"
                  sawsdl:modelReference="currency.owl#Currency"/>
      <xsd:simpleType name="Author"
                  sawsdl:modelReference="books.owl#Author"/>
      <xsd:simpleType name="Title"
                  sawsdl:modelReference="books.owl#Title"/>
      <xsd:simpleType name="Book-Type"
                  sawsdl:modelReference="books.owl#Book-Type"/>
    </xsd:schema>
  </wsdl:types>
  <wsdl:message name="get_PRICEResponse">
    <wsdl:part name="_PRICE" type="tns:PriceType" />
  </wsdl:message>
  <wsdl:message name="get_PRICERequest">
    <wsdl:part name="_BOOK" type="tns:BookType" />
  </wsdl:message>
  <wsdl:portType name="BookPriceSoap">
    <wsdl:operation name="get_PRICE">
      <wsdl:input message="tns:get_PRICERequest" />
      <wsdl:output message="tns:get_PRICEResponse" />
    </wsdl:operation>
  </wsdl:portType>
...
</wsdl:definitions>
```

Listing 1.2 Sample Annotated File

[2] http://wordnet.princeton.edu/

The *nameSim* function, due to the nature of the names normally included in an automatically generated WSDL, can be applied only after a tokenization process which produces the set of terms to be actually compared. Considering our example, terms like `get_PRICE` are difficult to find in WordNet or in any other ontologies. On the contrary, the resulting tokens can be found. Thus, giving $n_a = \{t_{a,i}\}$ and $n_b = \{t_{b,j}\}$ as the set of tokens composing n_a and n_b:

$$nameSim(n_a, n_b) = maxSim(termSim, \{t_{a,i}\}, \{t_{b,j}\}) \tag{1.2}$$

where $termSim : (t_a, t_b) \rightarrow [0..1]$ is the function that computes the similarity between two tokens. In the literature, several approaches are available to state the similarity and the relatedness among terms [12]. These algorithms usually calculate the similarity based on the relationships among terms defined in a reference ontology (e.g., *is-a*, *part-of*, *attribute-of*). In our approach, to compute the similarity among terms we adopt the approach proposed by Seco et al. [15] where the authors adapt existing approaches relying on the assumption that concepts with many hyponyms[3] convey less information than concepts that have less hyponyms or any at all (i.e, they are leaves in the ontology).

About the annotation similarity, $annSim : (a_q, a_p) \rightarrow [0..1]$ receives as input two annotations and returns their similarity according to the way in which they are related in the reference ontology. In the current implementation, we assume that a_q and a_p are included in the same ontology, otherwise *annSim* returns 0. In future workwill calculate the similarity of annotations referring to different ontologies. Since the annotations can be classes or properties, as shown in the Listing 1.3, the *annSim* has different behaviors.

```
function annSim(a_q, a_p) : double
  if ((a_q is class) and (a_p is class)) or
     ((a_q is property) and (a_p is property))
    annSim = 1/(pathlength(a_q, a_p)+1)
  elseif (a_q is class) and (a_p is property) and
         (a_q = domain(a_p))
        annSim = 1/#properties in a_q
  elseif (a_q is property) and (a_p is class) and
         (a_p = domain(a_q))
        annSim = 1
end function
```

Listing 1.3 Annotation similarity function

In case both annotations are classes or both annotations are properties, to compute the similarity between the two annotations we take into account the subsump-

[3] A hyponym is a word of more specific meaning than a general term applicable to it, i.e., spoon is a hyponym of cutlery.

tion path which connects them in the knowledge base. If there is no paths connecting the classes, or properties, the similarity is 0. In case a_q is a class and a_p a property, it is required that the domain of the property corresponds to the class. If so, it means that (i) the annotation in the query, i.e., a_q, refers to a class with all its properties, and (ii) the annotation in the published service, a_p, refers only to one of those properties. Finally, in the opposite case, i.e., a_q is a property and a_p is a class, if a_p corresponds to the domain of the property a_q, the similarity between annotations is 1; otherwise, it is 0. Indeed, now (i) the annotation in the query refers to a specific property, and (ii) the annotation in the published service certainly includes such a property since it refers to the whole set of properties for the defined class.

According to this scenario, URBE is based on both knowledge bases: the domain specific knowledge base offers more accuracy in the relationships of the terms and is mainly used in *annSim*; the general purpose one offers wider coverage and it is mainly used by *nameSim*. This happens because in a general purpose knowledge base a word may have more that one synonym set (a.k.a. *synset*): a set of one or more synonyms that are interchangeable in some context. On the contrary, we assume that in a domain specific ontology each word has a unique sense with respect to the domain itself. For instance, if we consider the noun *currency*, in WordNet it has two synsets. The first one is about the financial domain, i.e., the metal or paper medium of exchange currently being used; the second one is about a generic meaning, i.e., general acceptance or use. Comparing the term *currency* with the term *money*[4] we can realize that they are strictly related only if we consider the financial domain. On the other hand, if we consider the other synset the relationship is looser. Therefore, in case of general purpose ontologies, since it is hard to figure out which is the correct domain to consider, we employ the average similarity for each synset.

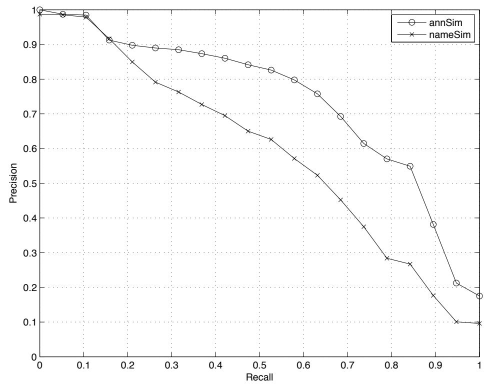
The approach presented in this chapter has been implemented in a prototype. The source code of URBE is freely downloadable from SourceForge[5]. In the current implementation, WordNet is available as a general purpose ontology and the Java WordNet similarity library[6] developed by Seco et al. [15] is used to compute similarity between terms in WordNet. SAWSDL4J is used to parse the SAWSDL and WSDL files. An open-source implementation of a Mixed Integer Linear Programming solver, i.e., LpSolve[7], is used to solve the linear programming model on which $fSim$ relies. Finally, the Jena library is used for accessing OWL-based domain-specific ontologies.

To evaluate how this approach affects the accuracy of the matchmaker, Figure 1.3 shows how the precision-recall trend changes if the annotations are considered or not, i.e., if only WordNet or also the domain specific knowledge bases are taken into account. For this experiment, we ran URBE twice: one time ignoring the anno-

[4] `http://marimba.d.umn.edu/cgi-bin/similarity.cgi`
[5] `http://sourceforge.net/projects/urbe/`
[6] `http://eden.dei.uc.pt/~nseco/javasimlib.tar.gz`
[7] `http://sourceforge.net/projects/lpsolve`

tations and comparing only the names (*nameSim* curve) and the other considering the annotation similarity (*annSim* curve), too.[8]

As shown in the table of the Figure 1.3, the existence of annotation improves not only the average precision (AP) by almost 20%, but also the response time by about 6%. This difference about the response time depends on the lower time required to compare the annotations in the ontology with respect to the time required to compare the names in the WordNet. Indeed, the annotation analysis does not need to tokenize the terms and, as a consequence, it does not result in more than one comparison.



	nameSim	*annSim*	*var*
Average Precision	60.6%	72.7%	+19.9%
Average Response Time	39.195 sec	36.996 sec	-5.9%

Fig. 1.3 Annotation similarity and Name similarity comparison.

Regardless of the existence of the annotations, it is also interesting to analyse how the accuracy of the URBE matchmaker varies with respect to a variation of the weights *wPTNameSim*, *wOpNameSim*, and *wParNameSim*. The results discussed above are obtained with the following configuration: $wPTNameSim = 0.1$, $wOpNameSim = 0.1$, and $wParNameSim = 0.7$. This means that to obtain the best results, the influence of the comparison at the different levels is the one shown in Figure 1.4. It is worth noting that these values for the weights are valid for the adopted benchmark but, considering the different types of services included in it, generally speaking we can say that the analysis of parameter names has more than half of the overall importance, i.e., $56.7\%(= 0.9 \cdot 0.9 \cdot 0.7)$, whereas the `portType` and `operation` names comparisons have a lower influence.

Varying these weights, it is interesting to see how the accuracy varies in a different way with respect to existence or absence of annotations. Now, for the sake of clarity, we decide to vary only *wParNameSim* due to its importance in the overall computation as discussed above. Tables 1.1 and 1.2 report the average precision

[8] All the experiments discussed in this chapter have been done on an Windows XP Pro installed on a Virtual Machine configured with Intel Core 2 Due 2.33 GHz and 512MB of RAM. The test collection is SAWSDL-TC v.1 (26 queries and 895 services). The average precision and the response time are obtained using the Semantic Web Service Matchmaker Evaluation Environment (SME2) available at: `http://projects.semwebcentral.org/projects/sme2/`.

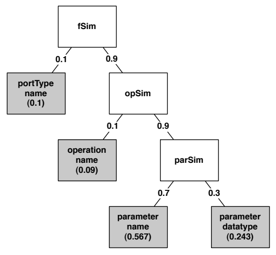


Fig. 1.4 Importance of the service elements in the similarity computation.

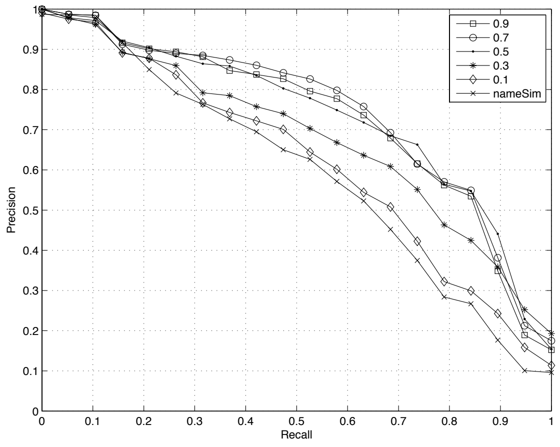


Fig. 1.5 Precision/Recall chart with different values of *wParNameSim*.

of the URBE matchmaker for different values of *wParNameSim*. It is worth noting how the accuracy is much more sensitive to the variation of the weight in case the annotations are considered. This because the annotations are more meaningful than the names. As a consequence, if two annotations are related *annSim* returns a value that is close to 1. Thus, even lowering the weight, the contribution of this comparison remains quite significant. On the other side, when the names are considered, even if they are strictly related, a significant difference in a token makes the result of *nameSim* lower. Thus, with lower values of the weight, the impact of this contribution is more reduced. As shown in the tables, the gap between the best and worst cases is 11.8%, whereas in case the annotations are not considered this gap is only 1.16%. Figure 1.5 shows the precision and recall curves for each of the five different values assigned to *wParNameSim* that correspond to the average precisions reported in Table 1.2 plus the best curve for the *nameSim* (i.e., when no annotations are available and *wParNameSim* $= 0.7$).

It is worth noting that, even in the worst case, i.e., *wParNameSim* $= 0.1$, the similarity computed considering the annotations is better than the best case when considering only the names, i.e., *nameSim* curve.

wParNameSim	*AP*
0.9	58.0%
0.7	60.6%
0.5	58.5%
0.3	58.4%
0.1	57.5%

Table 1.1 AP for different values of wParNameSim (considering nameSim)

wParNameSim	*AP*
0.9	71.3%
0.7	72.7%
0.5	72.0%
0.3	66.4%
0.1	60.9%

Table 1.2 AP for different values of wParNameSim (considering annSim)

Focusing on the other weights, i.e., $wPTNameSim$ and $wOpNameSim$, Figures 1.6 and 1.7 show how the best trends are obtained for lower values of these weights. It is worth noting that it does not mean that name analysis should be skipped. Indeed, the 'off' curves, that represent a null value of these two weights, are under the optimal behaviour, obtained with $wPTNameSim = 0.1$ and $wOpNameSim = 0.1$.

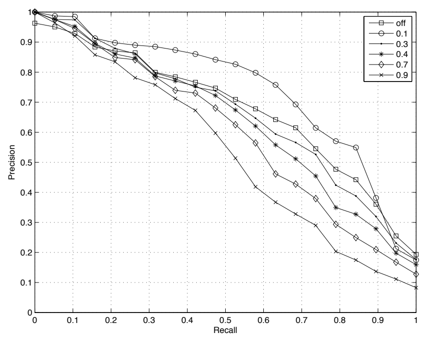


Fig. 1.6 Precision/Recall chart with different values of $wPTNameSim$.

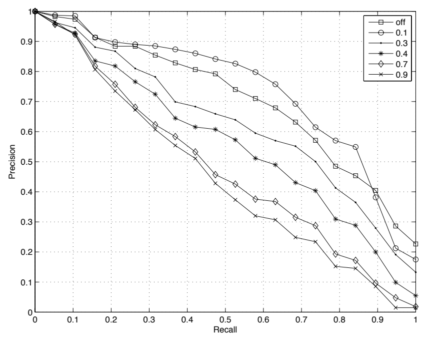


Fig. 1.7 Precision/Recall chart with different values of $wOpNameSim$.

1.4.2 Advantages and Disadvantages

In URBE the similarity computation takes into account all the aspects that define a Web service interface: the number of elements, the data types, the names, and the annotations. Although, the Precision/Recall trends is comparable to other approaches [10, 11], the response time is the main drawback. Indeed, the execution-time of $fSim$ is directly affected by the exponential complexity to solve the assignment in bipartite graphs problems [19]. Some heuristics [18] reduce the complexity to $O(n^2)$, where n is the sum of the cardinalities of the two sets we consider. As mentioned above, the current implementation uses LpSolve that is in charge of the resolution of the assignment problem, and the response time depends on the strategy internally adopted by this tool. According to the analysis presented in [9], on average a Web service has 3 operations and each operation has 4 parameters (considering both inputs and outputs). Comparing two Web services with these characteristics, our approach requires about 0.3 sec to calculate the similarity.

According to these values, the response time is acceptable if URBE is adopted as a tool for supporting the designers (as it is actually designed for). On the contrary, URBE is not suitable for run-time discovery, for instance in support of automatic composition. Nevertheless, URBE remains useful even at run-time if the number of available services is low. This requires a pre-filtering step before the actual execution of URBE. During this phase, given the initial query, the filter needs to figure out a set of services from the repository that can be considered as the best candidates. Next, URBE will perform a finer-grained analysis to rank them with respect to the similarity to the query.

The possibility to support both WSDL and SAWSDL Web service interface descriptions is another advantage of URBE. Indeed, although the accuracy of the algorithm in case of WSDL is not as good as in case of annotated interfaces, compared to the existing methods [13] the results remain acceptable. In this way, URBE can be adopted in a high number of situations since WSDL is the most used language for describing Web service interfaces, and has also the advantage that it can be automatically generated started from the service implementations.

Finally, from a semantic perspective, we noticed how considering annotations brings a reduction of the response time since the annotation similarity is computed only considering the path connecting the concepts in the ontologies or, in case properties are involved, the relationship between classes and properties. Nevertheless, annotation similarity also improves the accuracy.

1.4.3 Conclusions for Future Work

Further work must focus on improving performance in terms of execution time. First of all, a clustering of the Web services published in the registry can be periodically done in order to automatically create the application domain-based classification. In case the Web services are also described with OWL-S, we plan to exploit also in creating these clusters. Second, we will refer to [4], where the authors propose a set of basic principles towards efficient semantic Web service discovery. In particular, these principles focus on: semantic level (reducing ontology management) and matching level (reducing the number of comparisons).

Moreover, the semantic analysis of a WSDL specification can consider differently the comparison between method names and parameter names. About the former, the verb is more important since a method name should define an action. About the latter, the parameter name similarity should mainly consider the noun, i.e., the meaning of data on which the action is performed or its output. Considering the SAWSDL analysis, the next steps aim to consider in a single step the annotations at different levels in the structure. For instance, if the required operation is called *formatDocument*, whereas the offered operation has the operation and input parameter annotated with the ontology concepts *format* and *document*, then we should realize that they are strictly related.

So far, URBE has been designed to support the service substitution from the perspective of a client that is looking for a service providing an interface similar to one previously invoked. Changing the standpoint to the provider perspective, we can assume that the user request is mandatory and the provider must evolve its service to satisfy the user requirements. Inside the research area of Service Evolution [1], which also deals with this scenario, URBE can be adopted as a metric for estimating the effort needed to evolve a service to another one by analyzing the initial and the final interfaces.

Finally, we are now also investigating the possibility of translating the idea underlying the URBE matchmaker to a Constraint Logic Programming (CLP) problem [14]. In this case, from the query the matchmaker can define a set of contraints that the service to be analyzed needs to deal with. If all the constraints are satisfied then the similarity will be the maximum. On the other side a null similarity would indicate that none of the contraints are satisfied. In particular, the work needs to be based on an extension of CLP, i.e., Semiring-based Contraint Logic Programming [6], which also allows for ranking the solutions obtained by the constraint analysis process.

1.5 Summary

In this paper we have presented URBE, an approach for evaluating the similarity between Web service interfaces for substitution purposes. The Web service requestor, after submitting the interface of the desired Web service, can obtain a list of similar Web services. The evaluation of the similarity between Web services considers both the semantics and the structure of a WSDL description. The semantic analysis takes into account the names adopted to describe the elements composing a Web service (operations and parameters), whereas the structure analysis takes into account the number of operations as well as the number and data types of the parameters. In addition, our approach also supports SAWSDL as a description model. In this case, the semantic analysis takes advantage of the semantic relationships between annotations in SAWSDL, as demonstrated in the Semantic Service Selection (S3) Contest [10, 11].

A prototype of URBE has been developed as a UDDI-compliant registry that supports our retrieval model and has been used to validate of our approach. Based on this implementation, in this chapter we have shown how the annotations included in a SAWSDL file can influence the accuracy of the matchmaking algorithm in terms of precision and recall and how this accuracy is also affected by the different elements composing a Web service interface description: `portType`, `operation`, and `parameters`.

References

1. Andrikopoulos, V., Benbernou, S., Papazoglou, M.P.: Managing the Evolution of Service Specifications. In: Proceedings of the 20th Int'l Conference on Advanced Information Systems Engineering, CAiSE '08, pp. 359–374. Springer-Verlag, Berlin, Heidelberg (2008)
2. Ardagna, D., Comuzzi, M., Mussi, E., Pernici, B., Plebani, P.: PAWS: A Framework for Executing Adaptive Web-Service Processes. IEEE Software **24**, 39–46 (2007)
3. Asratian, A., Denley, T., Häggqvist, R.: Bipartite Graphs and their applications. Cambridge University Press (1998)
4. Ben Mokhtar, S., Kaul, A., Georgantas, N., Issarny, V.: Towards Efficient Matching of Semantic Web Service Capabilities. In: A. Bertolino, A. Polini (eds.) in Proceedings of International Workshop on Web Services Modeling and Testing (WS-MaTe2006), pp. 137–152. Palermo, Italy (2006)
5. Biron, P., Malhotra, A.: XML Schema Part 2: Datatypes Second Edition (W3C Recommendation). `http://www.w3.org/TR/xmlschema-2/`. Last accessed 04 Mar 2011. (2004)
6. Bistarelli, S., Montanari, U., Rossi, F.: Semiring-based constraint logic programming: syntax and semantics. ACM Trans. Program. Lang. Syst. **23**, 1–29 (2001)
7. Garofalakis, J., Panagis, Y., Sakkopoulos, E., Tsakalidis, A.: Contemporary Web service discovery mechanisms. Journal of Web Engineering **5**(3), 265–290 (2006)
8. Kephart, J.O., Chess, D.M.: The Vision of Autonomic Computing. IEEE Computer **36**(1), 41–50 (2003)
9. Kil, H., Oh, S.C., Lee, D.: On the topological landscape of Web services Matchmaking. In: CEUR (ed.) Proc. Int'l Workshop on Semantic Matchmaking and Resource Retrieval (VLDB-SMR'06), vol. 178 (2006). Published on line
10. Klusch, M.: OWL-S and SAWSDL Service Matchmakers. S3 Contest 2008 Summary Report. `http://www-ags.dfki.uni-sb.de/~klusch/s3/s3c-2008.pdf`. Last accessed 04 Mar 2011. (2008)
11. Klusch, M.: OWL-S and SAWSDL Service Matchmakers. S3 Contest 2010 Summary Report. `http://www-ags.dfki.uni-sb.de/~klusch/s3/s3c-2010-summary-report-v2.pdf`. Last accessed 04 Mar 2011. (2010)
12. Pedersen, T., Patwardhan, S., Michelizzi, J.: WordNet::Similarity - Measuring the Relatedness of Concepts. In: Proc. National Conf. on Artificial Intelligence, July 25-29, San Jose, California, USA, pp. 1024–1025 (2004)
13. Plebani, P., Pernici, B.: URBE: Web Service Retrieval Based on Similarity Evaluation. IEEE Trans. on Knowl. and Data Eng. **21**(11), 1629–1642 (2009)
14. Rossi, F., van Beek, P., Walsh, T.: Handbook of Constraint Programming (Foundations of Artificial Intelligence). Elsevier Science Inc., New York, NY, USA (2006)
15. Seco, N., Veale, T., Hayes, J.: An Intrinsic Information Content Metric for Semantic Similarity in Wordnet. In: Proc. Eureopean Conf. on Artificial Intelligence (ECAI'04), Valencia, Spain, August 22-27, pp. 1089–1090. IOS Press (2004)
16. Stroulia, E., Wang, Y.: Structural and Semantic Matching for Assessing Web-service Similarity. Int'l J. Cooperative Inf. Syst. **14**(4), 407–438 (2005)
17. Sun Microsystems: Code conventions for the Java programming language. `http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html`. Last accessed 04 Mar 2011. (1999)
18. Wang, J., Xiao, J., Lam, C., Li, H.: A Bipartite Graph Approach to Generate Optimal Test Sequences for Protocol Conformance Testing using the Wp-method. In: Proc. Asia-Pacific Software Engineering Conf. (APSEC'05), pp. 307–316 (2005)
19. Wolsey, L.: Integer Programming. John Wiley and Sons (1998)
20. Zaremski, A., Wing, J.: Signature matching: a tool for using software libraries. ACM Trans. Softw. Eng. Methodol. **4**(2), 146–170 (1995)