Combining Schema and Level-Based Matching for Web Service Discovery

Alsayed Algergawy[1], Richi Nayak[2], Norbert Siegmund[3], Veit Köppen[3], and Gunter Saake[3]

[1] Department of Computer Science, University of Leipzig, Germany
[2] Queensland University of Technology, 2434 Brisbane, Australia
[3] School of Computer Science, University of Magdeburg, Germany
algergawy@informatik.uni-leipzig.de, r.nayak@qut.edu.au,
{n.siegmund,veit.koeppen,saake}@iti.cs.uni-magdeburg.de

Abstract. Due to the availability of huge number of Web services (WSs), finding an appropriate WS according to the requirement of a service consumer is still a challenge. In this paper, we present a new and flexible approach, called SeqDisc, that assesses the similarity between WSs. In particular, the approach exploits the Prüfer encoding method to represent WSs as sequences capturing both semantic and structure information of service descriptions. Based on the sequence representation, we develop an efficient sequence-based schema matching approach to measure the similarity between WSs. A set of experiments is conducted on real data sets, and the results confirm the performance of the proposed solution.

Keywords: Web service, WS discovery, WSDL, Schema matching.

1 Introduction

Web Services (WSs) have emerged as a popular paradigm for distributed computing, and sparked a new round of interest from research and industrial communities. By adopting service oriented architectures (SOA) using WS technologies, enterprises can flexibly solve enterprise-wide and cross-enterprise integration challenges [8]. These advantages of WSs can also be used within a network of embedded systems which access and retrieve information from each other (e.g., a logistic hub consisting of sensors, PDAs, etc. [18]). Web services can then be used as an abstract interface for the devices to overcome communication and data integration problems resulting from the heterogeneity of the devices. Therefore, WSs can also be used to achieve the interoperability of a complex and heterogeneous system.

The research community has identified two major areas of interest: *Web service discovery* and *Web service composition* [15]. In this paper, we present the issue of locating WSs efficiently. As the number of WSs increases, the problem of locating desired service(s) from a large pool of WSs becomes a challenging research problem [20,11,13,7,4]. In addition, if WSs are generated on demand (e.g., to fulfill a certain task for a defined time, see [14,18]), it is difficult to discover the most suitable services. Several solutions have proposed, however, most of them suffer from the following two main disadvantages. Firstly, A large number of these solutions are syntactic-based. These methods use simple keyword search on Web service descriptions, and traditional attribute-based matchmaking algorithms to locate Web services according to a service request. However, these mechanisms are insufficient in the Web service discovery context since since they do not capture the underlying semantic of Web services and/or they partially satisfy the need of user search. This is due to the fact that keywords are often described by a natural language. As a result, the number of retrieved services with respect to the keywords are huge and/or the retrieved services might be irrelevant to the need of their consumers [15]. More recently, this issue sparked a new research into the Semantic Web where some research uses ontology to annotate the elements in Web services [5,16]. Nevertheless, integrating different ontologies may be difficult while the creation and maintenance of ontologies may involve a huge amount of human effort. To address the second aspect, clustering algorithms are used for discovering WSs. However, they are based on keyword search [11,16,15].

Secondly, most of the existing approaches are not scale well to large-scale and to large numbers of services, service publishers, and service requesters. This is due to the fact that they mostly follow a centralized registry approach. In such an approach, there is a registry that works as a store of WS advertisements and as the location where service publication and discovery takes place. The scalability issue of centralized approaches is usually addressed with the help of replication (e.g., UDDI). However, replicated Registries have high operational and maintenance cost. Furthermore, they are not transparent due to the fact that updates occur only periodically.

We see Web service discovery as a matching process, where available services' capabilities satisfy a service requester's requirement. There are two main aspects that should be considered during solving the matching process: the *quality* of the discovered service and the *efficiency* especially in large-scale environments. To obtain a better quality, not only is the textual description of Web services sufficient, but also the underlying structures and semantics should be exploited. Also to get a better performance, an efficient methodology should be advised.

In this paper, we propose a flexible and efficient approach, called *SeqDisc*, for assessing the similarity of Web services, which can be used to support locating WSs. We first represent WS document specifications described in WSDL as rooted, labeled trees, called *service trees*. By investigating service trees, we observe that each tree can be divided into two parts (subtrees), namely the *concrete* and *abstract* parts. We discover that the concrete parts from different WSDL documents have the same hierarchal structure, but may have different names. Therefore, we develop a *level-based matching* approach, which computes the name similarity between concrete elements at the same level. However, the abstract parts of the WSDL documents have differences in structure and semantics. To efficiently access the abstract elements, we represent them using the Prüfer encoding method [17], and then apply our sequence-based schema matching approach to the sequence representation. A set of experiments is conducted in order to validate our proposed approach employing real data sets. The experimental results showed that the approach performs well.

2 Preliminaries

A Web service is a software component identified by an URI, which can be accessed via the Internet through its exposed interface. Three fundamental layers are required to provide or use WSs [6]. First, WSs must be network-accessible to be invoked, HTTP is the de-facto standard network protocol for Internet available WSs. However, other network protocols can be used to enable the use of Web services in other kinds of networks (e.g., sensor networks). Second, WSs use XML-based messaging for exchanging information, and SOAP[1] is the chosen protocol. Finally, it is through a service description that all the specification for invoking a WS are made available; WSDL[2] is the de-facto standard for XML-based service description.

2.1 Web Service Modeling

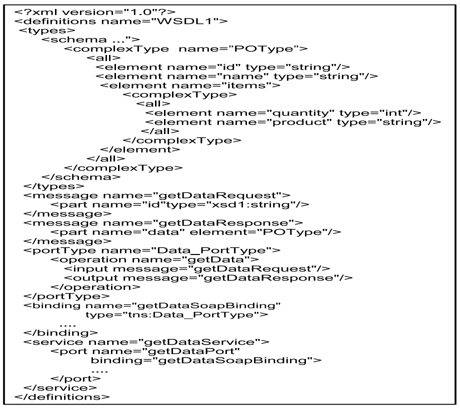
In this paper, we represent a WSDL specification as a rooted labeled tree, called *service tree, ST*, defined as follows:

Definition 1. *A service tree, (ST), is a 3-tuple element; $ST = (N, E, Lab)$, where: $N = \{n_{root}, n_2, ..., n_n\}$ is the set of nodes representing WSDL document elements, where n_{root} is the root node of the tree; $E = \{(n_i, n_j) | n_i, n_j \in N\}$ is the set of edges representing the parent-child relationship between WSDL document elements; and Lab is a set of labels associating to WSDL document elements describing the properties of them.*

Examining the hierarchical structure of the WSDL document, we found that a *service* consists of a set of *ports*, each containing only one *binding.* A binding contains only one *portType.* Each portType consists of a set of *operations*, each containing an *input message* and a set of *output messages.* A message includes a set of *parts*, where each part describes the logical content of the message. All WSDL document elements (except part elements) have two main properties: the *type* property to indicate the type of the element (*port, binding, operation,...*) and the *name* property to distinguish between similar type elements.

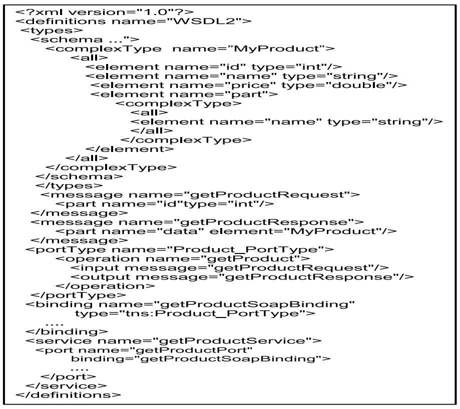
From the hierarchal structure of a service tree, we divide its elements into a *concrete* part and *abstract* part. The intuition for this classification is that service trees representing different web services have the same structure from the root node to the part node, while the structure of the remaining depends on the content of operation messages. The following are definitions for concrete and abstract parts of a service tree.

[1] `http://www.w3.org/TR/soap/`
[2] `http://www.w3.org/TR/wsdl20/`



```
<?xml version="1.0"?>
<definitions name="WSDL1">
 <types>
     <schema ...">
         <complexType  name="POType">
             <all>
               <element name="id" type="string"/>
               <element name="name" type="string"/>
                <element name="items">
                    <complexType>
                       <all>
                        <element name="quantity" type="int"/>
                        <element name="product" type="string"/>
                       </all>
                    </complexType>
                </element>
             </all>
         </complexType>
     </schema>
  </types>
  <message name="getDataRequest">
       <part name="id"type="xsd1:string"/>
  </message>
  <message name="getDataResponse">
       <part name="data" element="POType"/>
  </message>
  <portType name="Data_PortType">
       <operation name="getData">
          <input message="getDataRequest"/>
          <output message="getDataResponse"/>
       </operation>
  </portType>
  <binding name="getDataSoapBinding"
              type="tns:Data_PortType">
         ....
  </binding>
  <service name="getDataService">
       <port name="getDataPort"
               binding="getDataSoapBinding">
               ....
       </port>
  </service>
</definitions>
```

(a) WSDL1: *getData* Web Service.



```
<?xml version="1.0"?>
<definitions name="WSDL2">
 <types>
   <schema ...">
     <complexType  name="MyProduct">
         <all>
            <element name="id" type="int"/>
            <element name="name" type="string"/>
             <element name="price" type="double"/>
             <element name="part">
                  <complexType>
                     <all>
                     <element name="name" type="string"/>
                     </all>
                   </complexType>
            </element>
         </all>
     </complexType>
    </schema>
    </types>
     <message name="getProductRequest">
        <part name="id"type="int"/>
   </message>
   <message name="getProductResponse">
        <part name="data" element="MyProduct"/>
   </message>
  <portType name="Product_PortType">
        <operation name="getProduct">
           <input message="getProductRequest"/>
           <output message="getProductResponse"/>
        </operation>
   </portType>
  <binding name="getProductSoapBinding"
           type="tns:Product_PortType">
      ....
  </binding>
  <service name="getProductService">
    <port name="getProductPort"
           binding="getProductSoapBinding">
           ....
     </port>
  </service>
</definitions>
```

(b) WSDL2: *getProduct* Web service.

Fig. 1. Two WSDL specifications

Definition 2. *A concrete part of a service tree (ST) is the subtree (ST_C) extending from the root node to the portType element, such that $ST_C = \{N_C, E_C, Lab_C\} \subset ST$, $N_C = \{n_{root}, n_{port1}, n_{binding1}, n_{portType1}, ..., n_{portType_l}\} \subset N$, where l is the number of concrete elements in the service tree.*

Definition 3. *An abstract part of a service tree (ST) is the set of subtrees rooted at operation elements, such that $ST_A = \{ST_{A_1}, ST_{A_2}, ..., ST_{A_k}\}$, where k is the number of operations in the service tree.*

A service tree comprises a concrete part and an abstract part, i.e., $ST = ST_C \cup ST_A$. To assess the similarity between two WSs, we consequently compare their concrete and abstract parts. The problem of measuring similarity between Web services is converted into the problem of tree matching- comparing their concrete and abstract parts.

Let us now introduce an example of assessing the similarity between two WSs, which is taken from [20]. As shown in Fig. 1, we have two WSs described by two WSDL documents $WSDL1$ and $WSDL2$, respectively. $WSDL1$ contains one operation , *getData*, that takes a string as input and returns a complex data type named POType, which is a product order. The second document contains one operation, *getProduct*, that takes an integer as input and returns the complex data type MyProduct as output.

3 The SeqDisc Approach

The proposed SeqDisc approach is based on the exploitation of the structure and semantic information from WSDL documents. The objective is to develop a flexible and efficient approach that measures the similarity between WSs. The measured similarity is used as a guide in locating the desired service. To realize this goal, we first analyze WSDL documents and represent them as service trees using Java APIs for WSDL (JWSDL) and a SAX parser for the contents of the XML schema (the types element). Each service tree is examined to extract its concrete and its abstract parts. The concrete parts from different service trees have the same hierarchal structure. Hence, the similarity between concrete parts of two web services is computed using only concrete part element names by comparing elements with the same level. We call this type of matching level-based matching. Abstract parts from different service trees have different structures based on the message contents. Therefore, we propose a sequence-based abstract matching approach to measure the similarity between them. By the two mechanisms we gain a high flexibility in determining the similarity between WSs. In Section 6, we show two possibilities to compute the similarity. The first is to exploit abstract parts (operations), while the second is to use both the abstract and concrete parts. Furthermore, the proposed approach scales well. As it will be shown, the level-based matching algorithm has a linear time complexity as a function of the number of concrete elements, while the sequence-based matching algorithm benefits from the sequence representation to reduce time complexity.

4 Level-Based Matching

Once obtaining the concrete parts of service trees, $ST_{C1} \subset ST1$ and $ST_{C2} \subset ST2$, we apply our level-based matching algorithm that linguistically compares nodes at the same level, as shown in Fig. 2(a). The level-based approach considers only semantic information of concrete elements. It measures the elements (tag names) similarity by comparing each pair of elements at the same level based on their names.

Algorithm 1 accepts the concrete parts of the service tress, ST_{C1}, ST_{C2}, and computes the name similarity between the elements of the concrete parts. It starts by initializing the matrices, wherein the name similarities are kept. We have three levels for each service tree, *line* 2. When the loop index equals 1, $i = 1$, the algorithm deals with the port nodes, when $i = 2$ it deals with the binding nodes, and with the portType nodes when $i = 3$. To compute the similarity between elements at the same level, the algorithm uses two inner loops, *lines* 3 & 5. It first extracts the name of the node j at the level i, *line* 4, and the name of the node k at the same level, *line* 6. Then, the algorithm uses a name similarity function to compute the name similarity between the names of the nodes, *line* 7. Finally, depending on the level, it stores the name similarity matrix into the corresponding element matrix.

To compute the name similarity between two element names represented as strings, we first break each string into a set of tokens T_1 and T_2 using a customizable tokenizer using punctuation, upper case, special symbols, and digits, e.g. getDataService $\rightarrow$ $\{$get, Data, Service$\}$. We then determine the name similarity between the two sets of name tokens T_1 and T_2 as the the average best similarity of each token with a token in the other set. It is computed as follow:

$$Nsim(T_1, T_2) = \frac{\sum_{t_1 \in T_1} [\max_{t_2 \in T_2} sim(t_1, t_2)] + \sum_{t_2 \in T_2} [\max_{t_1 \in T_1} sim(t_2, t_1)]}{|T1| + |T2|} \quad (1)$$

Algorithm 1. Level-based matching algorithm

```
Require: Two concrete parts, $ST_{C1} \& ST_{C2}$
Ensure: $3 Name\ similarity\ matrices,\ NSimM$
 1: $PortSimM[][] \Leftarrow 0$, $BindSimM[][] \Leftarrow 0$, $PTypeSimM[][] \Leftarrow 0$;
 2: for $i = 1$ to $3$ do
 3:    for $j = 1$ to $l$ do
 4:       $name_1 \Leftarrow getName(ST_{C1}(i, j))$;
 5:       for $k = 1$ to $l'$ do
 6:          $name_2 \Leftarrow getName(ST_{C2}(i, k))$;
 7:          $NSimM[i][j] \Leftarrow NSim(name_1, name_2)$;
 8:       end for
 9:    end for
10:    if $i = 1$ then
11:       $PortSimM \Leftarrow NSimM$;
12:    else if $i = 2$ then
13:       $BindSimM \Leftarrow NSimM$;
14:    else
15:       $PTypeSimM \Leftarrow NSimM$;
16:    end if
17: end for
```

To measure the string similarity between a pair of tokens, $sim(t_1, t_2)$, we use two string similarity measures, namely the edit distance and trigrams [10]. The name similarity between two nodes is computed as the combination (weighted sum) of the two similarity values. The output of this stage is 3 $(l \times l')$ name similarity matrices, $NSimM$, where l is the number of concrete part elements of ST_{C1} and l' is the number of concrete part elements of ST_{C2} per level (knowing that the number of *ports*, the number of *bindings*, and the number of *protType* are equal). In the running example, see Fig. 2(a), $l = 1$ and $l' = 1$.

Algorithm Complexity. The algorithm runs three times, one for every level. Through each run, it compares l elements of ST_{C1} with l' elements of the second concrete part. This leads to a time complexity of $O(l \times l')$, taking into account that the number of elements in each level is very small.

5 Abstract Matching

In contrast to concrete parts, the abstract parts from different service trees have different structures. Therefore, to compute the similarity between them, we should capture both semantic and structural information of the abstract parts of the service trees. To realize this goal, we propose a sequence-based matching approach to achieve this goal. The approach consists of two stages: *Prüfer Sequence Construction* and *Similarity computation.*[3]. The Pre-processing phase is considered with the representation of each abstract item (subtree) as a sequence representation using the Prüfer encoding method. The sequences

[3] For more details about our sequence-based schema matching approach, see [3].

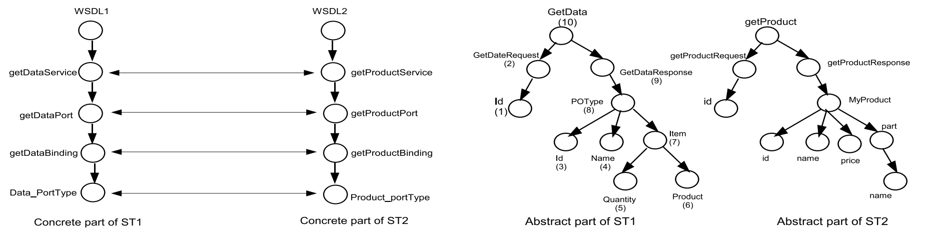
should capture both semantic and structure information of the service tree. The similarity computation phase aims to assess the similarity between abstract parts of different service trees exploiting both information to construct an operation similarity matrix.

The outline of the algorithm implementing the proposed schema matching approach is shown in *Algorithm* 2. The algorithm accepts two sets of abstract parts of the service trees input, $ST_{A1} = \{ST_{A11}, ST_{A12}, ..., ST_{A1k}\}$ and $ST_{A2} = \{ST_{A21}, ST_{A22}, ..., ST_{A2k'}\}$, where each item in the sets represents an operation in the service tree. k and k' are the number of operations in the two abstract parts, respectively. We first analyze each operation (abstract item) and represent it as a Consolidated Prüfer Sequence (CPS) using the Prüfer encoding method. Then, the algorithm proceeds to compare all CPS pairs to assess the similarity between every operation pair using our developed sequence matching algorithms. The returned similarity value is stored in its corresponding position in the operation similarity matrix, $OpSimM$.

Prüfer Sequence Construction. This aims to represent every item in the abstract part set (operation) as a sequence representation using the Prüfer encoding method. The semantic and structural information of service tree operations are captured in Label Prüfer Sequences (LPSs) and Number Prüfer Sequences (NPSs), respectively. The two sequences form what is called a Consolidated Prüfer Sequences ($CPS = (NPS, LPS)$) [19]. They are constructed by doing a *post-order traversal* that tags each node in the service tree operation with a unique traversal number, as shown in Fig. 2(b) for $ST1$. NPS is then constructed iteratively by removing the node with the smallest traversal number and appending its parent node number to the already structured partial sequence. LPS is constructed similarly but by taking the node labels of deleted nodes instead of their parent node numbers.

Example 1. Consider the abstract parts of the two service trees $ST1$ & $ST2$ shown in Fig. 2(b). $CPS(ST_{A1}) = (NPS, LPS)$, where $NPS(ST_{A1})$= (2 10 8 8 7 7 8 9 10 -) and $LPS(ST_{A1})$*.name = (id, getDataReequest, id, name, quantity, product, item, POType, getDataResponse, getData).*

This sequence representation of service trees makes the proposed framework able to cope with the two mentioned aspects in Section 1. From the quality



(a) Concrete parts of WSDL1 & WSDL2. (b) Abstract parts of $WSDL1$ & $WSDL2$.

Fig. 2. Concrete & abstract parts of $WSDL1$ & $WSDL2$

Algorithm 2. Schema matching algorithm

```
Require: Two abstract parts, $ST_{A1} \& ST_{A2}$
    $ST_{A1} = \{ST_{A11}, ST_{A12}, ..., ST_{A1k}\}$
    $ST_{A2} = \{ST_{A21}, ST_{A22}, ..., ST_{A2k'}\}$
Ensure: Operation similarity matrix, OpSimM
 1: $OpSimM[][] \Leftarrow 0$;
 2: for $i = 1$ to $k$ do
 3:     $CPS_1[i] \Leftarrow buildCPS(ST_{A1i})$
 4: end for
 5: for $j = 1$ to $k'$ do
 6:     $CPS_2[j] \Leftarrow buildCPS(ST_{A2j})$
 7: end for
 8: for $i = 1$ to $k$ do
 9:     for $j = 1$ to $k'$ do
10:         $OpSimM[i][j] \Leftarrow computeSim(CPS_1[i], CPS_2[j])$;
11:     end for
12: end for
13: return OpSimM;
```

point of view, CPS captures both semantic information in LPSs and structure information in NPSs, which increases quality of Web service discovery. From performance point of view, CPS provides several structural properties, which enable dealing with service trees in an efficient manner.

Similarity Computation. This stage aims to compute the similarity between abstract parts (operations) from different service trees. This task can be stated as follows: Consider we have two Web service document specifications $WSDL1$ and $WSDL2$, each contains a set of operations represented as the abstract part of the corresponding service tree. $ST_{A1} = \{ST_{A11}, ST_{A12}, ..., ST_{Ak}\}$ represents the operation set belonging to $WSDL1$, while $ST_{A2} = \{ST_{A21}, ST_{A22}, ..., ST_{A2k'}\}$ is the operation set of $WSDL2$. The task at hand is to construct a $k \times k'$ operation similarity matrix, *OpSimM*, where k is the number of operations in $WSDL1$ and k' is the number of operations in $WSDL2$. Each entry in the matrix, $OpSimM[i][j]$, represents the similarity between operation ST_{A1i} from the first set and operation ST_{A2j} from the second. The similarity computation algorithm operates on the sequence representations of service trees, see $Algorithm, line\, 6$, and consists of three steps.

1. *Linguistic matcher.* First, a linguistic similarity algorithm is used to compute a degree of linguistic similarity between the elements of service tree operation pairs exploiting their semantic information represented in LPSs. The output of this step are $k \times k'$ linguistic similarity matrices, $LSimM$. Equation 2 gives the entries of a matrix, where $Nsim(T_i, T_j)$ is computed using the same procedure in Eq. 1, *DataType* is a similarity function to compute the type/data type similarity between nodes, and $combine_l$ is an aggregation

function that combines the name and data type similarities.

$$LSimM[i,j] = combine_l(Nsim(T_i, T_j), DataType(n_i, n_j)) \tag{2}$$

2. *Structural matcher.* Once a degree of linguistic similarity is computed, we use the *structural algorithm* to compute the structural similarity between abstract part elements. This matcher is based on the node context, which is reflected by its ancestors and its descendants. The descendants of an element include both its immediate children and the leaves of the subtrees rooted at the element. The immediate children reflect its basic structure, while the leaves reflect the element's content. We consider three kinds of node contexts depending on its position in the service tree: *child, leaf*, and *ancestor* context. The context of a node is the combination of its ancestor, its child, and its leaf context. Two nodes will be structurally similar if they have similar contexts. To measure the structural similarity between two nodes, we compute the similarity of their child, leaf, and ancestor contexts utilizing the structural properties carried by sequence representations of service trees as follows:
   - **Child Context Similarity**; The child context of a node is the set of its immediate children. This set can be easily extracted from the CPS representation of operations considering each entry in CPS represents an edge from the parent node NPS to its immediate child node LPS. To compute the child context similarity between two nodes $n_i \in CPS1$ and $n_j \in CPS2$, we first extract the child context set for each node, then we get the linguistic similarity between each pair of children in the two sets. We select the matching pairs with maximum similarity values, and finally we take the average of best similarity values.
   - **Leaf Context Similarity**; The leaf context of a node is the set of leaf nodes of subtrees rooted at the node. This set can be efficiently extracted from CPS representation. To determine the leaf context similarity between two nodes $n_i \in CPS1$ and $n_j \in CPS2$, we extract the leaf context set for each node, then we determine the gap between the node and its leaf context set as a vector, and finally we use the cosine measure between the two vectors.
   - **Ancestor Context Similarity**; The ancestor context of a node is the path extending from the root node to the node. To measure the ancestor context similarity between two nodes $n_i \in CPS1$ and $n_j \in CPS2$, first we extract each ancestor context from CPS representation, say path P_i for n_i and P_j for n_j, then we compare the two paths. To compare between paths, we use the scores established in [9].

   The output of this step are $k \times k'$ structural similarity matrices, $SSimM$. Equation 3 gives entries of a matrix, where *child*, *leaf*, and *ancestor* are similarity functions that compute the child, leaf, and ancestor context similarity between nodes respectively, and *combine*$_s$ is an aggregation function combining these similarities.

$$SSimM[i,j] = combine_s(child(n_i, n_j), leaf(n_i, n_j), ancestor(n_i, n_j)) \tag{3}$$

3. After computing both linguistic and structural similarities between Web service tree operations, we combine them. The output of this phase are $k \times k'$ total similarity matrices, $TSimM$. Equation 4 gives the entries of a matrix, where *combine* is an aggregation function combining these similarities.

$$TSimM[i,j] = combine(LSimM[i,j], SSimM[i,j]) \quad (4)$$

Operation Similarity Matrix. We use $k \times k'$ total similarity matrices to construct the Web service operation similarity matrix, *OpSimM*. We compute the total similarity between every operation pairs by ranking element similarities in their total similarity matrix per element, selecting the best one, and averaging these selected similarities. Each computed value represents an entry in the matrix, $OpSimM[i,j]$, which represents the similarity between operation op_{1i} from the first set and operation op_{2j} from the second set.

Example 2. Applying the sequence-based matching approach to abstract parts illustrated in Fig. 2(b), we get $OpSim(getData, getProduct) = 0.75$.

Algorithm Complexity. The worst case time complexity of the schema matching algorithm can be expressed as a function of the number of nodes in each operation, the number of operation in each WS, and the number of WSs. Let n be the average operation size, k be the average operation number, and S be the number of input WSs. Following the same process in [3], it can be proven that the overall worst-case time complexity of the schema matching algorithm between two WSs is $O(n^2k^2)$.

Matching Refinement. For every WS pairs we have two sets of matrices: three *NSimM* matrices that store the similarity between concrete elements, and one *OpSimM* that stores the similarity between two WS operations. This provides the SeqDisc approach more flexibility in assessing the similarity between services. As a consequence, we have two different possibilities to get the similarity.

Using only abstract parts; Given, the operation similarity matrix, *OpSimM*, that stores the similarity between operations of two WSs, how to obtain the similarity between them. We can simply get the similarity between the two services by averaging the similarity values in the matrix. However, this method produce smaller values, which do not represent the actual similarity among services. And due to uncertainty inherent in the matching process, the best matching can actually be an unsuccessful choice [12]. To overcome these shortcomings, similarity values are ranked up to top-2 ranking for each operation. Then, the average value is computed for these candidates.

Using both abstract and concrete parts; The second possibility to assess the similarity between WSs is to exploit both abstract and concrete parts. For any operation pair, $op_{1i} \in WSDL1$ and $op_{2j} \in WSDL2$, whose similarity is greater than a predefined threshold (i.e. $OpSimM[i,j] > th$), we increase the similarity of their corresponding parents (portType, binding, and port, respectively).

6 Experimental Evaluation

In order to evaluate the degree to which the SeqDisc approach can distinguish between WSs, we need to obtain families of related specifications. We found such a collection published by XMethods[4] and QWS data set [1]. We selected 78 WSDL documents from six different categories. Table 1 shows these categories and the number of Web services inside each one. Using the "analyze WSDL" method provided by XMethods, we identify the number of operations in each WS, and get the total number of operations inside each category, as shown in the table. All the experiments below share the same design: each service of the collection was used as the basis for the desired service; this desired service was then matched against the complete set to identify the best target service(s).

Table 1. Data set specification

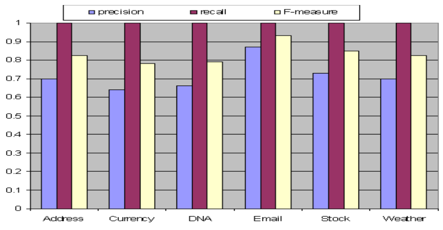
Category	No. of WSs	NO. of operations	Size (KB)
Address	13	50	360
Currency	11	88	190
DNA	16	48	150
Email	10	50	205
Stock quite	14	130	375
Weather	13	110	266

6.1 Experimental Results

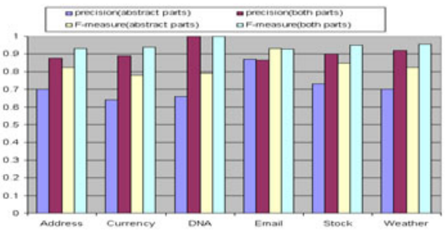
We use precision, recall, and F-measure to evaluate the effectiveness of the SeqDisc framework. We have two possibilities to assess Web discovery process by finding the similarity between Web services depending on the exploited information of WSDL specifications.

Assessing the WS similarity using only abstract parts (operations). In the first set of experiments, we match abstract parts of each service tree from each category against the abstract parts of all other service trees from all categories. Then, we select a set of candidate services, such that the similarity between individual candidate services and the desired one is greater than a predefined threshold. Precision and recall are then calculated for each service within a category. These calculated values are averaged to determine the average precision and recall for each category. Precision, recall and F-measure are calculated for all categories and illustrated in Fig.3(a). There are several interesting findings, which are evident in this figure. First, the SeqDisc approach has the ability to discover all WSs from a set of relevant services. As can be seen, across different six categories, the approach has a recall rate of 100% without missing any candidate service. This ability reflects the strong behavior of the approach of exploiting both semantic and structural information of WSDL specifications in an effective way. Second, Fig. 3 also shows that the ability of the approach to provide relevant WSs from a set of retrieved services is reasonable. The precision of the approach across six categories ranges between 64% and 86%. This means that while the approach does not miss any candidate service, however, it produces false match candidates. This is due to the WS assessment approach is based on lightweight semantic information and does not use any external dictionary or ontology. Finally, based on precision and recall, our framework is almost accurate with F-measure ranging from 78% to 93%.

[4] `http://www.xmethods.net`



(a) Quality measures (abstract parts only).



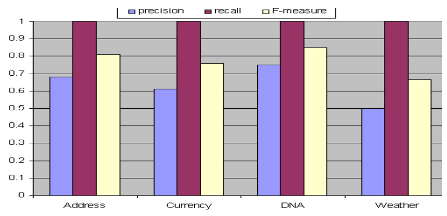
(b) Quality measures (both parts).

Fig. 3. Effectiveness evaluation of SeqDisc

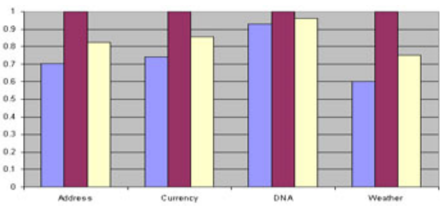
Assessing the WS similarity using abstract and concrete parts. In this set of experiments, we matched the whole parts (both abstract and concrete) of each service tree against all other service trees from all categories. Then, we selected a set of candidate services, such that the similarity between individual candidate services and the desired one is greater than a predefined threshold. Precision and recall are then calculated for each service within a category. These calculated values are averaged to determine the average precision and recall for each category. Precision and F-measure are calculated for all categories and illustrated in Fig. 3(b). We also compared them against the results of the first possibility. The results are reported in Fig. 3(b). The figure represents a number of appealing findings. (1) The recall of the approach remains at the unit level, i.e. no missing candidate services. (2) Exploiting more information about WSDL documents improves the approach precision, i.e. the number of false retrieved candidate services decreases across six different categories. The figure shows that the precision of the approach exploiting both concrete and abstract parts of service trees ranges between 86% in the Email category and 100% in the DNA category. (3) The first two findings lead to the quality of the approach is almost accurate with F-measure ranging between 90% and 100%.

Effect of Individual Matchers. We also performed another set of experiments to study the effect of individual matchers (linguistic and structure) on the effectiveness of WS similarity. To this end, we used data sets from the Address, Currency, DNA, and Weather domains. We consider the linguistic matcher utilizing either abstract parts or concrete and abstract parts. Figure 4 shows matching quality for these scenarios.

The results illustrated in Fig. 4 show several interesting findings. (1) Recall of the SeqDisc approach has a value of 1 across the four domains either exploiting only abstract parts or exploiting both parts, as shown in Fig. 4(a,b). This means that the approach is able to discover the desired service even if the linguistic matcher is only used. (2) However, precision of the approach decreases across the tested domains (except only for the DNA domain using the abstract parts). For example, in the Address domain, precision decreases from 88% to 70% utilizing both parts, and it reduces from 92% to 60% utilizing both parts in the Weather domain. This results in low F-measure values compared with the results shown in Fig. 3. (3) Exploiting both abstract and concrete parts outperforms exploiting



(a) Utilizing abstract parts.



(b) Utilizing both parts.

Fig. 4. Effectiveness evaluation of SeqDisc

only the abstract parts. This can be investigated by comparing results shown in Fig. 4(a) to results in Fig. 4(b).

In summary, using only the linguistic matcher is not sufficient to assess the similarity between WSs. Hence, it is desirable to consider other matchers. As the results in Fig. 3 indicate that the SeqDisc approach employing the structure matcher is sufficient to assess the similarity achieving F-measure between 90% and 100%.

Performance Comparison. Besides studying the performance of the SeqDisc approach, we also compared it with the discovery approach proposed in [7], called *KerDisc*[5]. To assess the similarity between the a service consumer request (user query) and the available WSs, the KerDisc approach first extracts the content from the WSDL documents followed by stop-word removal & stemming [7]. The constructed support-based semantic kernel in the training phase is then used to find the similarity between WSDL documents and a query when the query is provided. The topics of WSDL documents which are most related to the query topics are considered to be the most relevant. Based on the similarity computed using the support-based semantic kernel, the WSDLs are ranked and a list of appropriate Web services is returned to the service consumer.

Both SeqDisc and KerDisc have been validated using the data sets illustrated in Table 1. The quality measures have been evaluated and results are reported in Fig. 5. The figure shows that, in general, SeqDisc is more effective than KerDisc. It achieves higher F-measure than the other approach across five domains. It is worth noting that the KerDisc approach indicates low quality across the Address and Email domains. This is due to the two domains have common content, which produces many false positive candidates. The large number of false candidates declines the approach precision. Compared to the results of SeqDisc using only the linguistic matcher shown in Fig. 4(b), our approach outperforms across the Address and DNA domains, while the KerDisc approach is better across the other domains. This reveals two interesting findings: (1) KerDisc can effectively locate the desired service among heterogeneous WSs, while it fails to discover the desired service among a set of homogeneous services. In contrast, our approach could effectively locate the desired service among either a set of homogeneous or a set of heterogenous services. (2) SeqDisc clarifies the importance of exploiting the structure matcher.

[5] We give the approach this name for easier reference.

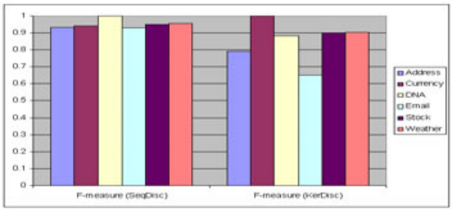


Fig. 5. Effectiveness comparison

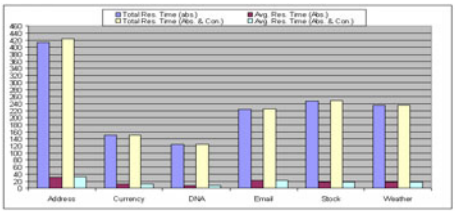


Fig. 6. Response time response

Efficiency Evaluation. From the response time point of view, Fig. 6 gives the response time that is required to complete the task at hand, including both pre-processing and similarity measuring phases. The reported time is computed as a total time and an average time. The total time is the time needed to locate desired Web services belonging to a certain category, while the average time is the time required to discover a Web service of the category. The figure also shows that the framework needs 124 seconds in order to identify all desired Web services in the DNA category, and it requires 7 seconds to discover one service in the category, while it needs 3.7 minutes to locate all services in the Email category. We also considered the response time and compared it to the response time of the first set (i.e, using only the abstract parts). The results are calculated and listed in Fig. 6. The figure shows that the response time required to locate the desired Web service using both abstract and concrete parts equals to the response time when only using abstract parts, or needs a few milliseconds more.

7 Conclusions

We introduced a new and flexible approach to assess the similarity between WSs, which can be used to support a more automated WS discovery framework. The approach makes use of the whole WSDL document specification and distinguishes between the concrete and abstract parts. The concrete parts from different Web services have the same hierarchal structure, hence we devised a level-based matching approach. The abstract parts have different structures, therefore, we developed a sequence-based schema matching approach to compute the similarity between them. We have conducted a set of experiments to validate our approach. Our experimental results have shown that our method is accurate and scale-well. However, we are still a long way from automatic service discovery. In our ongoing work, we plan to complete the service discovery framework exploiting more WSDL features, such as text values associated to each element through documentation.

Acknowledgements. This work is an extended version of the paper presented in [2]. The work of A. Algergawy is supported by the BMBF, grant 03FO2152. While, the work of N. Siegmund and V. Köppen is also funded by the BMBF, project 01IM08003C.

References

1. Al-Masri, E., Mahmoud, Q.H.: Qos-based discovery and ranking of web services. In: ICCCN 2007, pp. 529–534 (2007)
2. Algergawy, A., Schallehn, E., Saake, G.: Efficiently locating web services using a sequence-based schema matching approach. In: 11th ICEIS(1) (2009)
3. Algergawy, A., Schallehn, E., Saake, G.: Improving XML schema matching performance using prüfer sequences. DKE 68(8), 728–747 (2009)
4. Anadiotis, G., Kotoulas, S., Lausen, H., Siebes, R.: Massively scalableweb service discovery. In: AINA '09 (2009)
5. Atkinson, C., Bostan, P., Hummel, O., Stoll, D.: A practical approach to web service discovery and retrieval. In: ICWS 2007, pp. 241–248 (2007)
6. Avila-rosas, A., Moreau, L., Dialani, V., Miles, S., Liu, X.: Agents for the grid: A comparison with web services (part ii: Service discovery). In: AAMAS '02, pp. 52–56 (2002)
7. Bose, A., Nayak, R., Bruza, P.: Improving web service discovery by using semantic models. In: Bailey, J., Maier, D., Schewe, K.-D., Thalheim, B., Wang, X.S. (eds.) WISE 2008. LNCS, vol. 5175, pp. 366–380. Springer, Heidelberg (2008)
8. Cabral, L., Domingue, J., Motta, E., Payne, T.R., Hakimpour, F.: Approaches to semantic web services: an overview and comparisons. In: Bussler, C.J., Davies, J., Fensel, D., Studer, R. (eds.) ESWS 2004. LNCS, vol. 3053, pp. 225–239. Springer, Heidelberg (2004)
9. Carmel, D., Efraty, N., Landau, G.M., Maarek, Y.S., Mass, Y.: An extension of the vector space model for querying XML documents via XML fragments. SIGIR Forum 36(2) (2002)
10. Cohen, W., Ravikumar, P., Fienberg, S.: A comparison of string distance metrics for name-matching tasks. In: IIWeb, pp. 73–78 (2003)
11. Dong, X., Halevy, A., Madhavan, J., Nemes, E., Zhang, J.: Similarity search for web services. In: VLDB 2004, pp. 372–383 (2004)
12. Gal, A.: Managing uncertainty in schema matching with top-k schema mappings. Journal on Data Semantics 6, 90–114 (2006)
13. Hao, Y., Zhang, Y.: Web services discovery based on schema matching. In: ACSC 2007, pp. 107–113 (2007)
14. Köppen, V., Siegmund, N., Soffner, M., Saake, G.: An architecture for interoperability of embedded systems and virtual reality. IETE Tech. Rev. 26(5) (2009)
15. Ma, J., Zhang, Y., He, J.: Efficiently finding web services using a clustering semantic approach. In: CSSSIA 2008, p. 5 (2008)
16. Nayak, R., Lee, B.: Web service discovery with additional semantics and clustering. In: WI 2007, pp. 555–558 (2007)
17. Prufer, H.: Neuer beweis eines satzes uber permutationen. Archiv fur Mathematik und Physik 27, 142–144 (1918)
18. Siegmund, N., Pukall, M., Soffner, M., Köppen, V., Saake, G.: Using software product lines for runtime interoperability. In: RAM-SE, pp. 1–7 (2009)
19. Tatikonda, S., Parthasarathy, S., Goyder, M.: LCS-TRIM: Dynamic programming meets XML indexing and querying. In: VLDB '07, pp. 63–74 (2007)
20. Wang, Y., Stroulia, E.: Flexible interface matching for web-service discovery. In: WISE 2003, pp. 147–156 (2003)