# Flexible Interface Matching for Web-Service Discovery

Yiqiao Wang
*University of Alberta*
Yiqiao@cs.ualberta.ca

Eleni Stroulia
*University of Alberta*
Stroulia@cs.ualberta.ca

## Abstract

*The web-services stack of standards is designed to support the reuse and interoperation of software components on the web. A critical step, to that end, is service discovery, i.e., the identification of existing web services that can potentially be used in the context of a new web application. UDDI, the standard API for publishing web-services specifications, provides a simple browsing-by-business-category mechanism for developers to review and select published services. In our work, we have developed a flexible service discovery method, for identifying potentially useful services and assessing their relevance to the task at hand. Given a textual description of the desired service, a traditional information-retrieval method is used to identify the most similar service description files, and to order them according to their similarity. Next, given this set of likely candidates and a (potentially partial) specification of the desired service behavior, a structure-matching step further refines and assesses the quality of the candidate service set. In this paper, we describe and experimentally evaluate our web-service discovery process.*

## 1. Introduction

Faced with decreasing time-to-market and increasing requirement volatility, software-development processes are increasingly relying on reuse of existing software. Furthermore, the World Wide Web is increasingly being adopted as the medium of collaboration with partners and as a means of delivering information and services to consumers; thus, web-based applications constitute a substantial percentage of the currently developed applications.

The web-services set of standards is aimed at facilitating and improving the quality of component-based applications on the web. It consists of a set of related specifications, defining how reusable components should be specified (through the Web-Service Description Language – WSDL), how they should be advertised so that they can be discovered and reused (through the Universal Description, Discovery, and Integration API – UDDI), and how they should be invoked at run time (through the Simple Object Access Protocol API – SOAP).

A critical step in the process of reusing existing WSDL-specified components is the discovery of potentially relevant components. UDDI servers are essentially catalogs of published WSDL specifications of reusable components. These catalogs are organized according to categories of business activities. Service providers advertise services by adding their WSDL specifications to the appropriate UDDI directory category [1]. Through a well-defined API, software developers can browse the UDDI catalog by category.

This category-based service-discovery method is clearly insufficient. It is quite informal and relies, to a great extent, on the shared common understanding of publishers and consumers. It is the responsibility of the provider developer to publish the services in the appropriate UDDI category. The consumer developer must, in turn, browse the "right" category to discover the potentially relevant services. More importantly, these methods do not provide any support for selecting among competing alternative services that could potentially be reused.

In this paper, we discuss a flexible service discovery method, for identifying potentially useful services and estimating their relevance to the task at hand. This method is based on information retrieval and structure matching. Given a potentially partial specification of the desired service, all textual elements of the specification

Proceedings of the Fourth International Conference on Web Information Systems Engineering (WISE'03)
0-7695-1999-7/03 $17.00 © 2003 IEEE

are extracted and are compared against the textual elements of the available services, to identify the most similar service description files and to order them according to their similarity. Next, given this set of likely candidates, a structure-matching method further refines the candidate set and assesses its quality.

The intuition underlying this method is that an alternative means of querying UDDI servers is "query by example", i.e., by providing a (potentially partial) specification of the desired service. The consumer developer may define various aspects of the desired service, such as a description in natural language, the namespaces of its data types and the input/output structures of its operations, and the proposed method will return a set of candidate services with an estimate of their similarity to the provided example.

The remainder of the paper is organized as follows: section 2 discusses related work; section 3 explains in detail the design and implementation of our approach; section 4 discusses the results of our experimentation with the algorithm; section 5 outlines our plans for future work and concludes.

## 2. Related Work

The problem of web-service discovery is an instance of the more general problem of information retrieval and component discovery for which signature matching [14, 6] and specification matching [2,13] have been developed.

The Polylith system [6] proposed one of the earliest signature-matching methods for the purpose of interface adaptation and interoperation. The NIMBLE language in Polylith enabled programmers to specify coercion rules so that the parameters of the invoking module could be matched to the signature of the invoked module, including reordering, type mapping and parameter elimination. Zaremski and Wing [2,14] described signature matching as a means for retrieving functions and modules from a software library. Exact and relaxed function matching can be applied repetitively to match modules. Signature matching is an efficient means for component retrieval, for several reasons. Function signatures can be automatically generated from the function code. Furthermore, signature matching efficiently prunes down the functions and/or modules that do not match the query, so that more expensive and precise techniques can be used on the smaller set of remaining candidate components. However, signature matching considers only function types and ignores their behaviors; and two functions with the same signature can have completely opposite behaviors.

Specification matching aims at addressing this problem by comparing software components based on descriptions of their behaviors. Zaremski and Wing [13] extended their signature-matching work with a specification-matching scheme: two components match if their signatures and specifications match. Specification matching considers semantic information (behavior) about a component. However, there is no guarantee that specifications provided by the programmers correctly and completely reflect the component's behavior. Moreover, it is hard to motivate programmers to provide a formal specification for each component they write.

WSDL, the Web-Services Definition Language, is an XML-based interface-definition language. It describes "services" at a high level of abstraction, as a set of operations implemented by a set of messages involving a given set of data types [1]. WSDL [1] specifications of service-providing components are published in UDDI registries. UDDI [9] is designed as an online marketplace providing a standardized format for general business discovery. Developers can browse and query a UDDI registry using the UDDI API to identify businesses that offer services in a particular business category and/or services that are provided by a certain service provider.

WSDL service specifications do not include semantics. On the other hand, DAML-S [3] is a formal language that supports the specification of semantic information in RDF format. As part of the "semantic web" effort, it is intended as the means for specifying domain-specific semantics of ontologies. An extension of DAML-S supports service specification, including behavioral specifications of their operations; as a result, it enables discovery through specification matching, such as the method proposed in LARKS [7].

Traditional information retrieval methods include full text retrieval, signature files, inversion, and vector model and clustering [4]. Full text retrieval methods search all documents for the specified string. These methods are most straightforward and require minimum effort to maintain, but the response time is bad when files are large [4]. In signature file approach, each document generates a bit string as its signature, and searches are done on these signature files. The advantages are that it is easy to implement and it is robust. However, its response time is bad when used on large files [4]. Inversion methods represent documents by a list of alphabetically sorted keywords. Fast retrieval can be achieved using these methods, but storage overhead and cost of maintaining go up [4]. Clustering methods use vector model to generate document clusters and group similar documents together. Each document is represented as a t-dimensional vector

where t is the number of distinct words in the document. Representing documents and queries as vectors allows for relevance feedback, and increase effectiveness of the search [4].

## 3. The Web-service Discovery Method

Our service-discovery method is aimed at enabling service discovery, in a more precise and automatic manner than browsing, utilizing the information actually provided by WSDL. To that end, it integrates information retrieval and structure matching algorithms. The method assumes as input a (potentially partial) specification of the desired WSDL specification and a set of WSDL specifications of available services, such as the services advertised in UDDI. The vector-space model, an information retrieval method, is used to identify most similar service description files, and to order them according to their similarity. Given this ranked list of candidate web services, a structure-matching algorithm further refines and assesses the quality of the candidate service set.

### 3.1. Information Retrieval Using the Vector Space Model

In the vector-space model, documents and queries are represented as T-dimensional vectors, where T is the total number of distinct words in the document collection after the preprocessing step. Preprocessing includes eliminating stop words (very commonly used words) and conflating related words to a common word stem. Each term in the vector is assigned a weight that reflects the importance of a word in the document. This value is proportional to the frequency a word appears in a document and inversely proportional to number of documents that contain this word [10,8]. A common term importance indicator is *tf-idf* weighting [8] where the importance of a word *i* in document *j* is:

$w_{ij} = tf_{ij}\, idf_i = tf_{ij} \log_2 (N/df_i)$, where $tf_{ij}$ is normalized term frequency across the document, and $idf_i$ is the inverse document frequency of term *i*. *N* is total number of the documents in the collection, and *Log* is used to dampen the effect relative to *tf* [8].

The WSDL syntax allows textual descriptions for the service, its types and its operations, grouped under <documentation> tags [1]. Thus, given a natural language description of the desired service, it is possible to employ the vector-space model to retrieve these published WSDL services that are most similar to the input description on the respective vectors. A higher score indicates a closer similarity between the source and target specifications.

### 3.2. WSDL Structure Matching

Because it is based on XML syntax, WSDL specifications are hierarchical. At the lowest level of the hierarchy, lie the data types, which themselves are defined in XML and are hierarchical; one layer above the messages are defined, whose structures depend on the defined data types; the next layer specifies the service operations, which are composed of messages. Finally, the whole service is defined as the composition of its data types and operations. Service developers exploit the hierarchical nature of XML data types to capture the internal structure of the entities they model; similarly, they decompose the functionality delivered by the service in terms of messages and operations. Therefore, the hypothesis underlying our WSDL structure matching algorithm is that, if two services are conceptually similar they are more likely to also be structurally similar than otherwise. As a result, we have developed a heuristic, domain-specific tree-edit distance algorithm to assess the structural similarity of WSDL specifications. The tree-edit distance algorithm [5] calculates the similarity of two tree structures as the minimum number of node modifications required to match them. [5] describes such an algorithm for comparing arbitrary XML documents. Our algorithm assumes that the two trees being compared are WSDL specifications and relies on the structure of the WSDL schema to simplify the tree comparison.

The comparison of two WSDL files is a multi-step process: it involves the comparison of the operations' set offered by the services, which is based on the comparison of the structures of the operations' input and output messages, which, in turn, is based on the comparison of the data types of the objects communicated by these messages.

The overall process starts by comparing the data types involved in the two WSDL specifications, as described in Section 3.2.2. The result of this step is a matrix assessing the matching scores, i.e., the degree of similarity, of all pair-wise combinations of source and target data types. It is interesting to note here that the data types of web services specified in WSDL are XML elements; as such, they can potentially be highly complex structures.

The next step in the process is the matching of the service messages, described in Section 3.2.3. The result of this step is a matrix assessing the matching scores of all pair-wise combinations of source and target messages.

The degree to which two messages are similar is decided on the basis of how similar their parameter lists are, in terms of the data types they contain and their organization.

The third step of the process is the matching of the service operations, described in Section 3.2.4. The result of this step is a matrix assessing the matching score of all pair-wise combinations of source and target operations. The degree to which two operations are similar is decided on the basis of how similar their input and output messages are, which has already been assessed in the previous level.

Finally, the overall score for how well the two services match is computed by identifying the pair-wise correspondence of their operations that maximizes the sum total of the matching scores of the individual pairs.

After all target WSDL specifications have been matched against the source WSDL specification, they are ordered according to their "overall matching scores": a higher score indicates a closer similarity between the target and source specifications. For each target specification, the algorithm also returns the mapping of its data types and operations to the corresponding data types and operations of the source specification as an "explanation" of its assigned match score. The details of the process are described in the rest of this section.

**3.2.1 An Example Scenario.** Let us now introduce a specific example of matching two web services. The source web service contains one operation, getData, takes a string as input and returns a complex data type named POType which is a product order. The target web service contains one operation, getProduct, takes an integer as input and returns the complex data type MyProduct as output.

Figure 1 shows the WSDL specifications of the two services discussed above. In WSDL, all operations are grouped under the <portType> tag, with each unique operation specified under an <operation> tag. In Figure 1, operations are highlighted in boxes with bold, dashed lines. Operations' request and response messages are specified as input and output messages under <message> tags (highlighted in boxes with bold, solid lines). Complex data types are grouped under the <types> tag, with each data type defined under a <complexType> tag. Complex data type definitions are highlighted using solid-line boxes. POType is defined in Figure1a for the source web service and it consists of an ID(String), a name(String), and another complex data structure Items, which contains a product name (String) and quantity purchased (int). MyProduct, defined in Figure 1b for the target web service, contains an ID(int), a name(String), a price(float), and part (ProductParts), a complex data type that contains a name(String).

**3.2.2 Matching Data Types.** The basis of service, operation and message matching is the matching score of individual data types. The process of matching data types is guided by the following properties:

**Property 1.** Preference is given to the matches between data types with the same grouping organization (style) of their elements.

According to the WSDL syntax, the elements of a complex data type are organized according to the following "grouping styles": <all>, <choice>, or <sequence> [1]. If two data types have the same internal grouping style, their matching score is increased by a bonus score. In addition, if both complex types are <sequence>s, the order of their elements is not mixed during the match. For the other two grouping styles, i.e., <all> and <choice>, the ordering of elements within a complex data type is not important.

**Property 2**. If two data types have the same name and they are imported from the same namespace, they are identical. An exhaustive matching is thus unnecessary. Instead, the procedure matchIdenticalTypes assesses the similarity score between two identical data types using only the structural information of the data types since all elements of the two data types are identical. This score is a function of number of elements grouped at each level of structure times MAXSCORE, the score assigned to identical or compatible data types.

Let us now discuss the algorithm matchDataTypes, shown in Figure 2. This procedure identifies all possible matches between two lists of data types in accordance with the properties described above, and returns the data-type correspondence that maximizes the overall matching score between these two lists.

As can been seen in Figure 2, the algorithm takes as input two lists of data types: sourceList, which contains m data types, and targetList, which contains n data types. Using these two lists, it constructs an m⊗n matrix, whose rows correspond to the source data types and columns correspond to the target data types (line 1). Each cell in the matrix is eventually filled with a value that indicates the matching score between the two data types corresponding to the row and the column of the cell.

For each two compared data types from the source list and the target list (lines 2 to 5), if they are both primitive data types, the procedure matchPrimitiveTypes is invoked to look up a table that contains all primitive data types and their match scores (Lines 6-7) and the score is stored

in the corresponding cell of the matrix. Two primitive data types can be either compatible (MAXSCORE, defined as 10, is assigned), semi-compatible (MAXSCORE/2 is assigned), or incompatible (0 is assigned). Examples of compatible data types include identical data types and data types that can be adapted to each other at little cost such as long and float, etc. Examples of semi-compatible data types, i.e., types that can be adapted to each other at some cost such as int and float, etc. Please note that the idea of matching data types that are "semi-compatible" to avoid being too strict is similar to the ideas of "relaxed matching", "specialized/generalized matching", and "subtype matching" in related signature matching work [14, 6].

```
int matchDataTypes (sourceList(m), targetList(n)) {
(1) matrix = construct a m⊗n matrix;
//exhaustive matching
(2) for (int i=0; i<m; i++) {
(3)     for (int j=0; j<n; j++) {
(4)         sourceType = sourceList(i)
(5)         targetType = targetList(j)
(6)          if (both sourceType and targetType are
                 primitive)
(7)              matrix[i][j] = matchPrimitiveTypes
                         (sourceType, targetType);
(8)          if (both sourceType and targetType share
                 the  same name and namespace)
(9)              matrix[i][j] =
      matchIdenticalTypes(sourceType, targetType);
(10)         if (either sourceType or targetType is
                 complex) {
(11)             newSourceList =
           getCompositeDataElements(sourceType);
(12)             newTargetList =
           getCompositeDataElements(targetType);
(13)             matrix[i,j] =
      matchDataTypes (newBaseList, newTargetList)
     + organizationBonus(sourceType, targetType);
(14)        } } }
(15)       return the matches with the maximum score; }
```

**Figure 2. Matching Two Lists of Data Types**

Lines 8 and 9 state that if two complex structures are identical because they have the same name in the same namespace, procedure matchIdenticalTypes is called to assess their match, as described in Property 2.

If one (or both) of the data types being compared is (are) complex, then the procedure getCompositeDataElements collects all elements of the complex data structure(s) to form new, simpler data-type list(s) (as shown in lines 10-12) to be further matched recursively. The matching score of the original data types compared is the highest matching score of their elements plus a bonus if the two complex data structures have the same grouping style as discussed in Property 1 (line 13). A new matrix is created for each new mapping between two non-primitive data types. After a matrix is filled, the algorithm forms all possible matches between the two lists represented by the matrix and returns the highest matching score between two lists of data types (Line 15). are calculated in similar manners.

Let us now apply the algorithms discussed above to match the complex data types POType and MyProduct of the two WSDL specifications shown in Figures 1a and 1b. Because both data types are complex, we need to recursively match all the elements of the two complex structures to decide on their similarity scores instead of doing a simple table look-up, thus a 3⊗4 matrix is constructed (Table 1). We use the notation ?→ MatchScore to indicate this process; the question mark indicates that a match score is currently unknown and it will eventually be replaced by MatchScore that will be obtained from further recursive calculations. POType matches to MyProduct with a score of 45. We will now explain how this matching process is performed.

**Table 1. Matching POType and MyProduct**

| MyProduct<br>POType | Id:<br>Int | Name:<br>str | Price:<br>float | Part:<br>Product<br>Parts <all> |
|---|---|---|---|---|
| Id:str | 5 | 10 | 5 | ?→ 10 |
| Name:str | 5 | 10 | 5 | ?→ 10 |
| Item <all> | ?→ 10 | ?→ 10 | ?→ 5 | ?→ 20<br>(10+bonus) |

Table 1 shows how complex data structures POType and MyProduct are matched. Primitive data types are mapped by a simple table look up; for simplicity reasons we will show only how the complex structure Item contained in POType is mapped to ProductParts contained in MyProduct (Table 2).

In Table 2, the best match between Item and Part results by matching Product:String and name:String with a score of 10. Because both structures have an <all> grouping style, a bonus of 10 is added to the match score and thus Item maps to Part with a score of 20. The bottom-right cell of Table 1, which corresponds to this match, now has the value of ?→ 20. Other cells in Table 1 are filled in the same manner.

```
<definitions>
<types>
 <schema .... >
<complexType name="POType">
<all>
    <element name="id" type="string"/>
    <element name="name" type="string"/>
    <element name="items">
    <complexType>
        <all>
          <element name="item" type="tns:Item"
           minOccurs="0" maxOccurs="unbounded"/>
        </all>
    </complexType>
    </element>
</all>
</complexType>

<complexType name="Item">
<all>
     <element name="quantity" type="int"/>
     <element name="product" type="string"/>
</all>
</complexType>
 </schema>
</types>
    <message name="getDataRequest">
       <part name="id" type="string"/>
    </message>

    <message name="getDataResponse">
       <part name="data" type="POType"/>
    </message>

    <portType name="Data_PortType">
       <operation name="getData">
            <input message="tns:getDataRequest"/>
            <output message="getDataResponse"/>
       </operation>
    </portType>
</definitions>
```

**(a) WSDL Specification of Operation getData**

```
<definitions>
<types>
 <schema .... >
<complexType name="MyProduct">
<all>
    <element name="id" type= “int”/>
    <element name="name" type="string"/>
    <element name= “price" type="float"/>
    <element name="part" type="productParts"/>
</all>
</complexType>

<complexType name="productParts">
<all>
    <element name="name" type="string”/>
</all>
</complexType>

 </schema>
</types>
 <message name="getProductRequest">
     <part name="id" type="int"/>
 </message>

 <message name="getProductResponse">
      <part name="product" type="MyProduct”/>
 </message>

 <portType name="Product_PortType">
     <operation name="getProduct">
       <input message="getProductRequest"/>
       <output message="getProductResponse"/>
     </operation>
 </portType>
</definitions>
```

**(b) WSDL Specification of Operation getProduct**

**Figure 1. Two WSDL Specifications**

**Table 2. Matching Item and Part**

| Part<br>Item | Name: str |
|---|---|
| Quantity: int | 5 |
| Product: str | 10 |

**Table 3. Matching POType and MyProduct**

| Matches | Elements of POType | Elements of MyProduct | Score |
|---|---|---|---|
| Match1<br>Score: 35 | Id:str | Id:int | 5 |
| | Name:Str | Name:Str | 10 |
| | Item | ProductParts | 20 |
| Match2<br>Score: 35 | Id:Str | Name:Str | 10 |
| | Name:Str | Id:int | 5 |
| | Item | ProductParts | 20 |

The matching score between POType and MyProduct can be determined as soon as all the cells in Table 1 are filled. We form all possible pair-wise combinations of POType and MyProduct elements. The best match is the combination with the highest cumulative score. Table 3 lists the two best matches with scores of 35 between POType and MyProduct. Elements of POType in column 2 are matched to elements of MyProduct in column 3, and the scores in column 4 are their corresponding match scores according to Table 1. Finally because POType and MyProduct have the same grouping style, a bonus score of 10 is added; as a result, POType matches to MyProduct with a match score of 45.

**3.2.3 Matching Message Structures.** After evaluating the data-type matching scores, the structures of the source-service messages against the target-service messages are matched. Clearly, given a source and a target message, there are many possible correspondences between their parameter lists. The objective of this step, then, is to identify the parameter correspondence that maximizes the sum of their individual data-type matching scores. Figure 3 shows the algorithm matchMessages that takes as input two messages and returns as output their matching score.

Let us consider the two services listed in Figure 1 again. Both operations from the source and the target have two messages, so we first construct a 2⊗2 matrix (Table 4). Message getDataResponse maps to message getProductResponse with a score of 45 since POType maps to MyProduct with a score of 45 and they are the only data types in the lists. All cell values of the matrix are calculated in similar manners.

```
int matchMessages (message1, message2) {
    list1 = list of data types associated to message1;
    list2 = list of data types associated to message2;
    score = matchDataTypes (list1, list2)
    return score;  }
```

**Figure 3. Matching Two Messages**

**Table 4. Matching Messages of Web Services getData and getProduct**

| GetProduct<br>GetData | GetProduct<br>Request | GetProduct<br>Response |
|---|---|---|
| GetDataRequest | 5 | 10 |
| GetDataResponse | 10 | 45 |

**3.2.4 Matching Operations.** The process of matching operations is similar to the process of matching messages. The matching score between two operations is the sum of the matching scores of their input and output messages, as shown in the algorithm matchOperations of Figure 4.

Since matching between messages has already been performed in the previous stage, at the operation matching level we only need a simple matrix look-up. The internal organization of operations is again respected: the input messages of the source operation can only be mapped to the input messages of the target operation. And if the two input messages of two operations are mapped, their output messages have to be mapped correspondingly as well.

```
int matchOperations (o1, o2) {
  score = lookupMessageMatrix(o1input1, o2input) +
          lookupMessageMatrix(o1output, o2output)
  return score;  }
```

**Figure 4. Matching Two Operations**

Let us continue with our running example. We only have one operation in both source and target WSDL specifications. So the matchOperations procedure is invoked to match the source operation getData with the target operation getProduct getDataRequest maps to getProductRequest with a score of 5, and getDataResponse maps to getProductResponse with a score of 45 as can be seen from Table 4. Thus the two operations match with a score of 50.

**3.2.5 Matching Web Services.** Web services are specified in term of the operations they define. The algorithm matchWebServices is used to match all operations between the source and target WSDL specifications in a pair-wise fashion to identify the best source-target operation correspondence (Figure 5).

An m⊗n matrix is constructed, where m is the number of the operations specified in the source WSDL, and n is the number of operations in the target WSDL. The procedure matchOperations, in Figure 4, is invoked to calculate the match score between a single pair of operations. Then the algorithm explores all possible combinations of pair-wise matched operations and returns those with the highest match score, calculated as the sum of all individual pair-wise scores.

In the case of our example, since the source and target web services have only one operation each, the service matching score is the same as the operation matching score computed in the previous phase, i.e., 50.

```
int matchWebServices(service1, service2) {
  m=number of operations in service1;
  n=number of operations in sevice2;
  operationMatrix = construct m ⊗ n matrix;
  for (int i=0; i<m; i++)
    for (int j=0; j<n; j++)

operationMatrix[i][j]=matchOperations(list1[i],list2[j]);
return the matches with the maximum score; }
```

**Figure 5. Matching Two Web Services**

## 4. Evaluation

To evaluate our service-discovery method as a whole and the effectiveness of its constituent elements, we had to obtain families of related specifications in order to evaluate the degree to which our algorithm can distinguish among them. We found such a collection published by XMethods [12]. In the XMethods collection, we identified nineteen service descriptions from five categories: currency rate converter (three services), email address verifier (three services), stock quote finder (four services), weather information finder (four services), and DNA information searcher (five services). All the experiments below share the same design: each of the XMethods service was used as the basis for the desired service; different aspects of this desired service were matched then against the complete set to identify the best target service.

In this section, we report on three sets of experiments: service discovery with information retrieval only, with structure matching only, and with the two methods combined.

### 4.1. Experiments Using Information Retrieval

In the experiments that use only information retrieval method, we match text descriptions of each service from each category (requests) against the text descriptions of all other services from all categories (candidates). Note that although that "description" is not a mandatory element of WSDL specifications, in our experience all XMethods services had it. The similarity score between a given web service *S* and service requests from a given category *Q* is the average of similarity scores calculated between *S* and each request from category *Q*. Web services with similarity scores greater than zero are deemed relevant to service requests and only these services are ranked and returned to the users.

We evaluate the effectiveness of our retrieval methods by calculating their precision and recall. "Precision is the proportion of retrieved documents that are relevant, and recall is the proportion of relevant documents that are retrieved" [10]. Precision and recall for each test collection from each category of service requests are calculated and are listed in Table 5. The information retrieval method achieves a precision of 51% at 95% recall on average on this set of experiments.

**Table 5. Vector-Space Model Information Retrieval on the XMethods Collection**

| Requests | Precision | Recall |
|---|---|---|
| Currency Rate Converter | 42% | 100% |
| DNA info Searcher | 83% | 100% |
| Email Address Verifier | 30% | 100% |
| Stock Quote Finder | 50% | 100% |
| Weather Info Finder | 50% | 75% |

### 4.2. Experiments Using Structure Matching

Experiments with structure matching only were conducted in a similar manner: we match structure of each service from each category (requests) against structures of all other services from all categories (candidates). Averages are calculated between service requests from each category and all candidate service descriptions. Web services are ranked according to their similarity scores to the requests, and top 70% of web services on the lists are considered to be relevant to the requests and are returned to the users. We assume that if a web service ranks in the bottom of the list, chances that this web service is relevant to the request are low. Precision and recall are calculated for each set of queries (listed in Table 6), and on average, structure matching achieves a precision of 20% at 72% recall. The precision is rather low in this set of experiments because some related services have substantially different structures and some irrelevant services can often have higher matching scores because they have many spurious substructures that happen to match the query structure.

**Table 6. Structure Matching on the XMethods Collection**

| Requests | Precision | Recall |
|---|---|---|
| Currency Rate Converter | 14% | 67% |
| DNA info Searcher | 36% | 100% |
| Email Address Verifier | 14% | 67% |
| Stock Quote Finder | 28% | 100% |
| Weather Info Finder | 7% | 25% |

**Table 7. Combined Service Discovery Method on the XMethods Collection**

| Requests | Precision | Recall |
|---|---|---|
| Currency Rate Converter | 50% | 100% |
| DNA info Searcher | 100% | 100% |
| Email Address Verifier | 37.5% | 100% |
| Stock Quote Finder | 80% | 100% |
| Weather Info Finder | 40% | 50% |

### 4.3. Experiments Using Information Retrieval and Structure Matching

Finally, we have conducted a third set of experiments combining both information retrieval and structure matching. The information-retrieval step was used first on all services as described in section 4.1 to obtain a list of web services that have greater than zero similarities to the queries. Then, structure matching is applied to the pruned list of candidates as described in section 4.2. Similarity score between a web service *S* and a set of queries from a category *Q* is

$$Sim_{combined}(Q,S) = Sim_{IR}(Q,S) * 25 + Sim_{matcher}(Q,S)$$

where $Sim_{IR}$ (S,Q) and $Sim_{matcher}$ (S,Q) are average similarity scores calculated by the information-retrieval method and structure-matching method respectively. The similarity scores calculated by vector space model are normalized by a ratio of 25. The candidate services are matched and re-ranked, and top 80% of the services in the list are considered to be relevant and returned. We choose to return top 80% of the services (as opposed to 70% that we used before) because in this second round of pruning, setting the threshold too high results in a higher risk of eliminating relevant web services.

Precision and recall for each test collection are calculated and are listed in Table 7. On average, this retrieval system that uses both traditional IR techniques and WSDLMatcher achieves a precision of 61.5% at 90% recall. Compared to performance of experiments that use only information retrieval techniques, precision is increased by 10.5% from 51% and recall dropped by 5% from 95%. Both precision and recall improved significantly compared to the results obtained with structure matching only.

## 5. Conclusions and Future Work

In this paper, we described a web service discovery application that combines traditional information retrieval techniques with a structure-matching algorithm leveraging the structure of the XML-based service specification in WSDL. Currently developers can only browse UDDI registries and query the advertised services by business category. This is a very blunt and imprecise service-discovery mechanism and relies on the common-sense understanding of producers and consumers regarding business types. The semantic web effort, on the other hand, aims at formalizing the definition of these business types in order to support full-fledged type checking through description logic, at the cost of requiring complete formal specifications of web services and the underlying ontology of their data types. The intuition behind our work is that semantic information is already implicitly captured in the current WSDL specification in two forms: (a) in the structure of the data types, and (b) in the natural-language semantics of the WSDL descriptions and the chosen identifiers. Therefore structure matching and lexical information retrieval could be useful in assessing the similarity between a desired and an available web service.

The structure-matching algorithm is inspired by traditional signature-matching methods for component retrieval. It is designed to calculate the similarity between the structure of a desired service and the structures of a set of advertised services. The algorithm respects the structural information of data types and is flexible enough to allow relaxed matching and matching between parameters that come in different orders in parameter lists. Combined with traditional information retrieval techniques, this structure matching method constitutes an

important extension to the UDDI API, because it enables a substantially more precise service-discovery process.

We have conducted various experiments to evaluate the effectiveness of our retrieval system. Our initial results are encouraging. In the future, we plan to extend this algorithm to exploit the full WSDL syntax. Currently, we are not considering some of the syntax WSDL offers such as minOccurs, maxOccurs that indicate minimum and maximum occurrences of data types, and some other attributes of element tags [1]. We also plan to include WordNet [11], an on-line lexical database for the English language, to further understand the semantics of web services described. Finally we are currently working on an approximate structure matching algorithm to reduce the computational cost of the process.

## 6. References:


[1] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. "Web Service Description Language (WSDL)". http://www.w3.org/TR/wsdl.

[2] I. Cho, J. McGregor, and L. Krause. "A protocol-based approach to specifying interoperability between objects". In Proceedings of TOOLS'26. Santa Barbara, California, August 1998, pp. 84-96. IEEE Press.

[3] The DARPA Agent Markup Language. http://www.daml.org/

[4] C. Faloutsos. and D.W.Oard., "A survey of Information Retrieval and Filtering Methods, University of Maryland". Technical Report CS-TR-3514, August 1995.

[5] M. Garofalakis, and A. Kumar. "Correlating XML Data Streams Using Tree-Edit Distance Embeddings". In Proceedings of ACM PODS'2003. San Diego, California, June 2003, pp. 143-154. ACM Press.

[6] J. Purtilo and J. M. Atlee. "Module Reuse by Interface Adaptation". Software - Practice and Experience, 21(6), June 1991, pp 539-556.

[7] K. Sycara, S. Widoff, M. Klusch and J. Lu. "LARKS: Dynamic Matchmaking Among Heterogeneous Software Agents in Cyberspace". Journal of Autonomous Agents and Multi-Agent Systems, Kluwer Academic.2002, pp. 173-203.

[8] G. Salton, A. Wong and C.S. Yang. "A vector-space model for information retrieval", In Journal of the American Society for Information Science, volume 18. November 1975, pp. 13-620. ACM Press.

[9] UDDI technical paper, http://www.uddi.org/pubs/Iru_UDDI_Technical_White_Paper.pdf

[10]E. Voorhees. "Using WordNet for Text Retrieval", in C. Fellbaum (ed.), *WordNet: An Electronic Lexical Database*, 1998, pp.285-303. The MIT Press.

[11]. WordNet. http://www.cogsci.princeton.edu/~wn/

[12] XMethods. http://www.xmethods.com/

[13] A. M. Zaremski and J. M. Wing. "Specification Matching of Software Components". ACM Transactions on Software Engineering and Methodology, 6(4). October 1997, pp. 333-369. ACM Press.

[14] A. M. Zaremski and J. M. Wing. "Signature Matching: a Tool for Using Software Libraries". ACM Transactions on Software Engineering and Methodology, 4(2): April 1995, pp. 146-170. ACM Press.