

International Journal of Cooperative Information Systems
© World Scientific Publishing Company

STRUCTURAL AND SEMANTIC MATCHING FOR ASSESSING WEB-SERVICE SIMILARITY

ELENI STROULIA

*Computing Science Department, University of Alberta, 221 Athabasca Hall,
Edmonton, AB, T6G 2E8, Canada*

YIQIAO WANG

*Computing Science Department, University of Alberta, 221 Athabasca Hall,
Edmonton, AB, T6G 2E8, Canada*

Received (Day Month Year)

Revised (Day Month Year)

The web-services stack of standards is designed to support the reuse and interoperation of software components on the web. A critical step in the process of developing applications based on web services is service discovery, i.e., the identification of existing web services that can potentially be used in the context of a new web application. Discovery through catalog-style browsing (such as supported currently by web-service registries) is clearly insufficient. To support programmatic service discovery, we have developed a suite of methods that assess the similarity between two WSDL (Web Service Description Language) specifications based on the structure of their data types and operations and the semantics of their natural language descriptions and identifiers. Given only a textual description of the desired service, a semantic information-retrieval method can be used to identify and order the most relevant WSDL specifications based on the similarity of the element descriptions of the available specifications with the query. If a (potentially partial) specification of the desired service behavior is also available, this set of likely candidates can be further refined by a semantic structure-matching step, assessing the structural similarity of the desired vs. the retrieved services and the semantic similarity of their identifiers. In this paper, we describe and experimentally evaluate our suite of service-similarity assessment methods.

Keywords: web services; web-service matching; web-service discovery

1. Introduction

The development of web-based applications in the service-oriented architecture style, as supported by the web-services stack of standards, relies on a set of related specifications, defining how reusable components should be specified (through the Web-Service Description Language ¹⁷), how they should be advertised so that they can be discovered and accessed (through the Universal Description, Discovery, and Integration API ¹²), and how they should be invoked at run time (through the Simple Object Access Protocol API ⁸).

A critical step in the process of reusing existing WSDL-specified services for building web-based applications is the discovery of potentially relevant services. UDDI servers are essentially catalogs of published WSDL specifications of available services. These catalogs are organized according to categories of business activities. Providers advertise services by adding their WSDL specifications to the appropriate UDDI directory category. Through a well-defined API, software developers can browse the UDDI catalog by category to discover existing services potentially relevant to the systems they are developing.

This category-based service-discovery method is clearly insufficient, because it relies on the shared common-sense understanding of the application domain by the developers who publish and consume the specified services. It is the responsibility of the provider to publish the services in the appropriate UDDI category. In turn, the consumer must browse the “right” categories to discover the potentially relevant services and to assess which of the discovered candidate services are more likely to be useful in their systems.

In this paper, we discuss a set of complementary methods for assessing the similarity of WSDL specifications, which can be used to support a more automated service-discovery process, by distinguishing among the potentially useful and the likely irrelevant services and by ordering the potentially useful ones according to their relevance to the developer’s query. To assess the similarity between two WSDL specifications, these methods utilize, on one hand, the semantics of the identifiers and the natural-language descriptions of WSDL specifications and, on the other, the structure of their operations, messages and types. Given their different knowledge requirements, these methods are applicable under different conditions. Given only a textual description of the desired service, a semantic information-retrieval method can be used to identify and order the most similar of the available service specifications. This step assesses the similarity of the provided desired-service description with the available services, based on the natural-language semantics of the services descriptions using ¹⁶. If a (potentially partial) specification of the desired service behavior is also available, this set of likely candidates can be further refined by a semantic structure-matching step assessing the structure and semantic similarity of the desired vs. the retrieved services. Intuitively, these two types of queries are likely in different stages of software development. During the initial analysis phase, the developer may choose to explore what, if any, available components exist that may cover some of the functionalities desired of the new system. In this scenario, the developer is likely to have only a general natural-language description of the potentially useful services. During the subsequent design phase, the developer may already have designed stubs of the new system modules and their functionalities. Then, she can use such a stub as an example specification of the desired service, with which to query the UDDI repository.

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 experimental evaluation; finally, section 5

summarizes the novel contributions of this work and outlines some open issues it has raised.

2. Related Research

The problem of web-service discovery is analogous to the problems of component retrieval and information retrieval. On one hand, a WSDL specification is the specification of a “software component” including a specification of its interface signature and a specification of where the actual implementation exists and how it can be used. On the other, a WSDL specification usually includes a set of natural-language descriptions of the service and its elements. This section reviews the relations of our work with related research in these two areas as well as some new research on understanding and reasoning about web-service specifications.

2.1. Information retrieval

Traditional information-retrieval methods examine the words or phrases that documents in a collection have in common, in order to assess their similarity. The objective of such methods is to retrieve documents pertinent to a user’s queries or to organize the documents of a collection in thematically coherent clusters. The basic intuition underlying these methods is that the more words two documents share in common the more similar they are thematically.

This intuition is manifest in the vector-space model assumed by the majority of information-retrieval methods. In this model, documents are represented as t -dimensional vectors, where each element of the vector corresponds to one of the distinct words contained in the document. The vectors are usually constructed after preprocessing that eliminates stop words (i.e., commonly used words that are unlikely to be characteristic of any document’s content) and stems the documents’ words so that related words with a common stem are considered as a single word. Each term in the vector is assigned a weight that reflects the importance of this term in the document. The weight value is proportional to the frequency with which the term appears in the document represented by the vector and inversely proportional to the number of documents that contain this term¹⁰. A common term importance indicator is the term-frequency inverse-document-frequency $tf - idf$ ranking⁹, according to which, the importance of a word i in document j is

$$w_{ij} = tf_{ij} * idf_i = tf_{ij} * \log_2\left(\frac{N}{df_i}\right) \quad (2.1)$$

In the above equation, tf_{ij} is the frequency of term i in document j ; idf_i is the inverse document frequency of term i ; N is total number of documents in the collection; and df_i is number of documents containing the term i .

Queries are also represented as vectors. Similarity between a document vector,

4 *Eleni Stroulia, Yiqiao Wang*

d , and a query vector, q , can then be computed as the vector inner product:

$$Sim(d, q) = d \bullet q = \sum_{i=1}^t w_{id} \bullet w_{iq} \quad (2.2)$$

where w_{id} and w_{iq} are the weight of term i . A higher similarity score indicates a closer similarity between the query and retrieved documents.

The major shortcoming of the vector-space model methods arises from the fact that each word does not have a unique, unambiguous meaning - homonyms have the same spelling but different meanings - and that many words have synonyms, i.e., different words that have the same meaning. As a result two documents may have two similar vector representations and still contain very diverse subject matter.

WordNet¹⁶ is an on-line database, inspired by current psycholinguistic theories of human lexical memory. It organizes English words (nouns, verbs, adjectives and adverbs) into sets of synonyms corresponding to lexical concepts. Relationships between concepts, such as hyponyms (i.e., more specific terms) and hypernyms (i.e., more general terms), are represented as semantic pointers linking between the related concepts^{4,3}. WordNet has been used in numerous natural-language processing applications, including information retrieval, with the objective to circumvent the problems discussed above and to ameliorate traditional information-retrieval results^{2,7,13}.

2.2. Component matching

Component discovery, i.e., retrieving a software module from a library according to a desired requirements' specification is a problem that has received substantial attention in the software engineering literature. In general there are two types of methods to address this problem: signature matching^{6,18} and specification matching¹⁹.

Polyolith⁶ proposed one of the earliest signature-matching methods for interface adaptation and interoperation. Through its NIMBLE language, coercion rules could be specified 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 Wing¹⁸ also described exact and relaxed signature matching as a means for retrieving functions and modules from a software library. 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.

Signature matching considers only function types and ignores their behaviors; however, two functions with the same signature can have completely opposite behaviors (consider, for example, two functions for addition and subtraction of two integers). Specification matching aims at addressing this problem by comparing soft-

ware components based on formal descriptions of the semantics of their behaviors. Zaremski and Wing¹⁹ extended their signature-matching work with a specification-matching scheme. Specification-matching methods employ formal specifications of the components' behavior in terms of invariants, and pre- and post-conditions of their methods. Then various degrees of matching can be defined in terms of how well the desired behavior is covered by the available ones. For example, two behaviors may be an exact match, i.e., the exact same predicates may apply to the functions of both modules; or the available behavior may subsume the desired one, i.e., the pre-(post-)conditions of desired module functions are stronger(weaker) than those of the available module; or the module behaviors may be mismatched. Specification matching provides a precise assessment of the degree of matching between the behaviors of the desired and the available components. However, it requires the development of formal specifications to characterize the available components and as these specifications have to be developed independently from the component code, there is no guarantee that they correctly and completely reflect its behavior.

2.3. Reasoning about WSDL specifications

WSDL¹⁷, the Web-Services Definition Language, is an XML-based interface-definition language. It describes services as a set of operations implemented by a set of messages involving a well-defined set of data types. WSDL specifications of service-providing components are published in publicly accessible registries. UDDI¹² (Universal Description, Discovery and Integration) refers to a type of such public registries and the associated API for accessing them. The intention behind UDDI is to provide an online marketplace with a standardized format for general business and service discovery. Currently, in order to discover a useful service, the developer has to start with the category under which the desired service is most likely to be published or the name of the provider organization and browse through the retrieved services to recognize the useful one(s). Alternatively, if the developer knows for certain that the desired service exists and has its "t-model", i.e., a precise specification of the functionality implemented by the service, she can find all its implementations. The first approach is time consuming and error prone; it relies on the developer to sift through potentially many alternatives and to decide which of them is more likely to be useful. The latter approach has too strict knowledge requirements and is therefore of limited applicability. In a realistic scenario the developer may have an idea of the desired service functionality but will not know whether the service actually exists and will not have the exact t-model that it implements.

DAML-S¹ (the DARPA Agent Markup Language for Services) is a formal language that supports the specification of services, based on domain-specific ontologies specified in OWL⁵ (Ontology Web Language). As part of the "semantic web" effort, these languages are intended as the means for semantically specifying domain-specific ontologies and the functions and internal behaviors of the services that agents in these domains may provide. As a result, it enables discovery through spec-

ification matching, such as the method proposed in LARKS¹¹. If indeed services were specified in DAML-S instead of WSDL it would be possible to formally prove that the requirements of the desired service and a discovered service do not conflict. However, DAML and DAML-S are not yet widely adopted. Furthermore, the cost of formally specifying the provided and requested services in DAML-S is even higher than the functional specifications assumed by the component specification-matching techniques discussed above: the functional specifications assumed by the component-retrieval research are based on a mathematical ontology, agnostic of the application domain in the context of which the component was developed; they are therefore simpler than the domain-specific specifications in OWL and DAML-S.

The motivation driving our research is to provide a means for a lightweight semantic comparison of WSDL specifications. There is a need for a more automated service-discovery method, beyond UDDI browsing. At the same time, this method should not make knowledge assumptions beyond what is currently available as part of the web-services stack of standards. The similarity-assessment methods we developed exploit the syntactic structure of the WSDL specifications' data types, messages and operations and the natural-language semantics of their identifiers, comments and descriptions in order to support query-by-example service-discovery.

3. Web-service Similarity Assessment

The objective of the similarity-assessment method suite discussed in this paper is to support the programmatic discovery of relevant services. In the absence of formal specifications of the ontology of the data types of the available services and of the functional semantics of their operations, it is not possible to guarantee that a retrieved service can fulfill the requirements of the requester. WSDL does not provide for any semantic specifications. It is extendible and, in fact, the DAML-S¹ effort aims at extending WSDL with such semantic specifications. However, until such extensions become standards and actual services with such semantic specifications are published the issue of programmatically discovering relevant services among the multitude of published services makes the problem of service-similarity assessment extremely relevant.

The fundamental intuition underlying our research is that, although WSDL does not explicitly provide support for semantic specifications, it does contain information that can potentially be used to infer the semantics of the specified service. First, the WSDL specification contains several elements whose values are natural-language textual descriptions, explaining the data types and the operations of the service. Second, the internal structure of the data types, also represented in XML syntax, is designed to capture the domain-specific relations of the various data required and provided by the service. Finally, the identifiers chosen as names of the service data types, messages and operations are also usually meaningful. Depending on how much information the service requester can provide all or part of the above information of available published services can be used to assess which one of them

is potentially applicable to the consumer's request. For example, if the requester already has a partially designed software system, she may have a stub of the desired service; in that case, the structure of the desired service data types and operations - as specified in the stub - can be compared against the structures of the published services. If, on the other hand, the requester is still in the analysis stage and has only a high-level idea of the nature of the desired service, then she will most likely formulate a textual query; in this case, the published services' descriptions and identifiers can be used to assess the similarity of the published services with the request.

We have developed and experimented with four distinct methods for WSDL service similarity assessment. First, when only a natural-language description of the desired service is available, a classical vector-space model information-retrieval method can be used to retrieve services with similar descriptions^a in their WSDL specifications. At this stage, WordNet can also be used to elaborate the query and service documentation descriptions with synonym terms. Section 3.2 discusses both variants of information retrieval. When a (potentially) partial specification of the desired service data types, messages and operations is also available, the retrieved list of candidate services can be further refined by assessing the similarity of the internal structure of the requested service against the structures of the candidates (see section 3.3). Finally, the structure-matching similarity-assessment method has also been enhanced with WordNet, which is used to calculate the semantic distances between the identifiers of the WSDLs^{14,15} (see section 3.4).

3.1. *Example Scenario*

We will use the two example web services shown in Figure 1, `getData` and `getProduct`, to illustrate the matching processes of our similarity-assessment methods.

Let us assume that the service `getProduct` is advertised in the UDDI registry by its provider as having a "search product by id number" functionality. It implements the operation `getProductByNumber` that takes an identification `Number` as input, searches the back-end product database and returns as output the `product` with the provided id number. The returned product is an instance of `ProductType`; this is a composite data type consisting of a unique identification `Number` of type `String`, a natural-language `Description` of the product (of type `String`), the `Price` of the product (of type `float`), and a composite data type, `ProductParts`, which contains an element named `Part` (of type `String`) specifying names for different parts of a product.

Let us further assume that a potential consumer queries the service repository using the `getData` service as an example, looking for a service that "searches for

^aThe term "description" here refers to the collection of all the `<documentation>` elements in the WSDL specification and the identifiers of the WSDL data types, messages and operations. In section 4 we report on experiments with information retrieval applied to the former aspect of the description only and to the complete description as well.

8 *Eleni Stroulia, Yiqiao Wang*

```

<definitions>
  <types>
    <schema>
      <complexType name = DataType>
        <all>
          <element name = "Id" type = "String"/>
          <element name = "Category" type = "String"/>
          <element name = "details" type = "Item"/>
        </all>
      </complexType>
      <complexType name = "Item">
        <all>
          <element name = "Quantity" type = "int"/>
          <element name = "Component" type = "String"/>
        </all>
      </complexType>
    </schema>
  </types>
  <message name = getDataByIdRequest>
    <documentation> method takes in a String as ID</documentation>
    <part name = "Id" type = "String"/>
  </message>
  <message name = getDataByIdResponse>
    <documentation> method returns a product with specified ID number</documentation>
    <part name = "Data" type = DataType/>
  </message>
  <portType name = getData>
    <operation name = getDataById>
      <documentation>search data type with a unique Id</documentation>
      <input message = getDataByIdRequest />
      <output message = getDataByIdResponse />
    </operation>
  </portType>
  <service name = getData>
    <port name = getData binding = getData> </port>
  </service>
</definitions>

```

The requested
getData service

```

<definitions>
  <types>
    <schema>
      <complexType name = ProductType>
        <all>
          <element name = "Number" type = "int"/>
          <element name = "Description" type = "String"/>
          <element name = "Price" type = "float"/>
          <element name = "details" type = "ProductParts"/>
        </all>
      </complexType>
      <complexType name = "ProductParts">
        <all>
          <element name = "Part" type = "String"/>
        </all>
      </complexType>
    </schema>
  </types>
  <message name = getProductByNumberRequest>
    <documentation> this method takes a number for identification</documentation>
    <part name = "Number" type = "int"/>
  </message>
  <message name = getProductByNumberResponse>
    <documentation> returns a product type with the number </documentation>
    <part name = "Product" type = ProductType/>
  </message>
  <portType name = getProduct>
    <operation name = getProductByNumber>
      <documentation>search product by id number</documentation>
      <input message = getProductByNumberRequest />
      <output message = getProductByNumberResponse />
    </operation>
  </portType>
  <service name = getProduct>
    <port name = getProduct binding = getProduct> </port>
  </service>
</definitions>

```

The advertised
getProduct ser-
vice

Fig. 1. WSDL Specification of two web services: `getData` and `getProduct`

data with a unique id". The desired service, `getData`, should have an operation, `getDataById`, that takes an `Id` of type `String` as input and returns an instance of

the composite data type, **DataType**, as output. The **DataType** of the query service is a complex element consisting, in addition to the **Id**, of the **Category** to which the **DataType** belongs (of type String) and a composite data type **Item**, which in turn contains the elements **Quantity** (of type int), and **Component** (of type String).

Finally, both services have descriptions for their operations, and messages included under <documentation> tags under their corresponding declarations.

3.2. Vector-Space Model Information Retrieval and WordNet

The first two methods we developed exploit the WSDL textual descriptions for services, their types and operations, grouped under <documentation> tags. First, given a natural language description of the desired service, the query text is compared against the combined textual descriptions of the available services using the vector-space model of document-similarity assessment as described in section 2.1, with equation 2.2 (including stop-word removal and stemming preprocessing).

Second, to address the traditional vector-space model shortcoming of considering words at the lexical level only, ignoring semantic information regarding synonyms and homonyms, we also explored an extension of the traditional vector-space model with WordNet. The WordNet-powered vector-space information-retrieval method expands the query and the service-description texts with the synonyms (words of similar meanings), direct hypernyms (parents), hyponyms (children), and siblings (hyponyms of hypernyms) senses. This method constructs three sub-vectors for each document and query: stems of the original words in the document (i.e., the service description and the query), stems of words' synonyms for all word senses, and stems of words' direct hypernyms, hyponyms and siblings for all word senses. All word senses are included for all terms, thus bypassing the problem of word-sense disambiguation. Different weights are assigned to the different sub-vector matching scores. Similarity scores between the original document terms is considered to be the most important score, carrying a weight of 3. The similarity score between words' synonyms is considered to be the second important similarity factor with a weight of 2. Because a word has many hierarchically related words (hypernyms, hyponyms, and siblings) according to WordNet, and their semantic meanings are not always obviously close, similarity scores between words' family groups are given the least amount of weight which is 1. The overall matching score between a service description and a query is the average of their three sub-vector matching scores.

$$\begin{aligned} S_{total} &= \frac{S_{original} \bullet W_{original} + S_{synonyms} \bullet W_{synonyms} + S_{family} \bullet W_{family}}{3} \\ &= \frac{S_{original} \bullet 3 + S_{synonyms} \bullet 2 + S_{family} \bullet 1}{3} \end{aligned} \quad (3.3)$$

where $S_{original}$, $S_{synonyms}$, and S_{family} are the similarity scores between the original document terms (first sub-vector), the words' synonyms (second sub-vector), and words hypernyms, hyponyms, and siblings (third sub-vector) respectively. The $W_{original}$, $W_{synonyms}$, and W_{family} are weights assigned to $S_{original}$, $S_{synonyms}$, and S_{family} , which are 3, 2 and 1. A higher overall matching score indicates a closer similarity between the source and target specifications.

3.3. Structure Matching

Our structure-matching method of WSDL specifications is a natural extension of the signature-matching method for component retrieval. 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 communicated by these messages.

The overall process starts by comparing the data types involved in the two WSDL specifications, as described in section 3.3.1. The result of this step is a matrix representing 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 3.3.2. The result of this step is a matrix representing 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 3.3.3. The result of this step is a matrix representing 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. Figure 2 outlines the family of algorithms implementing the structure-matching method for WSDL similarity assessment.

3.3.1. Structure Matching of Data Types

The basis of service, operation and message matching is the matching of the individual data types. To assess the degree of similarity between two service data types, this method performs a domain-specific comparison of the “trees” corresponding to the XML syntax of these data types specifications. This comparison is based on the following three heuristics:

- **Heuristic 0:** Two simple data types are compared on the basis of their programming-language type.
- **Heuristic 1:** Complex data types are compared on the basis of their constituent elements and the XML grouping organization among them.
- **Heuristic 2:** Complex data types, imported from the same namespace, are considered identical if they have the same name.

Let us now discuss the algorithm *structureMatchDataTypes*, shown in Figure 2(a). This procedure identifies all possible matches between two lists of data types

```

int structureMatchDataTypes (sourceList(m), targetList(n))
(1)  matrix = construct a  $m \otimes n$  matrix
(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 types 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] = structureMatchDataTypes (newBaseList, newTargetList)
          + organizationBonus(sourceType, targetType);
(14)  return the matches with the maximum score;

```

(a) Algorithm structureMatchDataTypes for Matching Two Lists of Data Types

```

int structureMatchMessages ( $msg_1, msg_2$ )
  list1 = list of data types associated to  $msg_1$ ;
  list2 = list of data types associated to  $msg_2$ ;
  score = structureMatchDataTypes (list1, list2)
  return score;

```

(b) Algorithm structureMatchMessages for Matching Two Messages

```

int structureMatchOperations ( $op_1, op_2$ )
  score = structureMatchMessages ( $op_1\_input, op_2\_input$ ) +
         structureMatchMessages ( $op_1\_output, op_2\_output$ )
  return score

```

(c) Algorithm structureMatchOperations for Matching Two Operations

```

int structureMatchWebServices ( $service_1, service_2$ )
  m = number of operations in  $service_1$ 
  n = number of operations in  $service_2$ 
  operationMatrix = construct  $m \otimes n$  matrix
  for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++)
      operationMatrix[i][j] = structureMatchOperations(list1[i],list2[j])
  return the matches with the maximum score

```

(d) Algorithm structureMatchWebServices for Matching Two Web Services

Fig. 2. The structureMatch family of algorithms

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 be seen in Figure 2(a), 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 $m \times n$ 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 and target list (lines 2-5), if they are both primitive data types, procedure *matchPrimitiveTypes* is invoked to assess their similarity (Lines 6-7) and the score is stored in the corresponding cell of the matrix. We grouped all primitive data types into four data type categories, shown in Table 1). Two primitive data types can be either compatible if they belong in the same group, semi-compatible if they belong in different groups but can be coerced into each other, or incompatible otherwise. Compatible primitive data types (i.e., data types within the same group) are assigned a maximum matching score of 10, semi-compatible data types (i.e., boolean and integer groups, or any combination between integer, decimal and String groups) are assigned a matching score of 5 and non-compatible data types are assigned a matching score of 0. This idea of different degrees of type compatibility derives from the ideas of strict, relaxed, specialized-generalized and subtype matching in the signature-matching literature^{18,6}.

Table 1. Primitive Data Type Groups

Primitive Type Groups	Data types
Integer group	“byte”, “short”, “int”, “long”
Decimal number group	“float”, “double”, “decimal”
String group	“char”, “String”
Boolean group	“boolean”

If any of the data types being compared 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 an “organization bonus” if the two complex data structures have the same grouping style over their constituent elements, as discussed in heuristic 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 14).

Lines 8 and 9 address the case of complex structured types that are identical because they have the same name in the same namespace. In this case, the procedure

matchIdenticalTypes is invoked to assess the contribution of their match to the similarity score, according to equation 3.4. This procedure does not perform a comparison per se; it simply traverses the specification of the target data type in the original namespace and decides the similarity score as a function of the number of elements grouped at each level of the structure, multiplied by *MaxScore*, the score assigned to identical data types. Furthermore, for each level of the structure an organization bonus is added, according to heuristic 1.

$$Sim_{ident} = \sum_i \#of\ elements\ at\ level\ i * MaxScore + organization\ bonus \quad (3.4)$$

Let us now apply the algorithm discussed above to match the complex data types **DataType** and **ProductType** of the two WSDL specifications shown in Figure 1. Because both data types are complex, the algorithm needs to recursively match all the elements of the two complex structures to decide on their similarity scores, thus a 3×4 matrix is constructed (Table 2). We use the notation ? → to indicate this recursive process; the question mark indicates that a match score is currently unavailable and it will eventually be replaced by a match score obtained from further recursive calculations. The **DataType** structure matches to **ProductType** with a score of 45. Let us now see how this score is obtained through the *structureMatch-DataTypes* algorithm.

Table 2. Structure Matching **DataType** and **ProductType**

DataType	ProductType			
	Number: Int	Description: String	Price: float	ProductParts <all>
Id: String	5	10	5	? → 10
Category: String	5	10	5	? → 10
Item <all>	? → 10	? → 10	? → 5	? → 20 (10 + bonus)

Table 3. Structure Matching **Item** and **ProductParts**

Item	ProductParts
Quantity: int	Part: String
Component: String	5
	10

Table 2 shows how the complex data structures **DataType** and **ProductType** are matched. Primitive data types are compared by a simple table look up; if either data type being compared is composite, their match score cannot be calculated till their constituent elements are matched. For example, **DataType** → **Id** (String) and **ProductType** → **Number** (int) are semi-compatible primitive data types, and they are assigned a matching score of 5. **DataType** → **Item** and **ProductType** →

Table 4. Best Structure Matching Results of `DataType` and `ProductType`

Matches	Elements of <code>DataType</code>	Elements of <code>ProductType</code>	Score
Match1, Score: 35	Id:String	Number:int	5
	Category:String	Description:String	10
	Item	ProductParts	20
Match2, Score: 35	Id:String	Description:String	10
	Name:String	Number:int	5
	Item	ProductParts	20
Match3, Score: 35	Id:String	Description:String	10
	Category:String	Price: float	5
	Item	ProductParts	20
Match4, Score: 35	Id:String	Price: float	5
	Category:String	Description:String	10
	Item	ProductParts	20

`ProductParts` are both composite data types and the further matching of their elements is shown in Table 3.

Only one of the elements of `Item`, i.e., `Item` \rightarrow `Quantity` and `Item` \rightarrow `Component`, can be mapped to the `ProductParts` \rightarrow `Part`. The matching that yields the highest score, i.e., mapping the String `Item` \rightarrow `Component` to the String `ProductParts` \rightarrow `Part` with `MaxScore` (10), is preferred. Because the two composite data types have the same organization style, `<all>`, a bonus matching score of 10 is also applied to the score. Thus, `DataType` \rightarrow `Item` maps to `ProductType` \rightarrow `ProductParts` with a score of 20. The bottom-right cell of Table 2, which corresponds to this match, now has the value of ? \rightarrow 20.

The rest of the Table 2 cells are similarly filled and then the matching score between `DataType` and `ProductType` can be determined. The algorithm exhaustively forms all possible pair-wise combinations of `DataType` and `ProductType` elements. The best matches are the combinations with the highest cumulative scores. Table 4 lists the four best matches with scores of 35 between `DataType` and `ProductType`. Elements of `DataType` in column 2 are matched to elements of `ProductType` in column 3, and the scores in column 4 are their corresponding match scores according to Table 2. Finally `DataType` and `ProductType` have the same grouping style of `<all>`, and a bonus score of 10 is added; as a result, the query service `DataType` matches to the target service data type `ProductType` with a match score of 45.

3.3.2. Structure Matching of Messages

After evaluating the data-type matching scores, the structures of the query-service messages against the target-service messages are compared. 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 evaluate all pair-wise mappings resulting from all possible permutations of the messages' parameter lists and to identify the parameter correspondence that maximizes the sum of their individual data-type matching scores. Figure 2(b) shows the algorithm *structureMatchMessages*

that takes as input two messages and returns as output their structure matching score. The algorithm *structureMatchMessages* extracts the parameters lists associated with the two messages considered^b and invokes the *structureMatchDataTypes* algorithm discussed in the previous section to match the two lists of data types.

Let us continue with the running example listed in Figure 1 to illustrate how this step of the matching process works. Both source and target operations, `getDataById` and `getProductByNumber`, have a request and a response message. To compare the `getDataByIdRequest` and `getProductByNumberRequest` messages, the algorithm first extracts the data types associated to the messages - `Id` and `Number` - and then calls *structureMatchDataTypes* to match these parameters. The matching score between these two messages is the matching score of their data types, which is 5. Similarly, the matching score between the two response messages, `getDataByIdResponse` and `getProductByNumberResponse`, is the matching score of their associated parameters, `DataType` and `ProductType`, which is 45, as we have already seen in the previous section.

3.3.3. Structure Matching of Operations

The process of matching operations is based on the process of matching their request and response (and exception when applicable) messages. The matching score between two operations is the sum of the matching scores of their input and output messages. The algorithm *structureMatchOperations* is listed in Figure 2(c). The algorithm simply extracts the operations' input and output messages and calls algorithm *structureMatchMessages* to get the sum of input and output message matching scores and returns this score as the operations' matching score.

Returning to our example, we only have one operation in both services `getData` and `getProduct`. The *structureMatchOperations* procedure is invoked to match operations `getDataById` and `getProductByNumber`. `getDataByIdRequest` maps to `getProductByNumberRequest` with a score of 5, and `getDataByIdResponse` maps to `getProductByNumberResponse` with a score of 45 as described in previous section. The overall matching score between the two operations is the sum of the match scores of their input and output messages which adds up to be 50.

3.3.4. Structure Matching of Web Services

Web services define a set of operations. The algorithm *structureMatchWebServices* 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 is shown in Figure 2(d). An $m \times n$ matrix is constructed, where m is the number of the operations defined in the query WSDL, and n is the number of operations

^bNote that only messages playing the same role in the context of operations are compared; i.e., input messages of the source service are only compared against input messages of the target service.

in the target WSDL. The procedure *structureMatchOperations*, in Figure 2(c), is then invoked to assess the similarity 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 web services, both services `getData` and `getProduct` define only one operation `getDataById` and `getProductByNumber` respectively. In this case, the service matching score between services `getData` and `getProduct` is the matching score of their operations, which has been shown to be 50 in the previous section.

3.4. *Semantic Structure Matching*

The structure-matching algorithms of section 3.3 aim at optimizing the mapping of the XML syntactic structures of the two compared WSDL service specifications; the “quality function” that is being optimized is essentially the number of element pairs that have compatible or semi-compatible types, organized under the same XML grouping style. The semantic structure matching method extends the above intuition by taking into consideration the identifiers of the service elements, in addition to their programming-language types and syntactic relations. It uses WordNet to calculate the semantic distances between the identifiers of each pair of compared elements in the WSDL specification, i.e., of services, operations and data types, instead of evaluating their similarity based on their type compatibility. The intuition behind this similarity-assessment method is that the chosen names of the types, operations, and services usually reflect the semantics of the underlying capabilities of the service.

The overall process is similar to structure matching. It starts by comparing the data types’ identifiers involved in the two WSDL specifications, as described in section 3.3.1. The result of this step is a matrix assessing the matching scores of all pair-wise combinations of the query and target service data types. The next step in the process is the matching of the service operations, described in section 3.3.3. The result of this step is a matrix assessing the matching scores of all pair-wise combinations of source and target operations. The degree to which two operations are similar is decided on the semantic distance between operations’ names and how similar their parameter lists are, in terms of the identifiers they contain. Finally, the overall score for how well the two services match is computed by matching services’ names and by identifying the pair-wise correspondence of their operations that maximizes the sum total of the matching scores of the individual pairs. The details of the process are described in the remainder of this section.

3.4.1. *Semantic Structure Matching of Data Types*

The process of semantic structure matching of data types is similar to that of data-type structure matching. The only difference is that instead of evaluating the

similarity of two data types based on their type compatibility, this method assesses their similarity as the inverse of their “identifier semantic distance” according to WordNet. The internal organization of composite data types is respected as with the structure matching process, according to the three heuristics discussed in section 3.3.1.

```
double matchDocumentTerms (term1, term2)
    maxScore = 10;
    if (term1 is identical to term2) score = maxScore;
    else if (term1 and term2 are synonymous) score = 8
    else if (term1 and term2 have hierarchical relations) score = 6 / distance between them
    else score = 0
    return score
```

Fig. 3. Algorithm *matchDocumentTerms* for Matching Two Document Terms

Figure 3 lists the algorithm *matchDocumentTerms* that assesses the similarity between two document terms using WordNet. Intuitively, the term-similarity score is a function of the terms’ semantic distance in the WordNet hierarchy: terms that are “located” close to each other in the WordNet semantic hierarchy have similar meanings and therefore are assigned a higher similarity score than terms that are further apart in the WordNet hierarchy. More specifically, if two words are identical they are assigned a similarity score of 10^c. If they are synonymous (regardless of the words’ senses^d), their similarity score is 8. Otherwise, if two words are in a hierarchically semantic relation, i.e., they are hypernyms, hyponyms or siblings to each other, their similarity is inversely proportional to the shortest path in the WordNet hierarchy linking them semantically. The identifier-similarity score between two such terms is calculated by dividing 6 by their semantic distance. It is important to note here that, we assume that programmers follow Java-style naming conventions and use meaningful names for methods and data types. Under this assumption, all identifiers and names are broken into tokens by identifying delimiter characters such as underscores and capital letters.

The algorithm *semanticStructureMatchDataTypes* (Figure 4(a)) is largely adapted from the algorithm *structureMatchDataTypes* (Figure 2(a)) to identify all possible identifier correspondences between two lists of data types. It then returns the overall matching score between these two lists along with the data type correspondences as an “explanation” of the score. As can be seen in Figure 4(a), 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

^cThis is the same MaxScore assigned for two compatible data types in the structure-matching process.

^dAs with the WordNet-powered vector-space information-retrieval method, word senses are not disambiguated.

18 *Eleni Stroulia, Yiqiao Wang*

```

int semanticStructureMatchDataTypes (sourceList(m), targetList(n))
(1)  matrix = construct a  $m \times n$  matrix
(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] = matchDocumentTerms(sourceTypeName, targetType)
(8)      if (both types 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] = semanticStructureMatchDataTypes (newBaseList, newTargetList)
        + organizationBonus(sourceType, targetType)
(14)     return the matches with the maximum score;

```

(a) Algorithm semanticStructureMatchDataTypes for Matching Two Lists of Data Types

```

int semanticStructureMatchOperations ( $op_1, op_2$ )
   $op_1\_input$  = input data types associated to  $op_1$ ;
   $op_1\_output$  = output data types associated to  $op_1$ ;
   $op_2\_input$  = input data types associated to  $op_2$ ;
   $op_2\_output$  = output data types associated to  $op_2$ ;
  operationNameScore = matchDocumentTerms ( $op_1\_name, op_2\_name$ );
  parameterMatchScore = semanticStructureMatchDataTypes ( $op_1\_input, op_2\_input$ ) +
    semanticStructureMatchDataTypes ( $op_1\_output, op_2\_output$ );
  score = (operationNameScore * 2) + (parameterMatchScore * 1);
  return score;

```

(b) Algorithm semanticStructureMatchOperations for Matching Two Operations

```

int semanticStructureMatchWebServices ( $service_1, service_2$ )
  m = number of operations in  $service_1$ ;
  n = number of operations in  $service_2$ ;
  serviceNameScore = matchDocumentTerms( $service_1\_name, service_2\_name$ );
  operationMatrix = construct  $m \times n$  matrix;
  for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++)
      operationMatrix[i][j] = semanticStructureMatchOperations(list1[i], list2[j]);
  operationScore = maximum matching score according to operationMatrix;
  score = serviceNameScore * 3 + operationScore;
  return score;

```

(c) Algorithm semanticStructureMatchWebServices for Matching Two Web Services

Fig. 4. The semanticStructureMatch family of algorithms

constructs an $m \times n$ 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 2-5), if they are both primitive data types, the procedure *matchDocumentTerms* (Figure 3) is invoked to calculate the semantic similarity between names of the data types (Lines 6-7) and the score is stored in the corresponding cell of the matrix.

Similar to the structure-matching process, the similarity between identical data types is assessed using procedure *matchIdenticalTypes*, as described in heuristic 2 (lines 8 and 9). 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 identifier list(s) (as shown in lines 10-12) to be further matched recursively. Bonuses are given to matches between data types with the same organization styles (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 of identifiers represented by the matrix and returns the highest matching score between two lists of data types (Line 14).

Table 5. Identifier Matching **DataType** and **ProductType**

DataType	ProductType			
	Number: Int	Description: String	Price: float	ProductParts <all>
Id: String	3	0	0	? → 0
Category: String	0	3	0	? → 3
Item <all>	? → 3	? → 0	? → 0	? → 18 (8 + bonus)

Table 6. Structure Matching **Item** and **ProductParts**

Item	ProductParts
Quantity: int	3
Component: String	8

Let us now return to our example services *getData* and *getProduct* to illustrate how the semantic structure-matching process is performed. The algorithm *semanticStructureMatchDataTypes* (Figure 4(a)) is applied to match the composite data types **DataType** and **ProductType** of the two services. As in structure matching, because both data types are complex, all the identifiers of their sub-elements are matched to decide on their similarity score. A 3×4 matrix is constructed (Table 5). Table 5 shows how complex data structures **DataType** and **ProductType**

are matched. Identifiers of primitive data types can be mapped right away by a call to procedure *matchDocumentTerms*. For example, **DataType** \rightarrow **Id** matches to **ProductType** \rightarrow **Number** with a score of 3 because in WordNet, the words “id” and “number” are siblings under the common hypernym “identification, evidence of identity”. There are two semantic links between siblings, and according to algorithm *matchDocumentTerms*, these terms’ similarity score is 3 (6/2). In Table 5, a score of 0 indicates that there are no semantic relations between the corresponding terms in WordNet. If either data type being compared is composite, their semantic matching score depends on their elements’ matching scores. For example, **DataType** \rightarrow **Item** and **ProductType** \rightarrow **ProductParts** are both composite data types, and further matching of their sub elements is required (Table 6).

Table 7. Semantic Structure Matching **DataType** and **ProductType**

Matches	Elements of DataType	Elements of ProductType	Score
Match1, Score: 24	Id	Number	3
	Category	Description	3
	Item	ProductParts	18

Table 6 illustrates the matching process between **DataType** \rightarrow **Item** and **ProductType** \rightarrow **ProductParts**. Only one of **Item** \rightarrow **Quantity** and **Item** \rightarrow **Component** can be mapped to **ProductParts** \rightarrow **Part**. In WordNet, “quantity” and “part” are siblings with two links in between under the direct hypernyms of “concept, conception, construct”, thus “quantity” matches to “part” with a score of 3. In other senses of “part”, “part, portion, component part, component” are direct hypernyms of “item, point” (one semantic link in between). Therefore, the matching that achieves the highest score is between **Item** \rightarrow **Component** and **ProductParts** \rightarrow **Part** with a score of 8.

Similar to structure matching, because the two composite data types have the same organization style \langle all \rangle , a bonus matching score of 10 is applied and thus **Item** maps to **ProductParts** with a score of 16. The bottom-right cell of Table 5, which corresponds to this match, now has the value of ? \rightarrow 18. Other cells in Table 5 that correspond to matches between composite data types are filled in a similar manner.

The identifier matching score between **DataType** and **ProductType** can be determined as soon as all the cells in Table 5 are filled. All possible pair-wise combinations of **DataType** and **ProductType** elements are formed. The best match is the combination with the highest cumulative score. Table 7 lists the best match with a score of 24 between **DataType** and **ProductType**. Elements of **DataType** in column 2 are matched to elements of **ProductType** in column 3, and the scores in column 4 are their corresponding match scores according to Table 5. Finally because both **DataType** and **ProductType** have the same grouping style of \langle all \rangle , a bonus score of 10 is added. As a result, **DataType** matches to **ProductType** with a score of 34.

3.4.2. Semantic Structure Matching of Operations

Message identifiers are not matched in this method. This is because they are not necessarily always programmer-defined names. WSDL service specifications can be obtained by automatic translation tools, which translate the modules' public interface into their corresponding WSDL descriptions. For example, for services implemented in Java, the publicly exposed class methods are translated into operations in WSDL with method names mapped to operations' names. During this translation process, names of the request and response messages are created by concatenating the operation name with suffixes such as "in/out" or "request/response". Therefore the developer's intent in naming identifiers is captured in the operations' identifiers and the messages' identifiers are redundant.

After evaluating the data-type identifier matching scores, the query and target services' operations are matched. Given a source and a target operation, there are many possible correspondences between their parameter lists. The algorithm identifies the parameter correspondence that maximizes the sum of their individual data type identifier matching scores. Both operations' input and output parameter lists are matched, and the best parameter matching score between two operations is the sum of the best match scores between their input and output parameter lists.

In addition to matching the parameter lists associated to the operations, the operations' identifiers are also matched according to the *matchDocumentTerms* algorithm. The operations' identifiers matching score is given twice the weight given to their parameter lists matching score because we assume that names defined in the higher levels of service structures are more indicative about the capabilities of the services. Moreover, during the match, identifiers are broken into tokens by identifying delimiter characters such as underscores and capital letters and the best correspondence of the source and target tokens are then assessed. Operations matching score is the sum of operations' names matching score and their parameter lists matching score. Figure 4(b) shows the algorithm *semanticStructureMatchOperations* that takes as input two operations and returns as output their matching score.

We will now illustrate the matching process described above by matching the example services' operations, `getDataById` and `getProductByNumber`. During this process, the source operation name `getDataById` is broken into four tokens by identifying capital characters: "get", "data", "by" and "id". Tokens "get" and "by" are removed because they are stop words, and source operation name contains two remaining tokens: "data" and "id". Similarly after preprocessing, the target operation name, `getProductByNumber`, also contains two tokens: "product" and "number". Table 8 illustrates the matching between the resulting source and target operation tokens.

As shown in Table 8, the words "id" and "number" match with a score of 3 because they are siblings under the common hypernym "identification" according to WordNet. Other identifier tokens such as "data" and "product" have matching

scores of 0 because they do not have any semantic relations in WordNet. The best correspondence between the two operations names are then the mapping between tokens “id” and “number” with a score of 3. Because the matching scores of operations’ names are given twice the weight as that given to parameter matching scores as described in algorithm *semanticStructureMatchOperations* listed in Figure 4(b), the operations’ names `getDataById` and `getProductByNumber` match with a score of 6.

Table 8. Semantic Structure Matching
`getDataById` and `getProductByNumber`

<code>getDataById</code>	<code>getProductByNumber</code>	
	Product	Number
Data	0	0
Id	0	3

The operations’ input and output parameter lists are then matched. Operations `getDataById` and `getProductByNumber` take as input an `Id` and a `Number` respectively. They return as output an instance of `DataType` and `ProductType` respectively. Procedure *semanticStructureMatchDataTypes* (Figure 4(a)) is called to assess the similarity between the in and out parameter lists. Because both input parameters `Id` and `Number` are primitive, their similarity can be assessed right away using the procedure *matchDocumentTerms* and the score - as discussed in section 3.4.1 - is 3. Since both operations’ output parameters, `DataType` and `ProductType`, are composite, further matching of their sub elements is required; according to the discussion in section 3.4.2 the `DataType` and `ProductType` elements are mapped to each other with a score of 34. Therefore, operations input and output parameter lists match with a total score of 37

The operations semantic match score is the sum of matching scores of their identifiers and of the parameter lists associated with them. We have seen that operations’ names match with a score of 6 and their parameter lists match with a score of 35; therefore, the similarity score between operations `getDataById` and `getProductByNumber` adds up to 43.

3.4.3. Semantic Structure Matching of Web Services

How well two web services match depends on how well their identifiers and the operations they define correspond to each other. The algorithm *semanticStructureMatchWebServices* is used to match source and target services’ names and their operations in a pair-wise fashion to identify the best source-target operation correspondence (Figure 4(c)).

Similar to matching operations, the semantic similarity between the services’ identifiers are assessed using the WordNet semantic-distance calculation. Services’ names matching score is given a weight of 3 because the identifiers chosen for services

more frequently than not reflect the service capabilities, possibly more so than the operations' and the data-type identifiers.

An $m \times n$ matrix is constructed to match the operations associated with web services, where m is the number of the operations defined in the source service, and n is the number of operations in the target service. The procedure *semanticStructureMatchOperations*, in Figure 4(b), is invoked to assess the similarity 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. The overall service matching score is the sum of services' names matching score and their operations matching score.

Returning to our running example, the similarity between the services' names `getData` and `getProduct` is assessed. The strings `getData` and `getProduct` are separated into tokens "get", "data", and "get", "product" respectively. "Get" is a stop word and it is eliminated from the match. Therefore, matching of the services' identifiers boils down to matching the words "data" and "product". According to WordNet, there are no semantic relations between these two words. Thus, "data" matches to "product" with a score of 0, and consequently, services' names `getData` matches to `getProduct` with a score of 0.

Both services define only one operation `getDataById` and `getProductByNumber` respectively. In this case, the services' operations matching score is the matching score between these two operations which is 41 as calculated in the previous section. Thus, services `getData` and `getProduct`'s identifier matching score is 41.

4. Experimental Evaluation

To evaluate the effectiveness of the above WSDL similarity-assessment methods, we performed two experiments with two data sets. These two data sets consist of 814 and 145 WSDL specifications correspondingly, authoritatively categorized. They were both developed by Kushmerick's research group <http://moguntia.ucd.ie/repository>^e.

For each service in each data set, we queried the complete data set to retrieve the services most similar to it, using six different methods:

- (1) vector-space information retrieval on the service documentation
- (2) vector-space information retrieval on the service documentation and identifiers
- (3) WordNet-powered vector-space information retrieval on the service documentation
- (4) WordNet-powered vector-space information retrieval on the service documentation and identifiers
- (5) structure matching

^eOur own group extended further classified some the unlabeled services contained in the first data set.

(6) semantic structure matching

We then plotted the precision-recall graphs corresponding to each method's performance, assuming that for each service query, all services in the same authoritative category were an appropriate specification to be returned. Table 9 reports on some performance metrics of the various similarity assessment methods.

Table 9. Similarity-assessment methods' Performance

Method		Data Set	
		First	Second
vector-space information retrieval on the service documentation	# of failed ^f queries:	644 (79%)	93 (64%)
	# of queries with correct answer first:	75	30
	avg first position of useful service	5.45	2.15
	avg last position of useful service	78.3	12.82
WordNet-powered vector-space information retrieval on the service documentation	# of failed queries:	648 (79%)	87 (58%)
	# of queries with correct answer first:	73	29
	avg first position of useful service	5.76	3.34
	avg last position of useful service	65.94	13.98
vector-space information retrieval on the service documentation and identifiers	# of failed queries:	644 (78%)	93 (64%)
	# of queries with correct answer first:	75	30
	avg first position of useful service	5.45	2.15
	avg last position of useful service	78.38	12.82
WordNet-powered vector-space information retrieval on the service documentation and identifiers	# of failed queries:	610 (74%)	75 (51%)
	# of queries with correct answer first:	85	41
	avg first position of useful service	6.81	3.54
	avg last position of useful service	70.61	14.85
structure matching	# of failed queries:	456 (56%)	52 (35%)
	# of queries with correct answer first:	79	18
	avg first position of useful service	12.51	12.39
	avg last position of useful service	118.98	30.91
semantic structure matching	# of failed queries:	454 (55%)	54 (37%)
	# of queries with correct answer first:	149	47
	avg first position of useful service	9.69	7.48
	avg last position of useful service	117.13	24.89

The first subrow of each method row in Table 9 reports on a number of failed queries. In many cases, for a given query a method did not return a similarity score for all other services in the data set; these are the failed queries reported in the Table. All information-retrieval based methods suffer from a higher failure rate than the structure-matching methods. For the first data set this rate is at the high 70%; for the second data set, this rate ranges from 50% to 64%. This is due to the fact that a majority of the WSDL specifications in the data set do not have any

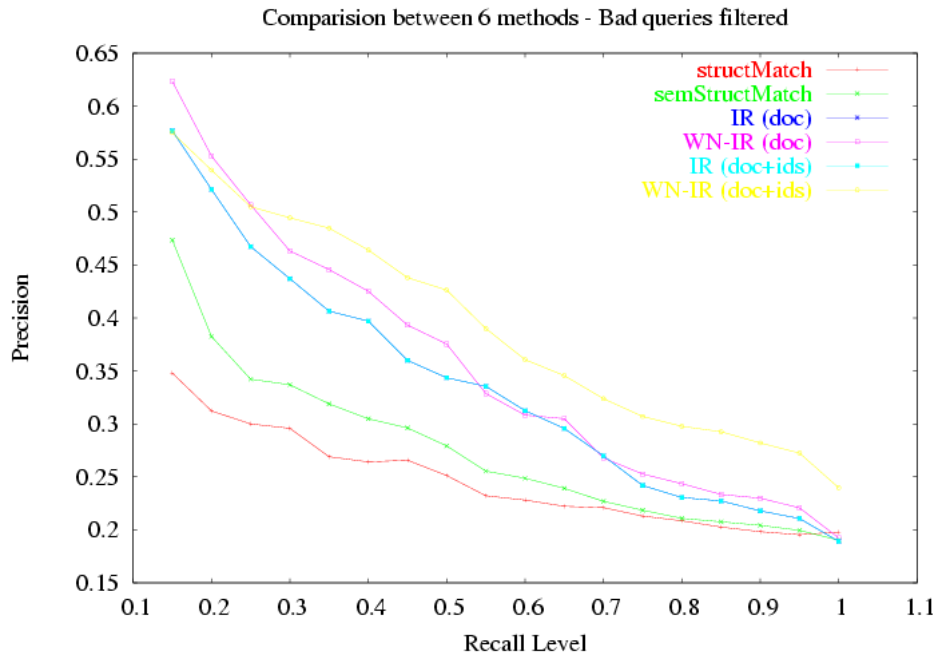
useful documentation: they either have no documentation at all, or they may have documentation in a different language or they have rudimentary documentation which gets almost completely eliminated after the stop words have been removed. The structure-matching methods suffer from a lesser failure rate, approximately 55% and 35% for the two data sets respectively, due mainly to insufficient memory of the Java virtual machine. This finding indicates that the structure-matching methods are more robust, however, a more robust implementation is needed.

Having a closer look at the performance of the methods when they did not fail, it is interesting to examine the average position of the first and the last useful service returned by these methods, since this number reflects the extent to which each of the methods may be usable. The information-retrieval methods outperform the structure-matching methods: on average, they return the first useful result around position 5 and the last useful result around position 70 in the first data set, where the structure-matching methods return the first useful result around position 10 and the last useful result around position 118. For the second data set, the corresponding numbers are 3 and 13 for the information-retrieval methods, 12 and 30 for the structure-matching method and 7 and 24 for the semantic structure matching method. This means that, on average, a developer using the structure-matching methods would have to sift through more data than a developer using the information-retrieval methods, since the potentially useful services start appearing later in the answer set and are spread more widely.

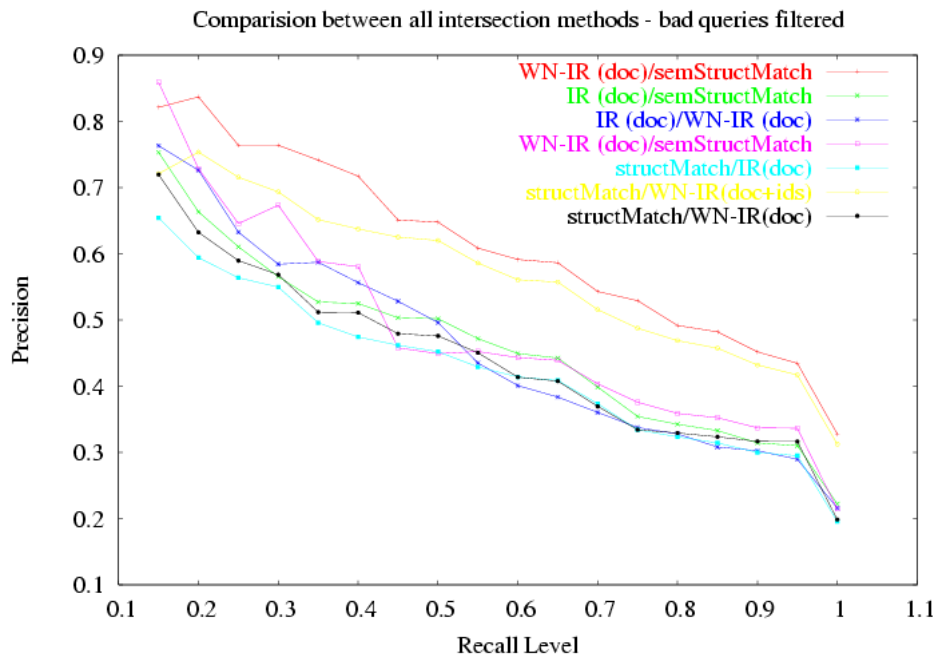
However, there is another metric for usability: namely, how frequently was the best match returned in the first position? From Table 9, we can see that in 79 out of 358 (22%) of the times and 149 out of 360 (41%) of the times for structure matching and semantic structure matching respectively, the best specification was returned as the first result. This is in contrast to 44%, 43%, 44% and 41% for the various information-retrieval based methods. The trend is similar in the second data set also. This result indicates that the natural-language elements of the WSDL specification are very characteristic of the service functionality; this is why all information-retrieval methods perform well with respect to this metric. A positive result is that, according to this metric, the more robust semantic structure-matching method is as good as the information-retrieval methods. This result is very interesting: it indicates that considering the identifier semantic similarity in the context of the specification structure is equally informative of the intended semantics of the service as the more extensive natural-language documentation elements contained in the specification.

Reviewing the data in Table 9, it also becomes evident that the traditional vector-space model information-retrieval methods applied to the service documentation and the service documentation and identifiers produce fewer failed queries than their WordNet-powered counterparts.

We then produced the precision-recall graphs shown in Figures 5 and 6 for the first and second data sets respectively. The graphs in both these Figures exclude the failed queries.



(a) the six methods' performance on the first data set



(b) intersections of the original six methods on the first data set

Fig. 5. Precision/Recall analysis of the various similarity-assessment methods on the first data set.

The first graph of Figure 5 comparatively represents the performance of the six similarity-assessment methods on the first data set. There are several interesting findings, evident in this graph.

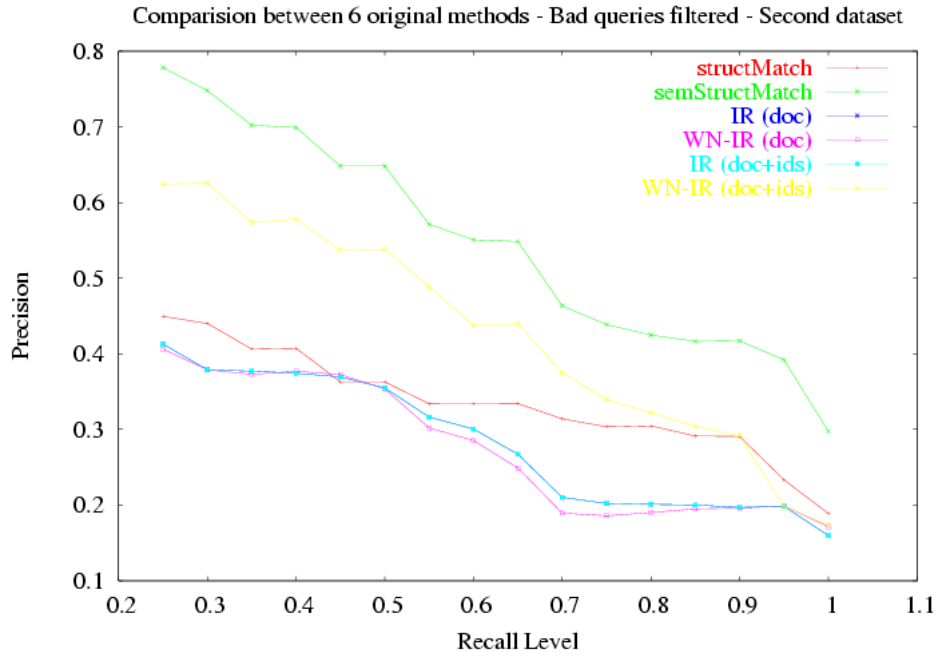
First, it is interesting to compare the performance of the various information-retrieval methods. The simple vector-space model information-retrieval method performs the same irrespective of whether or not it considers the documentation or the identifiers of the available services (the two precision-recall lines for the simple vector-space model information retrieval methods coincide). Furthermore, the WordNet-powered vector-space model information-retrieval method performs better when it considers both the documentation and the identifiers of the available services; however, the simple information retrieval performs the same as the WordNet-powered information retrieval when they are applied only to the service documentation. The implication is that identifiers are usually covered by the documentation - a developer would most likely use the same meaningful terms to characterize the modules data types and functionalities as to discuss about them in the module documentation - but they may contribute some new information when their WordNet family of terms are considered. Intuitively, it would seem that WordNet multiplies the effect of any distinct identifier term, not covered by the documentation.

The graph also shows that all information-retrieval methods perform better than both the structure-matching methods. This is again intuitive given the data reported in Table 9 about the spread of useful retrieved services in the information-retrieval answer set vs. the answer set of the structure-matching methods. In the same vein, the graph shows that the semantic structure-matching method performs better than the simple structure-matching method.

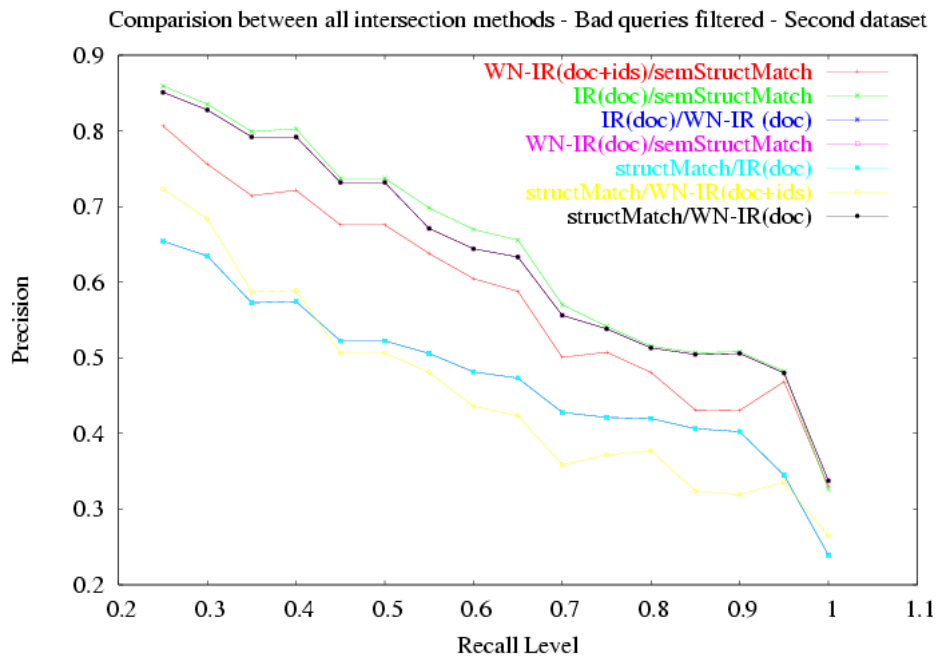
The second graph of Figure 5 comparatively represents the performance of various intersections of the original similarity-assessment methods. Not surprisingly all intersection methods are better than their respective constituent methods. Furthermore, validating the collective hypothesis underlying this work the intersection of the two most elaborate methods - WordNet-powered information retrieval and semantic structure matching - outperforms all other combinations.

Figure 6 reports on the precision and recall of the various similarity-assessment methods and their intersections on the second data set. As with the first data set, the intersection of semantic structure matching and WordNet-powered information retrieval outperforms all methods (see second graph of Figure 6). Unlike the first data set, the first graph of the Figure shows that, in this data set, the semantic structure-matching method outperforms all other original methods; furthermore the structure-matching method outperforms all information-retrieval methods except the WordNet-powered information retrieval on documentation and identifiers. The structure-matching methods have much fewer failed queries in this data set - in fact all precision-recall ratios are better than the corresponding ones of the first experiment - and it is possible that the increased failure rate of the first data set resulted in under-reporting the quality of these methods.

28 Eleni Stroulia, Yiqiao Wang



(a) the six methods' performance on the second data set



(b) intersections of the original six methods on the second data set

Fig. 6. Precision analysis of the various similarity-assessment methods on the second data set.

5. Conclusions

In this paper, we described a suite of web-service discovery methods that combine traditional information retrieval and two WordNet-based 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.

Our web-service discovery methods are inspired by traditional information retrieval methods, signature matching methods and many experiments conducted with WordNet for component retrieval. They are designed to calculate semantic and structural similarity between a desired service and a set of advertised services. WordNet-based methods do not attempt to resolve word senses; this problem has been proven difficult by current research, but fortunately it does not apply in the case of the WSDL descriptions, comments and identifiers, which are not likely to be complete grammatical sentences. WordNet is used as a query expansion semantically similar words retrieved from WordNet database for all documents and queries to ameliorate information retrieval results. The structure-matching 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. Furthermore, the semantic structure-matching method, takes into account the semantics of the chosen identifiers in structure matching, thus combining both our original hypotheses that implicit semantic information is hidden in the internal structure of the WSDL specification and the names of its elements. The superior performance of this method, especially when combined with the WordNet-powered information-retrieval method, validates our motivating hypotheses.

Our experiments have shown that we are still a long way from automatic programmatic service discovery. The similarity-assessment methods we developed are neither precise nor robust enough to discover the desired service without the developer's intervention. On the other hand, they are a great improvement over the current discovery-by-catalog-browsing functionality, supported by the UDDI API. They constitute an important extension to the UDDI API, because they enable a substantially more precise service-discovery process, greatly facilitating the user's task, who now has to closely examine much fewer specifications to find the desired one.

In the future, we plan to assess our similarity-assessment methods in the context of more coherent data sets. There is a move towards private UDDI registries with more focused, domain-specific services. We expect that the higher the coherence of the specification collection, the more precise our methods will be.

Acknowledgments

The authors wish to thank Tu Hoang for his thorough work, developing parts of these algorithms and conducting the reported experiments. This research was sup-

30 *Eleni Stroulia, Yiqiao Wang*

ported by a grant from the IRIS (Institute for Robotics and Intelligent Systems) Network of Centres of Excellence.

References

1. The DARPA Agent Markup Language <http://www.daml.org/>
2. R. Mandala, T. Takenobu and T. Hozumi. The Use of WordNet in Information Retrieval. In Proceedings of the COLING/ACL Workshop on Usage of WordNet in Natural Language Processing Systems, Montreal, 1998, 31-37.
3. G.A. Miller. Nouns in WordNet: A Lexical Inheritance System, International Journal of Lexicography, Vol3, No.4, 1990, 245-264.
4. G.A. Miller, R. Beckwith, C. Fellbaum, D. Gross and K. Miller. Introduction to WordNet: An On-line Lexical Database, International Journal of Lexicography, Vol. 3, No.4, 1990, 235-244.
5. OWL Web Ontology Language Overview <http://www.w3.org/TR/owl-features/>
6. J. Purtilo and J.M. Atlee. Module Reuse by Interface Adaptation. Software Practice and Experience, Vol. 21, No. 6, 1991, 539-556.
7. R. Richardson and A.F. Smeaton. Using WordNet in a knowledge-based approach to information retrieval. Dublin City University School of Computer Applications Working Paper CA-0395.
8. Simple Object Access Protocol (SOAP) <http://www.w3.org/TR/2003/REC-soap12-part0-20030624>
9. G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval, In Information Processing and Management: an International Journal, v.24 n.5, p.513-523, 1988
10. G. Salton, A. Wong and C.S. Yang. A vector-space model for information retrieval, In Journal of the American Society for Information Science, Vol. 18. November 1975, 13-620. ACM Press.
11. K. Sycara, S. Widoff, M. Klusch and J. Lu. LARKS: Dynamic Matchmaking Among Heterogeneous Software Agents in Cyberspace. Autonomous Agents and Multi-Agent Systems, 5, 173203, 2002.
12. UDDI technical paper, http://www.uddi.org/pubs/Tru_UDDI_Technical_White_Paper.pdf
13. E. Voorhees. Using WordNet for Text Retrieval, in C.Fellbaum (ed.), WordNet: An Electronic Lexical Database 1998, The MIT Press, Cambridge, MA. 1999, 285-303.
14. Y. Wang, E. Stroulia: Flexible Interface Matching for Web-Service Discovery, 4th International Conference on Web Information Systems Engineering, December 10-12 2003, Roma, Italy, pp. 147-156, IEEE Press.
15. Y. Wang, E. Stroulia: Semantic Structure Matching for Assessing Web-Service Similarity, 1st International Conference on Service Oriented Computing, pp. 194-207, Lecture Notes in Computer Science, Springer 2003, December 15-18 2003, Trento, Italy.
16. WordNet <http://www.cogsci.princeton.edu/~wn/>
17. Web Services Description Language (WSDL) <http://www.w3.org/TR/wsdl>
18. A. M. Zaremski and J. M. Wing. Signature Matching: a Tool for Using Software Libraries. ACM Transactions on Software Engineering and Methodology, Vol. 4 No. 2, 146-170, Apr. 1995.
19. A. M. Zaremski and J. M. Wing. Specifications Matching of Software Components. ACM Transactions on Software Engineering and Methodology, Vol. 6, No. 4, 333-369, Oct. 1997.