Detecting Semantic Matching in Service Oriented System Integration

Dario Saveliovsky, Jiayao Zhou, Yuhong Yan
Dept. of Computer Science and Software Engineering, Concordia University, Montreal, Canada

***Abstract*—In the real world, Web services are developed by independent service providers. Even if the services provide the same functionality, the names for the service parameters and the data structures of these parameters do not follow any standards. This kind of interface incompatibility is a major obstacle when people try to automate service integration, i.e. to connect services by using the outputs of a service as the inputs of another. In the existing research, people try to classify the interface incompatibility into several patterns and provide resolutions on these patterns. However, real world services largely do not follow these patterns. This paper tries to find element mappings across service interfaces by employing schema matching techniques extended for the specifics of Web service definitions. In this method, the matching of the XML elements is not only based on their semantics, but also their data structure. In addition, several techniques, such as stop words removal, are developed to improve the performance. A proof of concept service integration assistant is presented and tested on sample data sets that include a real world case which is the collaboration between mainstream Enterprise Resource Planning and Product Lifecycle Management software systems.**

***Keywords*-semantic matching; service adaptation; service composition; service integration**

I. Introduction

Service Oriented Architecture (SOA) is a software development paradigm that is based on the design of services which are published and provided over a network through a set of common interfaces [11]. These services can be described in machine-readable formats based on open standards, which allows for the possibility of services to be automatically integrated. This is a major breakthrough from previous technologies, where integration of multiple information systems was a necessarily labour-intensive task with developers having to write adaptors manually. Therefore, as the SOA paradigm is adopted by more and more software systems, it becomes more relevant to find an efficient way for services to be integrated seamlessly.

Some of the proposed solutions for the automatic integration of services (such as [16], [15], [4]) suggest the inclusion of metadata in service definitions in order to provide a semantic description of the services, extending current web service standards to include links to predefined ontologies. However, for this technique to work, a standard would need to be adopted by the industry. As this has not happened yet, the majority of web services in existence today do not include semantic information in their definitions [16]. Other approaches –such as the ones in [9] and [13]– try to work around the lack of semantic information in service definitions by analyzing their contents and finding recurring patterns across services. An important issue that arises in this situation is the identification of semantically equivalent concepts across services –i.e., a mapping between elements representing the same concepts over multiple services. These approaches tend to focus on the control flow part of the problem, either leaving out the problem of interface mapping or dedicating limited resources to it.

In this paper, we develop a method for the automated identification of semantically equivalent elements across services –which distinguishes itself from other works in the fact that it targets the specific particularities of the WSDL format, making it a more appropriate strategy for web service integration.

This paper is organized as follows. Section II is an initial introduction to the problem of service integration along with a survey of the literature on the subject. Section III presents our method in detail. Section IV delineates the features and implementation of the service integration assistant tool and describes the experiments to evaluate our method. To conclude, Section V offers a review of the contributions of this paper as well as the ideas to be explored in future work.

II. The System Integration Problem

A. Modeling the Problem

Service integration is the action of bringing together individual services into one system to provide new functionality. In the real world, stateless services are the most common services that a service interface specified in WSDL declares. Although there are ways to automatically analyze WSDL files and generate service invocation code, this kind of analysis is based on XML data types. The semantics interoperability problem is not well solved.

For this research, we have teamed up with our industry partner Helix Systems Inc [5] to provide a real-world case study.

Example 1: Helix's PLM360 has multiple modules covering all the functions from project management to product data management. The modules can be installed in different servers and they communicate via SOAP messages. Many PLM360 users are also users of ERP systems. Instead of using PLM360 to manage their employee information, these companies store their employee information in an ERP system. Ideally employees would need to enter their time sheet information into the ERP system only, and an automated process would transfer this data into the PLM360 system.

In theory, these two pieces of software could be made to communicate with each other through their Web services in order to transfer data from the ERP to the PLM360 system. However, for this integration to happen, an adaptor is needed since the two applications were not specifically designed to work together. A similar situation arises when a user switches from one ERP system to another, *e.g.,*from SAP to PeopleSoft. The messages that invoke SAP services cannot be used to invoke PeopleSoft services. Instead of developing a client from scratch, we want to use an adaptor between the client and the new service.

We consider services described in WSDL files in this paper. We begin with the interface definition, which is essentially a simple formalization of WSDL.

Definition 1: Given a set C of concepts, a service w is a set O of operations, where for each $o \in O$, $M_o = \{m_{in}, m_{out}\}$ is the set of messages associated with o: $o : m_{in} \leftrightarrow m_{out}$. A message $m \in M$ has optionally $i \geq 1$ parts, represented as $m = \langle c_1, c_2, \ldots, c_n \rangle$, $c_i \in C$, $i \in [1, \ldots, n]$.

In this definition, the operations are modeled as functions between the messages. m_{in} and m_{out} are input and output messages respectively. Faulty messages are considered as a special case of input or output messages. m_{in} and m_{out} can be an empty set $\emptyset$, meaning there is no input message or output message for this operation. The concepts C represent the parts in the messages. The concepts used in the messages are nested like a tree, *e.g.,*address has sub-parts like street name and zip code etc. Each concept $c_i \in C$ has a correspondent xml data type $d_j \in D$, where D is a set of xml data types.

Assume two services, $w_s = (C_s, M_s, O_s)$ as the provider service and $w_c = (C_c, M_c, O_c)$ as the client service. If w_c needs to send a message to invoke w_s, we need to consider their compatibility in three levels[1]:

- Semantics compatibility: the same names in different services may not mean the same thing, *e.g.,*address can contain different parts. Or vice versa, different names may mean the same thing, *e.g.,*"employee" and "worker".
- Data type compatibility: even if two names have the same semantics, their xml data types may be different. For example, "date" can be defined as xml "`date`" type in one service and xml "`string`" type in another.
- Data value compatibility: an operation may have valid inputs and invalid inputs. For example, for a stock quote service, the valid inputs are the valid stock names. Other names may trigger exceptions.

Both semantics and data types for a service are defined in its WSDL file. Therefore, analyzing the xml schema of the WSDL files is a way of solving the incompatibility problem. Data values, if collectible, can also contribute to xml schema matching [12]. However, data values are not used in this paper due to lack of data.

[1]If the services are modeled as stateful, *i.e.,*there is a communication protocol to define the sequence orders of the messaging, protocol compatibility is another level to consider. Related research can be found in [13], [9]

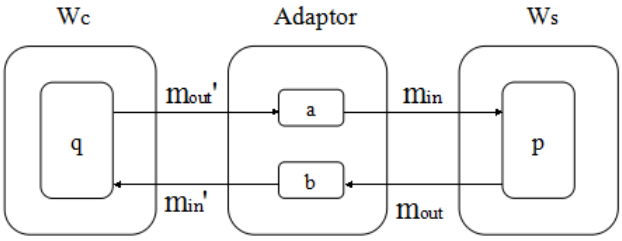

Figure 1. Interface mapping problem

As a solution to service integration, a service adaptor can be built. This adaptor is responsible for the mediation between the two services not originally designed to work together [7]. We define our Web service interface mapping problem as follows:

Definition 2: Given two services $w_s = (C_s, M_s, O_s)$ and $w_c = (C_c, M_c, O_c)$, a service adapter of w_s and w_c is also a service $a = (C_a, M_a, O_a)$ such that: for an operator $p : m_{in} \leftrightarrow m_{out}, p \in O_s$ and $q : m_{in}\prime \leftrightarrow m_{out}\prime, q \in O_c$, there are an operation $a : m_{out}\prime \rightarrow m_{in}, a \in O_a$, and an operation $b : m_{out} \rightarrow m_{in}\prime, b \in O_a$.

Figure 1 presents this definition. The adaptor works between two services. After the message matching, the adaptor can transform one SOAP message to another, using, for example, an XSLT based technique [3]. For the non-matching part, the adaptor needs to fill the data fields with some default values. One can see that after the message matching is done, programming the adaptor is trivial.

B. Schema Matching and Previous Work

Schema matching is the task of identifying semantic correspondences among elements across different schemas. It plays a central role in many data application scenarios: in *data integration* to identify and characterize inter-schema relationships across multiple schemas; in *data warehousing* to map data sources to a warehouse schema; in *E-business* to map messages between different xml formats; in the *semantic Web* to establish semantic correspondences between concepts of different ontologies [1]. In the survey paper [12], different automatic schema matching methods are summarized. These methods can be classified using different criteria, for example schema vs. instance (*i.e.,*data values); element vs structure; language vs. constraints (*e.g.,*based on keys and relationships). For WSDL files, schema-level techniques are more suitable for our purposes, since instances of service invocations are not always available. Linguistic similarity and structural information are the kinds of information we can use.

An XML schema can be modeled as a tree structure, using nodes to represent elements and attributes in the schema, and edges to represent the relationships between them. A formal definition of this structure can be seen in Definition 3 [1].

Definition 3: A schema tree T is a 4-tuple T = (N_T, E_T, Lab_{NT}, l) where:

- $N_T = \{n_1, n2, ..., n_n\}$ is a set of uniquely identified nodes, which represent either an element or an attribute definition in the XML schema.
- $E_T = \{(n_i, n_j) | n_i, n_j \in N_T\}$ is a set of edges, which represent a parent-child relationship between two nodes.
- Lab_{NT} is a set of labels, which represent node properties, including name and data type.
- $l : N_{NT} \rightarrow Lab_{NT}$ which maps every node to its labels.

The following linguistic measures are used in our method:

Levenshtein or Edit Distance: This is a measure of the number of character operations needed to transform one string into another. The allowed operations are insertion, deletion and substitution. For example, the distance between the words *manually* and *January* is 3, obtained by replacing *m* for *j*, removing the first *l*, and replacing the second one by an *r*. This metric can be normalized into a [0, 1] interval, as in [2]: $norm\_ed\_dist(t_1, t_2) = \frac{\max(|t_1|,|t_2|) - ed\_dist(t_1,t_1)}{\max(|t_1|,|t_2|)}$.

Dice Coefficient or n-gram Distance: n-grams are the sequences of n contiguous characters contained in a string. For example, the 3-gram sets for *manually* and *January* are {man, anu, nua, ual, all, lly} and {jan, anu, nua, uar, ary} respectively. The ratio of common n-grams over the total number of n-grams is computed, the following formula: $dist(s_1, s_2) = 2 * \frac{|ngrams(s_1) \cap ngrams(s_2)|}{|ngrams(s_1)| + |ngrams(s_2)|}$ [6]. The common 3-gram between *manually* and *January* are *anu* and *nua*, so the distance is $2 * 2/11 = 0.3636$.

These metrics can be used to compare the terms that are extracted from node names by splitting these names into tokens. The process then continues by comparing the token sets of each pair of nodes n_S and n_T belonging to the source and target tree respectively. The data types of the XML elements are also compared during this phase, by using a lookup table [1] that assigns compatibility values between pairs of XML data types. For example, the compatibility value between `string` and `string` is 1, between `string` and `decimal` is 0.2. The combination of these measures is called **linguistic similarity**.

We look at the XPrüM system [1] for addressing **structure similarity**. A node is associated with its post-order number and its parent's post-order number. A node's parents, children, and siblings can easily be computed with these numbers. A node *n* is said to have: a) a child context, containing the immediate children of *n*; b) a leaf context, containing all leaf nodes descending from *n*; and c) an ancestor context, containing all nodes in the path from the root node to *n*.

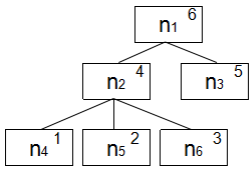

Figure 2. Schema tree with post order numbers

Example 2: Figure 2 shows a sample schema tree with its nodes labeled with both their names and their post order numbers. We present the tree as $(n_4(4), n_5(4), n_6(4), n_2(6), n_3(6), n_1(-))$, where the names are sorted by their post order numbers and their parents' post order numbers are in the bracket. From the sequence $(4, 4, 4, 6, 6, -)$, we can easily know the root of the tree, the parent, children, and the siblings of a node. The child context of n_1 is $\{n_2, n_3\}$. The leaf context of node n_1 is $\{n_4, n_5, n_6, n_3\}$. The ancestor context of node n_4 is $\{n_1 \setminus n_2\}$.

To structurally compare a pair of nodes, the similarity of each of these contexts is measured, and the three values are combined. The child contexts of nodes n_S and n_T are compared by finding, for each child of n_S, the child of n_T which has the maximum linguistic similarity to it, then computing an average of these maximum linguistic similarity values. This grants higher ***child context similarity*** values to pairs of nodes with higher proportion of children that are linguistically similar. ***Leaf context similarity*** is based on the ***gap vectors*** of the nodes being compared. A gap vector contains the differences between the post order number of a node and that of each of its descending leaves. For example, the gap vector of n_1 in Figure 2 is $\{5, 4, 3, 1\}$. The similarity between two leaf contexts is then defined as the cosine measure of their gap vectors. ***Ancestor context similarity*** is a measure of how two paths –from the tree root to the node being compared– resemble each other. This is obtained by computing the weighted sum of three measures, all of them normalized to fit in the [0, 1] range:

- A measure of the gaps between the nodes in the paths, in order to give higher scores to nodes that are closer together. We set the initial gap value to the longest length of two nodes' paths (P1 and P2). Then we find the occurrences of the nodes in P1 and the nodes in P2 that are close to each other. If the linguistic similarity between the node in P1 and the one in P2 exceeds a predefined threshold, the gap reduces.
- The difference between the lengths of the paths, assigning a higher value to paths of similar length.
- The longest common sequence between the paths, to ensure similar nodes in both paths appear in the same order. To determine if two nodes are common, their

linguistic similarity is checked against a predefined threshold. The following equation shows how to compute the longest common sequence, where $lsim(n_i, n_j)$ is a function that returns the linguistic similarity between the two nodes n_i and n_j, and th is a predefined threshold:

$$LCS[i,j] = \begin{cases} 0 & i = 0, j = 0 \\ LCS[i-1, j-1] + 1 & lsim(n_i, n_j) \geqslant th \\ \max(LCS[i-1, j], LCS[i, j-1]) & lsim(n_i, n_j) < th \end{cases} \quad (1)$$

III. Techniques for WSDL Schema Matching

A. The Challenges in WSDL Schema Matching

Web service parameters are defined as XSD schemata. Therefore, the identification of semantically equivalent parameters across a couple of services can be seen as a type of schema matching. However, the nature of WSDL adds some particularities to this variation of schema matching, which makes it worth studying as a separate problem. We summarize some of these issues below:

- **Lack of a formal ontology.** Web service parameters are defined using terms that are meaningful to the designers and the intended main target users of the service. However, these terms do not usually conform to a standard industry-agreed ontology.
- **Stop words.** Some of the terms which appear in parameter names do not contribute to the meaning of the fields in which they occur.
- **Highly context dependent.** The terms used to define fields have a specific meaning in the context of the industry to which the service belongs. Resulting from this, a generic stop word list may not be useful to filter out non-meaningful terms. Similarly, when trying to match terms sharing the same meaning, a standard thesaurus does not usually help.
- **Random schema structure.** Some schema matching techniques put emphasis in the similarity of their structures [14]. However, the structure of web service parameters can vary greatly and this practise may not help identifying semantically equivalent elements.
- **Variation in cardinality.** Another item that varies from service to service is the cardinality of the elements. One operation may handle a list of a specific concept while the other one works with individual instances. Therefore, this cannot be used as a comparison factor.
- **Random data types.** Similarly to the previous two points, the choice of XML data types for the definition of atomic fields can vary among multiple services, so using it as a similarity factor may not improve the results.
- **Non-matchable concepts.** The occurrence of concepts in one schema that have no matching counterpart in the other makes some branches of the schema non-matchable.

B. Processing

Our approach starts by building a schema tree for each of the two operations involved. The one corresponding to the consumer operation parameters will be called the *source tree*, while the tree that represents the parameters from the producer operation will be called the *target tree*.

Having constructed the trees, the next steps will analyze their node names to extract the terms that will be used for comparison. Following are the series of steps performed to accomplish this:

1) **Tokenization.** Node names can often contain multiple terms, formatted in different ways. For instance, they can be underscore separated or space separated. Thus, node names have to be tokenized and normalized. For example, a node called *employeeStartDate* will be tokenized into {*employee, start, date*}.
2) **Elimination of stop words.** The next task we want to accomplish is cleaning up the schema trees from terms that are not relevant with a list of stop words. Examples of generic stop words are “a” and “and”. In our real-world case, the data set includes time sheet information. Therefore, the words “time” and “sheet” are associated to the data set in general, but do not provide meaning about the individual nodes. These words will thus be added to the stop word list. Nodes in both trees will be scanned and any occurrences of stop words in their token sets will be removed from them.
3) **Addition of synonyms.** As similar concepts are often described using different terms by the distinct services, we use a thesaurus to find synonyms for these terms. For each node in the schema trees, tokens are looked up in the thesaurus, and their synonyms are added to the node’s token set.
4) **Combination of parent and child nodes.** Because of the hierarchical structure of Web service parameters, it is common to have nodes whose names are not sufficiently descriptive of the concept they represent. Instead, the concept is defined by both the node and its ancestors. To circumvent this issue, we have decided to propagate a node’s tokens toward its descendants. One possibility was to assign, for a given node, the union of all tokens found in the path from it to the root. The problem with this option is that it essentially flattens the tree structure, and makes most nodes similar to each other. Therefore, a better option was to combine only a node’s tokens with the ones from its parent. As shown on section IV-D, our experiments have confirmed that this mechanism produces better results.
5) **Removal of non-matchable elements.** When we can be sure that some terms in one service do not have

matching terms in the other service, we can add them to a list which is ignored by the matching algorithm. This is one manual function in our tool (cf. Subsection IV-C).

C. Computing Linguistic Similarities between Nodes Tokens

Having obtained and cleaned up the token sets for the tree nodes, the next step consists in measuring the similarity between nodes in the source and target trees. To this end, we use the measure of linguistic similarity in Definition 4.

Definition 4: A measure of linguistic similarity $Lsim$ is computed as follows:

$$Lsim(n_1, n_2) = w_1 * Tcomp(n_1, n_2) + w_2 * Nsim(n_1, n_2) \quad (2)$$

where,

- $Tcomp$ is a lookup function that checks the data types of two nodes and returns a value between 0 and 1 representing their type compatibility.
- w_1 and w_2 where $w_1 + w_2 = 1$, are the weights assigned to type compatibility and name similarity respectively.
- $Nsim$ is a metric of name similarity between two nodes. We define T_1 and T_2 as the token sets of n_1 and n_2 and the name similarity between T_1 and T_2 as the average best similarity between each token in T_1 and T_2. For ($\forall t_1 \in T_1, \forall t_2 \in T_2$), $sim(t_1, t_2)$ is a combination of bigram measure and a normalized edit distance (cf. Subsection II-B). The name similarity is computed as follows:

$$Nsim(T_1, T_2) = \frac{\sum_{t_1 \in T_1} [\max_{t_2 \in T_2} sim(t_1, t_2)] + \sum_{t_2 \in T_2} [\max_{t_1 \in T_1} sim(t_2, t_1)]}{|T_1| + |T_2|} \quad (3)$$

Intuition suggested that name similarity is a more important factor in determining node similarity so we should give it a higher weight than that of data type compatibility. Our experiments have shown that this is in fact the case, with the best results obtained when the value of w_2 is five times that of w_1.

D. Complex nodes matching

With the linguistic similarities computed, we continue by comparing all pairs of **complex nodes** (n_S, n_T) where both n_S and n_T have children. We use an average of three metrics for this comparison:

- The child context similarity measure of n_S and n_T.
- The leaf context similarity measure of n_S and n_T.
- The linguistic similarity of n_S and n_T.

By using these three metrics, we can consider not only the similarity of the two nodes being compared, but also the similarity of the nodes that descend from them. In other words, for a pair of nodes, the similarity of their descendants contributes to their own similarity. We then proceed to select **compatible nodes**, i.e. the pairs of complex nodes whose similarity exceeds a predetermined threshold.

E. Simple nodes matching

Simple nodes are the leaf nodes. For WSDL, simple nodes are the ones that matter, because they correspond to the parts in the messages. The correct matching of the simple nodes determines the effectiveness of our method. As simple nodes do not have child context and leaf context, only the ancestor context and linguist similarity are used.

We assume only the simple nodes within the matching complex nodes need to be examined. For each pair of the matching complex nodes, we extract their ***category sets*** as done in [1]. A category set is built from the union of: a) all its leaf children; b) all its non-leaf children that are not compatible nodes; and c) all the children of the latter which are leaves.

Example 3: Examine n_1 in Figure 2. If its child n_2 is a compatible node, the category set of n_1 is $\{n_3\}$. Otherwise, the category set of n_1 is $\{n_3, n_4, n_5, n_6\}$.

We start to compare the similarity between all pairs of simple nodes (n_S, n_T) in the two category sets using:

- The ancestor context similarity of n_S and n_T.
- The linguistic similarity of n_S and n_T.

Algorithm 1 Three Thresholds Filter

Data: n_S: a source leaf node; $[n_1, n_2, \ldots, n_\alpha]$: a sorted list of leaf candidates; TH_{tc}, TH_{df}, TH_{cs}: three thresholds;
Functions: SSV: calculate simple similarity value; CSV: calculate candidate similarity value

```
1: if SSV(n_S, n_1) ≥ TH_tc then
2:   matchings ← n_1
3:   for n_i ∈ [n_2, ..., n_α] do
4:     if SSV(n_S, n_1) − SSV(n_S, n_i) ≤ TH_df then
5:       positiveList ← n_i
6:     end if
7:   end for
8:   for n_i ∈ positiveList do
9:     if CSV(n_1, n_i) ≥ TH_cs then
10:      matchings ← n_i
11:    end if
12:  end for
13: end if
14: return matchings;
```

We get one *Simple Similarity Value (SSV)* by combining the above two metrics. Assume function $SSV(p, q)$ computes SSV. It would be possible at this point to take the top target node —or the top *k* target nodes— for each source node and present them as the matches. However, some of these candidate matches may not be strong enough, and we would like to filter them out. Normally one predefined

threshold is used as the cut line, for example in [1]. However, we have found that there is considerable variation between the similarity values of different correct matches. In addition, we want to be able to find multiple matches (*i.e.,*one node has multiple matches), if they exist. To get a better filtering mechanism, we propose a more complex filtering algorithm, Algorithm 1, where three thresholds are used.

First, we have a source element n_S and top α matching candidates $[n_1, \ldots, n_\alpha]$ sorted by their SSV values (Inputs in Algorithm 1). We compare n_S with its top candidate n_1 against the first threshold TH_{tc} called "*top candidate threshold*" (Line 1 in Algorithm 1). If the similarity value exceeds the threshold, n_1 is considered as a positive match (Line 2). Then we check how close are the SSV values of the other candidates to the SSV values of the top candidate (Line 3). If the difference is lower than the second threshold TH_{df} called "*difference threshold*" (Line 4), we add the candidate into a potential positive list (Line 5). Now we begin to compare the similarity of the top candidate n_1 with other candidates n_p in the potential positive list. For this comparison, we use two metrics:

- The linguistic similarity of n_1 and n_p.
- The gap difference to ensure the occurrences of P_{n_1} (path from root node to n_1) nodes in the P_{n_p} (path from root node to n_p) are close to each other.

The combination of the above two metrics is called *Candidate Similarity Value (CSV)*. Assume $CSV(p, q)$ is a function to compute CSV value between two nodes p and q. If a CSV value exceeds the third threshold TH_{cs} called "*candidate similarity threshold*" (Line 9), we consider the candidate as a positive match (Line 10). This method guarantees that multiple matches can be totally identified. And section IV-D offers a comparison of the results obtained using the one-threshold and three-threshold selection methods.

IV. Experimental result

A. Data Collection

In order to test our method for service parameter matching, we have compiled a series of sample data sets for both real-world and synthetic cases. The real-world example using the PLM360 software from our industry partner Helix is taken from their employee time sheet web service. As a matching counterpart we selected an employee time sheet query operation provided by SAP.

As synthetic examples, we defined two pairs of services for a computer repair context ("Suppliers" and "Order" in Table I). In the first case, the source service offers a *getSupplier* operation while the target service offers its counterpart called *addSupplier*. The second pair of services offer operations *getOrder* and *placeOrder* respectively, which deal with purchase orders of computer parts. The last pair of sample services was defined based on Amalgam, a schema and data integration benchmark suite developed at the University of Toronto [8]. This suite contains a set of schemata of bibliographic databases ("Biblio" in Table I). We used two of these schemata as a basis for two library operations called *getBookByAuthor* and *addBook*. Table I shows details about each of these data sets. All the data sets are downloadable from our site [17].

	SAP/PLM360	Suppliers	Biblio	Orders
Nodes (source / target)	157/250	15/19	18/31	19/50
matching nodes	20	7	8	12

Table I
DATA SETS

B. Evaluation Criteria

To evaluate the effectiveness of our approach, we ran it over the data sets and analyzed the results obtained. Before running the experiment on a data set, we manually identified the semantically equivalent parameters between the two services involved. We then checked how many of these computed mappings are in the list of correct mappings — these are called the *true positives*. Conversely, the computed mappings that do not occur in the list of manually identified mappings are the *false positives*. Finally, the source nodes for which no match has been found are also checked against the correct mappings. The ones that have in fact a correct mapping are our *false negatives*. With these three values, we can compute three measures: precision, recall and F-measure, as shown below [10].

$$precision = \frac{tp}{tp + fp}, recall = \frac{tp}{tp + fn},$$

$$F - measure = \frac{2}{\frac{1}{precision} + \frac{1}{recall}}$$

Precision measures the correctness of the results. Recall, on the other hand, is a measure of the completeness of the results. The F-measure is defined to combine both of these metrics in a single value by computing their harmonic mean.

C. Semi-automatic WSDL Schema Matcher

We have developed a semi-automatic tool, WSDL Schema Matcher, aimed at aiding developers in the finding of semantical matches across two web services. We have run tests using this tool on the data described in Section IV-A. Figure 3 shows the architecture of our tool. The inputs of our tool are a pair of WSDL documents and the output is a list of matching pairs. The tool has several modules for a series of tasks, such as WSDL file parsing, name tokenization, stop words removal, and various similarity calculations (cf. the boxes at the right end in Figure 3). Some of the modules use auxiliary information, such as stop words, thesaurus, and a list of non-matchable concept terms. These modules are

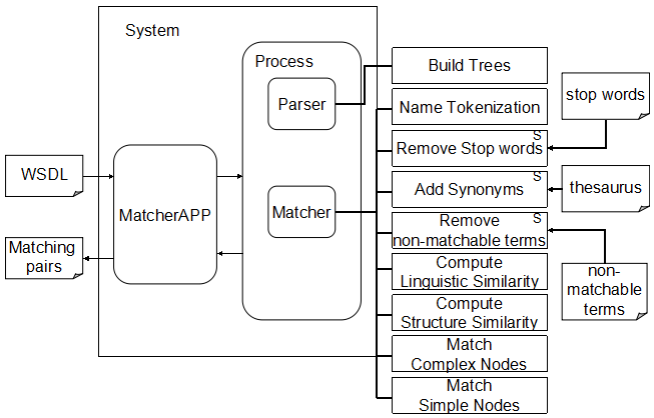

Figure 3. Semi-automatic WSDL Schema Matcher

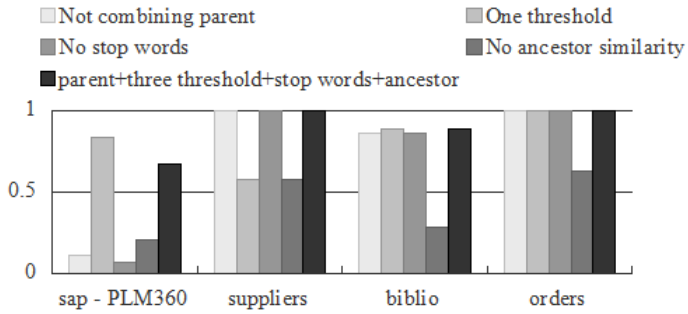

Figure 4. Data sets: Precision

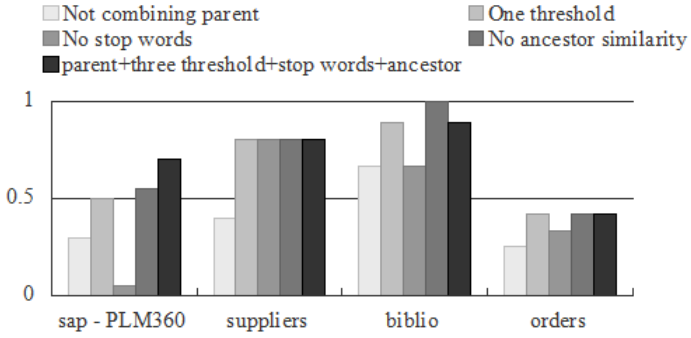

Figure 5. Data sets: Recall

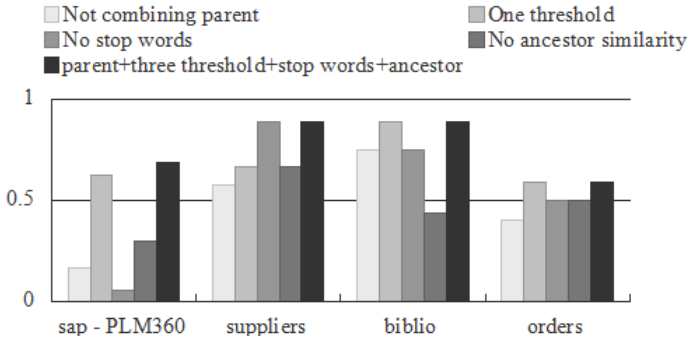

Figure 6. Data sets: F-measure

semi-automatic, in the sense that the user can manually add domain specific stop words, synonyms, etc., if the generic ones are not sufficient. Semi-automatic tasks have a mark "S" at the right corner of its box. The calculations of linguistic and structure similarities are fully automatic. It is because no techniques can match schemas 100% correctly that we need human intervention to improve performance.

D. Experimental results

All the results presented below have been obtained by running an implementation of the techniques described in section III and on the sample data described in Section IV-A.

The first experiment is to evaluate the impacts of the different techniques on the performance of our Matcher. These techniques are: **combining parent node name**, **three threshold filtering**, **the usage of stop words**, and **the usage of ancestor similarity**. Our baseline is to use all these four techniques. Then, we drop one technique from the baseline at a time. We test the precision, recall, and F-measure on each case and present the results in Figure 4, 5, and 6. The technique of removing non-matchable elements is very helpful to improve precision and recall, if the user understands the domain. However, we regard it is not very meaningful to evaluate its efficiency, because of strong human intervention.

In Figure 6, we can see that our method has the highest F-measure for all the four datasets, which means our method has the best performance. Among the matching techniques, we can see missing one technique lowers the performance in most of the cases, however, not always. One reason should be related to the nature of the data, which we do not fully understand. Others reasons like not combining parent name makes nodes less similar so that the precision of this method when compiling "orders" is 1 but recall is very low. In the case of SAP and PLM360, precision of one threshold filtering is higher than three threshold filtering's due to the latter being able to identify multiple matchings, improving recall but losing precision. Without ancestor similarity, the results will include many false positives although many right matching pairs will be identified.

Our second experiment is **comparison with COMA++.** The COMA++ system is a generic schema matching tool developed at the University of Leipzig, aimed at different schema formats and application domains.

Table II shows the results, using both our technique and COMA++. The outcomes from our approach are better than the ones from COMA++. This works as evidence that our contributed methods provide an improvement when working with the matching of Web services, since they

	TPs	FPs	FNs	PREC.	REC.	F
suppliers						
our method	4	0	1	1.0000	0.8000	0.8889
COMA++	3	3	2	0.5000	0.6000	0.5455
biblio						
our method	8	1	1	0.8889	0.8889	0.8889
COMA++	6	3	3	0.6667	0.6667	0.6667
orders						
our method	5	1	7	0.8333	0.4167	0.5556
COMA++	4	4	8	0.5	0.3333	0.4000
sap - PLM360						
our method	14	7	16	0.6667	0.7000	0.6829
COMA++	4	40	16	0.0909	0.2000	0.1245

Table II
COMPARING RESULTS FROM OUR TECHNIQUE AND COMA++

are specifically directed toward the features of the WSDL format.

V. CONCLUSION

In this paper we have offered a look into the major issue of the automated service integration problem which consists in finding semantically equivalent elements across service definitions. Our methods leverage schema matching techniques to take advantage of the XSD format of Web service operation parameters, while at the same time expanding them to tackle specific features of WSDL —namely, the lack of formal ontologies, highly context dependent meaning of terms, randomness in the schema structures and cardinality of elements and data types, and the presence of non-matchable terms. We have built a prototype of an interactive tool which implements our techniques and tested this on data samples. The samples include the real-world case study of the integration of time sheet information between Helix' PLM360 and SAP's ERP software, and synthetic examples. The results from our experiments look promising, comparing positively against the ones obtained by using a general-purpose schema matching tool. Moreover, the results of this tool can then be used in the building of a script to transform instances of messages between the formats of the services involved. In future work, we will concentrate our efforts on the automation of this task, which will bring us closer to the goal of an automated service integration solution.

REFERENCES

[1] Alsayed Algergawy, Eike Schallehn, and Gunter Saake. Improving XML schema matching performance using Prüfer sequences. *Data Knowl. Eng.*, pages 728–747, 2009.

[2] Alsayed Algergawy, Eike Schallehn, and Gunter Saake. A sequence-based ontology matching approach, 2009.

[3] Altova. Mapforce. http://www.altova.com/mapforce.html, 2014.

[4] Ajay Bansal, Srividya Kona, Luke Simon, and Thomas D. Hite. A universal service-semantics description language. In *Proceedings of the Third European Conference on Web Services*, ECOWS '05, pages 214–, Washington, DC, USA, 2005. IEEE Computer Society.

[5] Helix. Plm360 website. http://www.helix-sys.com/, 2014.

[6] Grzegorz Kondrak. N-gram similarity and distance. In *Proc. Twelfth Intl Conf. on String Processing and Information Retrieval*, pages 115–126, 2005.

[7] Woralak Kongdenfha, Hamid Reza Motahari-Nezhad, Boualem Benatallah, Fabio Casati, and Regis Saint-Paul. Mismatch patterns and adaptation aspects: A foundation for rapid development of Web service adapters. *IEEE Trans. Serv. Comput.*, 2(2):94–107, April 2009.

[8] Renée J. Miller, Daniel Fisla, Mary Huang, David Kymlicka, Fei Ku, and Vivian Lee. The Amalgam Schema and Data Integration Test Suite. www.cs.toronto.edu/ miller/amalgam, 2001.

[9] Hamid Reza Motahari Nezhad, Boualem Benatallah, Axel Martens, Francisco Curbera, and Fabio Casati. Semi-automated adaptation of service interactions. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 993–1002, New York, NY, USA, 2007. ACM.

[10] David L. Olson and Dursun Delen. *Advanced Data Mining Techniques*. Springer, 2008.

[11] Michael P. Papazoglou. *Web Services: Principles and Technology*. Pearson, 2008.

[12] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *VLDB J.*, 10(4):334–350, 2001.

[13] Zhe Shan, Akhil Kumar, and Paul Grefen. Towards integrated service adaptation a new approach combining message and control flow adaptation. In *Proc. of International Conference of Web Services*, pages 385–392, 2010.

[14] Pavel Shvaiko and Jrme Euzenat. A survey of schema-based matching approaches. In *Journal on Data Semantics IV*, volume 3730 of *Lecture Notes in Computer Science*, pages 146–171. Springer Berlin / Heidelberg, 2005.

[15] Kaarthik Sivashanmugam, Kunal Verma, Amit Sheth, and John Miller. Adding semantics to Web services standards, 2003.

[16] Katia Sycara, Massimo Paolucci, Anupriya Ankolekar, and Naveen Srinivasan. Automated discovery, interaction and composition of semantic Web services. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(1), 2011.

[17] Yuhong Yan. Testing data sets. http://users.encs.concordia.ca/~yuhong/2014/data/data_set.rar, 2014.