WSXplorer: Searching for Desired Web Services

Yanan Hao[1], Yanchun Zhang[1], and Jinli Cao[2]

[1] School of Computer Science and Mathematics, Victoria University
PO Box 14428, Melbourne, VIC 8001, Australia
{haoyn, yzhang}@csm.vu.edu.au
[2] Department of Computer Science and Computer Engineering, La Trobe University
Bundoora, VIC 3086, Australia
j.cao@latrobe.edu.au

Abstract. With the rapid development of e-commerce over Internet, web services have attracted much attention in recent years. Nowadays, enterprises are able to outsource their internal business processes as services and make them accessible via the Web. Then they can dynamically combine individual services to provide new value-added services. A main problem that remains is how to discover desired web services. In this paper, we propose WSXplorer, a novel scheme for identifying potentially relevant web services given a textual description of services. In particular, we propose a new schema matching algorithm for supporting web-service operations matching. The matching algorithm catches not only structures, but even better semantic information of schemas. Based on service operations matching, the concept of *attribute closure* is introduced to identify associations between web-service operations. We also propose a ranking strategy to satisfy a user's top-k requirements. Experimental evaluation shows that our approach can achieve high precision and recall ratio.

1 Introduction

A web service is programmatically available application logic exposed over Internet. It has a set of operations and data types. The current set of web service specifications defines how to specify reusable operations through the Web-Service Description Language(WSDL), how these operations can be discovered and reused through the Universal Description, Discovery, and Integration(UDDI) API, and how the requests to and responses from web-service operations can be transmitted through the Simple Object Access Protocol API(SOAP).

With the rapid development of e-commerce over Internet, web services have attracted much attention in recent years. Nowadays, enterprises are able to outsource their internal business processes as services and make them accessible via the Web (see, e.g.,[1,2,3,4,5]). Then they can combine individual services into more complex, orchestrated services. A main problem that remains is how to discover desired web services. To find a service in UDDI, a user needs to input some keywords about the required service and then to browse the relevant UDDI category to locate relevant web services. Considering a large amount

J. Krogstie, A.L. Opdahl, and G. Sindre (Eds.): CAiSE 2007, LNCS 4495, pp. 173–187, 2007.

of service entries, this process is time consuming and frustrating. Furthermore, this method does not provide a mechanism assisting users in selecting relevant services and composing with them. Since a web service is usually used as part of an application, users often would like to know relevant services as much as possible. For example, consider the examples shown in Fig. 1. A user searching for a *CreateOrder* service may also be interested in a *TransportOrder* service. There is an association between these two services, in which the output of *CreateOrderService*, *Order*, is also the input of *TransportOrderService*. This form of association potentially involves more web services. It is particularly useful and challenging in service composition.

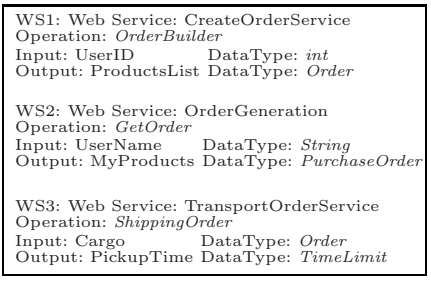

Fig. 1. Sample web-service operations

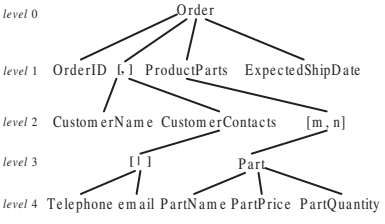

Fig. 2. XML schema tree of *Order* type

To address the problems above in searching for web services, we propose WSXplorer (Web Services eXplorer), a novel scheme for identifying potentially relevant web services given a textual description of services. The contribution of the work reported here is summarized as follows:

1. We propose algorithms for supporting web-service operations matching. The key part of our algorithms is a schema tree matching algorithm, which employs a new cost model to compute tree edit distances. Our new schema tree matching algorithm can not only catch structures, but also the semantic information of schemas.
2. Based on service operations matching, an approach to identify associations between web-service operations is presented. This approach uses the concept of *attribute closure* to obtain sets of operations. Each set is composed of associated web-service operations.
3. We also introduce a ranking strategy to satisfy a user's top-k requirements. Experimental evaluation shows that WSXplorer can achieve acceptable result with high performance.

The rest of this paper is organized as follows. Section 2 reviews the related work. Section 3 gives an overview of WSXplorer. Section 4 describes a web-service operation matching algorithm, in which a new cost model and some XML schema transformation rules are defined. In section 5 we present how to cluster web-service operations and how to find associations between them. Section 6 describes our experimental evaluation. Section 7 gives some concluding remarks.

2 Related Work

Finding similar web-service is closely related to software components matching. In [6], signatures are used to describe a component's type information (which is usually statically checkable), and formal specifications are defined to describe the component's dynamic behaviour. Two components match if their signatures and specifications match. However, the formal specifications used there are function's post conditions, which are not available in web services.

Several approaches use text or structural matching to find similar web services for a given web service. The earlier technique tModel presents an abstract interface to enhance service matching process. But the tModel needs to be defined while authors publishing in UDDI [7]. In [8], the authors propose a SVD-Based algorithm to locate matched services for a given service. This algorithm uses characteristics of singular value decomposition to find relationships among services. But it only considers textual descriptions and can not reveal the semantic relationship between web services. Wang etc. [9] discover similar web services based on structure matching of data types in WSDL. The drawback is that simple structural matching may be invalid when two web-service operations have many similar substructures on data types.

Recently, some methods have been proposed to annotate web services with additional semantic information. These annotations are used to match and compose services. For example, in [10] the authors extended DAML-S to support service specifications, including behavior specifications of operations; The Web Service Modeling Ontology (WSMO) [11] is a conceptual model for describing Web services semantically, and defines the four main aspects of semantic Web service, namely Ontologies, Web services, Goals and Mediators. However, currently, most of existing web services use WSDL specifications, which do not contain semantics. Annotating the collection of services requires much effort, and it is infeasible in our case. [12] formally defines a behaviour model for web service by automata and logic formalisms. However, the behaviour signature and query statements need to be constructed manually, which can be very hard for common users.

Woogle [13] develops a clustering algorithm to group names of parameters of web-service operations into semantically meaningful concepts. Then these concepts are used to measure similarity of web-service operations. It relies too much on names of parameters and does not deal with composition problem however. In our previous work [14] we use schema to find web services, but the associations between services are not considered. In [15] the authors propose a syntactic approach to web service composition, given only the input-output types of web services available in their WSDL descriptions. Discover [16,17] and DBXplore [18] operate on relational databases and facilitates information discovery on them by allowing users to issue keyword queries without any knowledge of the database schema. They return sets of tuples that are associated by joining on their primary and foreign keys. Inspired by these methods, we model each web-service operation as a dependency (schema) according to its data types (attributes), and then find associations between web-service operations.

3 Overview of WSXplorer

The goal of WSXplorer is to find relevant web-service operations given a natural language description of desired web services and WSDL specifications of all available services published through UDDI. The WSDL files consist of textual description of web-service operations. Thus, firstly we use traditional IR technique TF (*term frequency*) and IDF (*inverse document frequency*) [19] to find service operations that are most similar to the given description. We call these operations *candidate operations*. To do this, we extract words from web-service operation descriptions in WSDL. These words are pre-processed and assigned weight based on IDF. According to these weights, the similarity between the given description and a web-service operation description can be measured. A higher score indicates a closer similarity. For more details on measuring similarity among documents interested readers are referred to see [20]. After obtaining candidate operations, we employ a schema-match based method to measure similarity among them. Then based on the matching result the candidate operations are clustered into some *operation sets*. Each operation set contains a group of similar operations. Finally, all associations between operation sets are generated using the concept of *type closure*. Operations involved in one association are considered as a search result. Since each candidate operation has a score, we can rank search results simply by accumulating the score of operations. In the following sections we describe the models and algorithms underlying WSXplorer, in particular we show how to measure similarity between web-service operations based on schema matching.

4 Web-Service Operation Matching

4.1 Web-Service Operation Modelling

Definition 1. *A web service is a triple* $ws = (TpSet, MsgSet, OpSet)$*, where TpSet is a set of data types; MsgSet is a set of messages(parameters) conforming to the data types defined in TpSet;* $OpSet = \{op_i(input_i, output_i) | i = 1, 2, ..., n\}$ *is a set of operations, where* $input_i$ *and* $output_i$ *are parameters(messages) for exchanging data between web-service operations.*

Figure 1 gives three web-service operations used as examples in this paper. According to definition 1, a web service can be briefly described as a set of operations.

Definition 2. *Each web-service operation is a multi-input-multi-output function of the form* $f : s_1, s_2, ..., s_n \rightarrow t_1, t_2, ..., t_m$*, where* s_i *and* t_j *are data types in according with XML schema specification. We call f a* ***dependency*** *and* s_i/t_j *a* ***dependency attribute****.*

A dependency attribute can be a complex data type or a primitive data type. Complex data types, such as *Order* and *PurchaseOrder* in Fig. 1, define the structure, content, and semantics of parameters, whereas primitive data types,

like *int* and *string*, are typically too coarse to reflect semantic information. Since parameters usually can be regarded as data types, we can convert primitive data types to complex data types by replacing them with their corresponding parameters. For example, in Fig. 1 *string* is converted into *UserName* type while *int* is converted into *UserID* type. Both *UserName* and *UserID* are considered as complex data types with semantics. Thus, each data type defined in a web-service operation carries semantic meaning. An XML schema can be modelled as a tree of labelled nodes. We categorize a node n by its label:

1. **Tag node**: Each tag node n is associated with an element type T. T is also the tag name of node n.
2. **Constraint node**:
   - **Sequence node**: A sequence node indicates its children are an ordered set of element types. We use [,] to denote a sequence node.
   - **Union node**: A union node represents a choice complex-type, that is, the instance of which can only be one of the children types in accordance with the XML Schema specification. We use [|] to denote a union node.
   - **Multiplicity node**: Each node may optionally have a multiplicity modifier $[m, n]$ indicating that in the instance, its occurrence is between m and n. This corresponds to the *minOccurs* and *maxOccurs* constraints in XML Schema. We use $[m, n]$ to denote a multiplicity node.

As an example, the schema tree of data type *Order* is shown in Fig. 2.

As we can see, data types defined in web-service operations carry semantic information. Intuitively, we consider two web-service operations similar if they have similar input/output data types. Thus the problem of web-service operation matching is converted to the problem of schema tree matching.

4.2 Tree Edit Distance

Many works have been done on the similarity computation on trees. Among them *tree edit distance* is one of the efficient approaches to describe difference between two trees. We introduce tree edit operations first. Generally, the tree edit distance operations include: (a) *node removal*, (b) *node insertion*, and (c) *node relabelling*. Such a set of operations can be represented by a mapping with minimum cost between the two trees. The concept of mapping is formally defined as follows [21]:

Definition 3. *Let T_x be a tree and let $T_x[i]$ be the* i*th node of tree T_x in a pre-order traverse of the tree. A mapping between a tree T_1 and a tree T_2 is a set* M *of ordered pairs (i, j), satisfying the following conditions for all (i_1, j_1),$(i_2, j_2) \in M$*

1. *$i_1 = i_2$ iff $j_1 = j_2$;*
2. *$T_1[i_1]$ is on the left of $T_1[i_2]$ iff $T_2[j_1]$ is on the left of $T_2[j_2]$;*
3. *$T_1[i_1]$ is an ancestor of $T_1[i_2]$ iff $T_2[j_1]$ is an ancestor of $T_2[j_2]$.*

Figure 3 gives an example of tree mapping. This mapping also shows the way of transforming the left tree to the right one. A dotted line from a node of T_1 to a

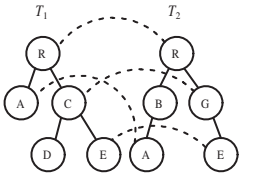

Fig. 3. Example of tree mapping

node of T_2 indicates that the node of T_1 should be changed if the corresponding nodes are different, remaining unchanged otherwise. Nodes of T_1 not connected by dotted lines are deleted, and nodes of T_2 not connected are inserted. Each of these operations is assigned a cost. The tree edit distance between two trees is defined as the minimal set of operations to transform one tree into the other.

Our schema matching algorithm is based on tree edit distance. However, the problem in our case is more complex than the traditional tree edit distance for the following reasons:

1. The labels of an XML Schema tree can carry complex type information (e.g., union, multiplicity) which makes simple relabelling operations inapplicable. For instance, let T_1 and T_2 be the schema trees of *Order* and *PurchaseOrder* respectively. Let us imagine there exits a mapping M between T_1 and T_2, and there are two node-mapping pairs (i_1, j_1),$(i_2, j_2) \in M$, where $T_1[i_1]$ =[*telephone*|*email*], $T_2[j_1]$ =*email*, $T_1[i_2]$=*price*, and $T_2[j_2]$=*quantity*. The edit operation of (i_1, j_1) should have less cost than that of (i_2, j_2). But the existing work consider all tree edit operations to have same unit distance.
2. The labels of nodes carry semantic information. So a relabelling from one node to another unrelated node will have more cost than to a semantic related node. For example, relabelling *part* to *item* is less costing than relabelling *price* to *email*.
3. We argue that tree edit operations on low-level nodes of a tree should have more influence than operations on high-level nodes. For example, in Fig. 2, node *Order* is more important than node *PartPrice*, because *Order* denotes broader semantics information than *PartPrice*. So, if a *PartPrice* node of the first tree is mapped into an *Order* node of the second tree, the edit operation cost should not be zero. But the traditional works on tree edit distance do not consider the difference and assign each edit operation unit cost.

In the next section, we present a new cost model to compute the cost of tree edit operation, as a consequence, the tree edit distance of two schema trees.

4.3 Cost Model

Measuring similarity between two XML schema trees equals to finding a mapping with minimum cost. So, the cost of each edit operation involved in the mapping needs to be computed first. [22] proposed a algorithm for fast computing tree

edit distance, but it assigned the same cost for each unit edit operations on all nodes and overlooked nodes difference. Authors in [23] introduced a summary structure for computing structural distance and took weight information into account for nodes in distance computation, but it did not consider the semantic difference or similarity. In this section we introduce a new cost mode based on tree edit distance presented in [22,23]. The new cost model integrates weights of nodes and semantic connections between nodes. Let T_1, T_2 be two schema trees and let n, $node_1$ and $node_2$ be tree nodes. Formally, the cost model is defined as

$$cost(\rho) = \begin{cases} weight(n)/W(T_1,T_2), & \text{if} \rho = insert(n) \\ weight(n)/W(T_1,T_2), & \text{if} \rho = delete(n) \\ \alpha \times wd(node_1, node_2) & \text{if} \rho \text{ relabels} \\ +\beta \times sd(node_1, node_2) & node_1 \text{to } node_2 \end{cases} \quad (1)$$

where ρ indicates a tree edit operation. $weight(n)$ shows the weight of node n, which is defined in definition 6. $wd(node_1, node_2)$ and $sd(node_1, node_2)$ give the weight and semantic difference of $node_1$ and $node_2$, respectively. α and β are weights of wd and sd, satisfying $\alpha + \beta = 1$. $W(T_1, T_2)$ is defined as $W(T_1, T_2) = weight(T_1) + weight(T_2)$, where $weight(T_i)$ is the sum of all node weights of tree $T_i (i = 1, 2)$. $wd(node_1, node_2)$ is defined as

$$wd(node_1, node_2) = \frac{\|weight(node_1) - weight(node_2)\|}{W(T_1, T_2)} \quad (2)$$

where $node_1 \in T_1$ and $node_2 \in T_2$.

In equation 1, $weight(n)/W(T_1, T_2)$ explains the cost of inserting or deleting node n. For the relabel operation, both weight and semantics of $node_1$ and $node_2$ can be different, so we use the combination of weight and semantic difference as the relabel cost. All the costs are normalized by $W(T_1, T_2)$, i.e. the sum of all nodes weights of tree T_1 and T_2.

In the next two sections, we propose a set of schema-tree transformation rules and a semantic similarity measure to compute wd and sd, i.e. the weight and semantic difference of nodes.

4.4 XML Schema Tree Transformation

Definition 4. *The tag name of a node is typically a sequence of concatenated words, with the first letter of every word capitalized (e.g., ExpectedShipDate). Such a set of words is referred to as a* ***word bag****. We use $\pi(n)$ to denote the word bag of node n.*

Definition 5. *Two word bags $\pi(n_1)$ and $\pi(n_2)$ are said to be equal, only if they have same words.*

Two nodes are considered different if they have different word bags. The word bag reflects semantic meaning of a node. As we shall see later, using word bags we can measure the semantic similarity between two schema-tree nodes.

Definition 6. *Let level(n) denote the level of node n in schema tree T. The weight of node n is defined by a weight function:*

$$weight(n) = 2^{depth(T)-level(n)} (\forall n \in T) \tag{3}$$

The weights of all nodes fall in the range of $[2, 2^{depth(T)}]$. Each weight reflects the importance of a node in schema tree T.

From section 4.2, it can be seen that traditional tree-edit-distance algorithm is not suitable for XML schema trees. It does not deal with constraint nodes. We propose three transformation rules to solve this problem. These rules are used to transform constraint nodes, specifically, sequence nodes, union nodes and multiplicity nodes to tag nodes. At the same time, the weights of nodes are reassigned.

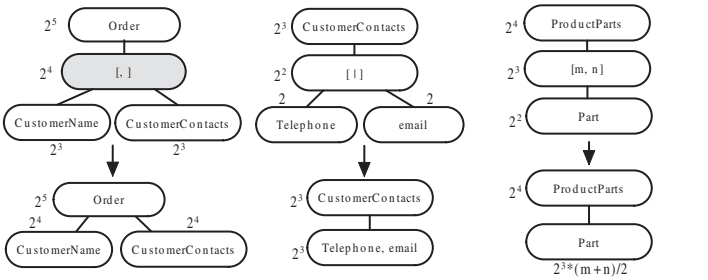

Fig. 4. XML Schema tree transformation

1. *split*: This rule is applied to sequence nodes. A sequence node $l = [l_1, l_2, ..., l_s]$ is split into an ordered list of nodes $l_1, l_2, ..., l_s$, where $l_i (i = 1, 2, ..., s)$ is a child node of the sequence node l. After the split process, each sequence node is replaced by its child nodes. Each child node l_i inherits the weight of its parent node l as a new weight. Figure 4(a) gives an example of the split rule.
2. *merge*: This rule is applied to union nodes. After the merge process, each union node is replaced by all its option nodes, i.e. all its child nodes. All child nodes of the union node $l = [l_1|l_2|...|l_s]$ are merged into a new node l^*, while the union node l is deleted. The weight of node l^* is s times the weight of l. Each l_i's$(i = 1, 2, ..., s)$ word bag is also merged into a new word bag. Formally, we have $weight(l^*) = weight(l) \times s$. Figure 4(b) gives an example of the merge rule.
3. *delete*: This rule is applied to multiplicity nodes. We delete a multiplicity node $l = [m, n](m, n \in N)$ and scale up the weight of each of its child nodes l_i. After the deletion process, each multiplicity node is replaced by its child nodes. We have $weight(l_i) = weight(l) \times (m + n)/2$. Figure 4(c) gives an example of the delete rule.

Note that the definition of complex types can be nested according to XML schema specification. Thus, given a schema tree, we apply the three transformation rules to its nodes level by level, from bottom to top. This process is formally

```
   input  : schema tree T
   output: transformed schema tree T*
 1 d = GetDepth(T);
 2 for i ← d to 0 do
 3     foreach node p ∈ level_i do
 4         if p is a sequence node then
 5             weight(each of p's child nodes)=weight(p);
 6             add p's child nodes to p's parent's child list;
 7             delete p;
 8         end
 9         if p is a union node with s options {l_i|i = 1, ..s} then
10             merge p's child nodes into a new node q;
11             add q to p's parent's child list;
12             weight(q) = weight(p) × s;
               π(q) = ⋃_{i=1}^{s} π(l_i) ;
13
14             delete p;
15         end
16         if p is a multiplicity node [m, n] then
17             add p's child node to p's parent's child list;
18             weight(p's child node)=weight(p) × (m + n)/2;
19             delete p;
20         end
21     end
22 end
```

Algorithm 1. Bottom-up-transformation

described as the *bottom-up-transformation* algorithm (see Algorithm 1). The time complexity of Bottom-up-transformation is $O(n)$, where n is the number of nodes in the XML schema tree.

4.5 Semantic Measurement Between Schema-Tree Nodes

After the bottom-up transformation, schema tree T is converted into a new schema tree T^*. Each node n of T^* is a tag node, whose word bag may come from two or more word tags because of nodes merge by the merge rule. Formally, node n can be regarded as a vector (W, B), where W is the weight of node n and B is the word bag of node n. As we can see, after transformation the weight difference between two nodes can be computed by the new cost model. In this section, we present a strategy to determine the semantic similarity of two schema-tree nodes, i.e. the semantic distance between two word bags.

WSXplorer relies on a hypothesis that two co-occurrence words in a WSDL description tend to have same semantics. We exploit the co-occurrence of words in word bags to cluster them into meaningful concepts. To improve accuracy of semantic measurement, a pre-processing step is carried out first before words clustering. Pre-processing includes word stemming, removing stop words and expanding abbreviations and acronyms into the original forms.

Let $I = \{w_1, w_2, ..., w_m\}$ be a set of words. These words come from word bags of all schema-tree nodes to which similarity measurement is applied. Let D be a set of candidate web-service operation descriptions available in WSDL files. We introduce association rules to reflect the notion of word co-occurrence. An *association rule* is an implication of the form $w_i \rightarrow w_j$, where $w_i, w_j \in I$. The rule $w_i \rightarrow w_j$ holds in the descriptions set D with *support* s and *confidence* c, where s is the probability that w_i occurs in an web-service operation description; c is the probability that w_j occurs in an operation description, given w_i is known to occur in it. All association rules can be found by the A-Priori algorithm [24]. We are only interested in rules that have confidence above a certain threshold t.

We use the agglomeration algorithm [24] to cluster words set $I = \{w_1, w_2, ..., w_m\}$ into concept set $C = \{C_1, C_2, ...\}$. There are three steps in the clustering process. It begins with each word forming its own cluster and gradually merges similar clusters.

1. Set up a confidence matrix $M_{m \times m}$. M_{ij} is a two-dimensional vector (s_{ij}, c_{ij}), where s_{ij} and c_{ij} are the support and confidence of association rule $w_i \rightarrow w_j$, respectively.
2. Find the two-dimensional vector $M_{ij} = (s_{ij}, c_{ij})$ with the largest c_{ij} in the confidence matrix M. If, for both of them, $c_{ij} > t$ and $s_{ij} > t$ then merge these two clusters and update M by replacing the two rows with a new row that describes the association between the merged cluster and the remaining clusters. The distance between two clusters is given by the distance between their closest members. There are now $m - 1$ clusters and $m - 1$ rows in M.
3. Repeat the merge step until no more clusters can be merged.

Finally, we get a set of concepts C. Each concept C_i consists a set of words $\{w_1, w_2, ...\}$. To compute semantic similarity between schema-tree nodes, we replace each word in word bags with its corresponding concept, and then use the TF/IDF measure. After schema-tree transformation and semantic similarity measure, the tree edit distance can be applied to match two XML schema trees by the new cost model.

4.6 Web-Service Operations Matching

As it has been mentioned before, we use tree edit distance to match two schema trees. It is equivalent to finding the minimum cost mapping. Let M be a mapping between schema tree T_1 and T_2, let S be a subset of pairs $(i, j) \in M$ with distinct word bags. Let D be the set of nodes in T_1 that are not mapped by M, and I be the set of nodes in T_2 that are not mapped by M. The mapping cost is given by $C = Sp + Iq + Dr$, where p, q and r are the costs assigned to the relabel, insertion, and removal operations according to the cost model proposed in section 4.3. We call C the *match distance* between T_1 and T_2, denoted as $C = ED(T_1, T_2)$. Match distance reflects semantic similarity of two schema trees.

Now let us see how to match web-service operations. Given two web-service operations $op_1 : s_1, s_2, ..., s_n \rightarrow t_1, t_2, ..., t_m$ and $op_2 : x_1, x_2, ..., x_l \rightarrow y_1, y_2, ..., y_k$,

for each schema tree of op_1, we find its corresponding schema tree of op_2 with the minimum match distance. We simply identify all possible matches between two lists of schema trees, and return the source-target correspondence that minimizes the overall match distance between the two lists. It does not depends on whether the number of parameters in the same or not between the two operations. We omit the algorithm details because of space limit.

5 Finding Associated Web-Service Operations

5.1 Clustering Web-Service Operations

Suppose $OP = \{op_1, op_2, ..., op_q\}$ is a set of web-service operations and each pair of operations op_i and op_j $(i, j = 1, 2, ..., q)$ match with the distance of z_{ij}. We classify OP into a set of clusters $\{op_{c1}, op_{c2}, ...\}$. The clustering algorithm is described as below. It begins with each operation forming its own cluster and gradually merges similar clusters.

1. Set up a match matrix $M_{q \times q}$. M_{ij} is the match distance of operation op_i and op_j.
2. Find the smallest M_{ij} in the match matrix M. If $M_{ij} <$ threshold δ then merge these two clusters and update M by replacing the two rows with a new row that describes the association between the merged cluster and the remaining clusters. The distance between two clusters is given by the distance between their closest members. There are now $q - 1$ clusters and $q - 1$ rows in M.
3. Repeat the merge step until no more clusters can be merged.

Finally, a set of clusters $\{OPC_1, OPC_2, ...\}$ is obtained. Given a cluster OPC_i and an operation $OPC_{ik} \in OPC_i$, OPC_{ik} is called a *pivot* of OPC_i if i t minimizes the sum of match distances to all the other operations in OPC_i. We consider all operations in OPC_i as *instances* of OPC_{ik}.

For example, in Fig. 1 we give a clustering result. There are two clusters of web-service operations. One is $\{WS1, WS2\}$, and the other is $\{WS3\}$. In cluster $\{WS1, WS2\}$ the pivot is *GetOrder* and the instrances of *GetOrder* are *GetOrder* and *OrderBuilder*. In cluster $\{WS3\}$ the pivot is *ShippingOrder*, which is also an instance of itself.

5.2 Identifying Associations

A set of web-service operations is said to be *associated* if they potentially contribute to a user's web-service composition. Clearly, given two web-service operations op_1 and op_2, if the output attributes of op_1 are similar to the input attributes of op_2 then op_1 and op_2 may participate in a user's service composition together. The objective of this step is to find all associations between web-service operations. To do this, we first find associations among clusters $\{OPC_1, OPC_2, ...\}$. Let OPC_{ik}, say $x_1, x_2, ..., x_k \rightarrow y_1, y_2, ..., y_j$ be a pivot of

OPC_i. Let $X = \{x_1, x_2, ..., x_k\}$ and $Y = \{y_1, y_2, ..., y_j\}$.We first compute the *attribute closure* X^+ with respect to X, which is the set of attributes A such that $X \rightarrow A$ can be inferred by transitivity. At the same time, a pivot set PS associated with OPC_{ik} is computed. The overall process is shown as algorithm 2. We perform a worst case time analysis of algorithm 2. The repeat loop is executed as most $|S|$ times, where $|S|$ is the total number of pivots corresponding to all clusters. The calculation of q takes time $|S| - |T|$, where T is the number of pivots in the pivot set PS. Hence the total execution time takes in the worst case time $O(S^2)$.

```
   input  : A pivot p : x_1, x_2, ..., x_k → y_1, y_2, ..., y_j
   output: A pivot set PS containing associated pivots
 1 X = {x_1, x_2, ..., x_k}; Y = {y_1, y_2, ..., y_j};
 2 Closure = X;
 3 PS = {X → Y};
 4 repeat
 5     if there is a pivot q : U → V such that the match distance of U and
       Closure is less than threshold δ then
 6         set Closure = Closure ⋃ V;
 7         set PS = PS ⋃ q;
 8     end
 9 until there is no change ;
```

Algorithm 2. Algorithm for computing attribute closure and pivot set

We first choose a pivot OPC_{ik} for each cluster OPC_i. For each pivot, we compute a pivot set. We eliminate duplicate pivot sets. If two pivots are in a same pivot set, then their corresponding instances are associated.

Each pivot set $PS = \{p_1, p_2, ..., p_k, ...\}$ can generate a set of operation groups in the form of $\{p'_1, p'_2, ..., p'_k, ...\}$, where p'_i is an instance of p_i. Operations in a same group are associated. To obtain an operation group, we simply replace each pivot p_i in PS with one of its corresponding instances. All possible operation groups are outputted as search results.

For example, a pivot set for the clusters given in Fig. 1 is $\{GetOrder, ShippingOrder\}$. It can generate two search results, one is $\{GetOrder, ShippingOrder\}$ and the other is $\{OrderBuilder, ShippingOrder\}$.

Recall that each candidate web-service operation is assigned a score indicating similarity to the given description. Thus, each operation group acquires a *group score* by counting the sum of operation scores in it. A higher group score indicates a more desirable search result, so the user's top-k requirements can be satisfied.

6 Experiments and Evaluations

We have implemented a prototype system, called WSXplorer, and conducted some experiments to evaluate the effectiveness and efficiency. The data set used

in our tests is a group of web services collected from [25,26,27]. Their WSDL specifications are available so we can obtain the textual descriptions and XML schemas of input/output data types. The data contains 223 web services including 930 web-service operations. We chose 7 web-service operations from three domains: *order*(3), *travel*(2) and *finance*(2). Each operation description was used as the basis for desired operations.

We used *recall* and *precision* ratio to evaluate the effectiveness of our approach. The precision(p) and recall(r) are defined as $p = \frac{A}{A+B}, r = \frac{A}{A+C}$ where A stands for the number of returned relevant operations, B stands for the number of returned irrelevant operations, C stands for the number of missing relevant operations, $A + C$ stands for the total number of relevant operations, and $A + B$ stands for the total number of returned operations. Specially, the top 100 search results were considered in our experiments for each web-service operation search.

We first evaluated the efficiency of WSXplorer by comparing the recall and precision of operation search with three other methods: keyword searching method, structure matching [9] and Woogle [13]. We computed the recall/precision ratio manully and plotted them in Fig. 5(a) and Fig. 5(b), respectively. As can be seen, the precisions of WSXplorer are 92%, 87% and 78% respectively, almost always outperforming that of keyword, structrure and Woogle. The precision is higher on *order* operations but lower in *finance* operations because *order* operations have more complex structures and richer semantics in input/output data types. This indicates that, by combining structural and semantic information, the precision of WSXplorer improves significantly, compared to the results obtained with structural or semantic information only. It is also can be seen that by keyword method the precision is rather low whereas the recall is rather high. This demonstrates textual description of operations contain much useful information but also much noise at the same time.

Then, we labeled the associated web-service operations in data set manually. The average recall/precision curve is used in Fig. 5(c) to evaluate the performance of WSXplorer on identifying associated operations. This figure illustrates that WSXplorer can achieve good recall and precision by integrating structural and semantic measurements.

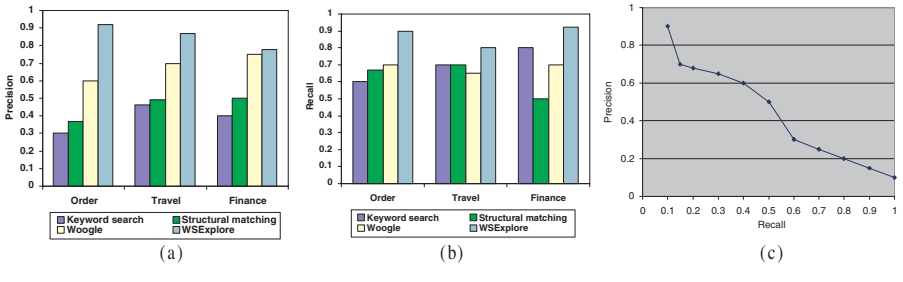

Fig. 5. Performance of WSXplorer

7 Conclusions

In this paper we have presented WSXplorer, a novel method to retrieve desired web-service operations of a given textual description. The concept of tree edit distance is employed to match web-service operations. Meanwhile, some algorithms are proposed for measuring and grouping similar operations. The proposed matching algorithm catches not only structures, but even better semantic information of schemas. We also introduced attribute closure for identifying associations between web-service operations. Our approach can be used for web-service searching tasks with top-k requirements.

As part of on-going work, we are interested in improving efficiency of the web-service operation matching algorithm in terms of running time, since the computation of extended tree edit distance is costly. Our proposed technique assumes structures of XML schema are trees. However, their structures may also be graphs and contain cycles. In the future, we plan to extend our algorithm to support graph matching. In order to further understand the semantics of web services descriptions and integrate more semantic information to our system, we also plan to use WordNet to handle word stems and synonyms to improve the precision of our algorithm.

References

1. Wang, H., Zhang, Y., Cao, J., Varadharajan, V.: Achieving secure and flexible M-services through tickets. IEEE Transactions on Systems, Man, and Cybernetics, Part A. 33(6), 697–708 (2003)
2. Bhiri, S., Perrin, O., Godart, C.: Ensuring required failure atomicity of composite Web services. In: Proc. of WWW Conference. pp.138–147 (2005)
3. Wang, H., Cao, J., Zhang, Y.: A Flexible Payment Scheme and Its Role-Based Access Control. IEEE Transactions on Knowledge and Data Engineering 17(3), 425–436 (2005)
4. Limthanmaphon, B., Zhang, Y.: Web Service Composition Transaction Management. In: Proceedings of Australasian Database Conference (ADC). pp.171–179 (2004)
5. Limthanmaphon, B., Zhang, Y.: Web Service Composition with Case-Based Reasoning. In: Proceedings of Australasian Database Conference (ADC). pp.201–208 (2003)
6. Zaremski, A.M., Wing, J.M.: Specification Matching of Software Components. ACM Trans. Softw. Eng. Methodol. 6(4), 333–369 (1997)
7. Booth, D., Haas, H., McCab, F., Newcomer, E., Champion, M., Ferris, C., Orchard, D.: Web Services Architecture. (2004) http://www.w3.org/TR/ws-arch/
8. Sajjanhar, A., Hou, J., Zhang, Y.: Algorithm for Web Services Matching. In: Proc. of Asia-Pacific Web. Conference (APWeb) 3007, 665–670 (2004)
9. Wang, Y., Stroulia, E.: Flexible Interface Matching for Web-Service Discovery. In: Proc. of International Conference on Web Information Systems Engineering (WISE) (2003)
10. Sycara, K.P., Widoff, S., Klusch, M., Lu, J.: Larks: Dynamic Matchmaking Among Heterogeneous Software Agents in Cyberspace. Autonomous Agents and Multi-Agent Systems 5(2), 173–203 (2002)

11. Roman, D., Lausen, H., Keller, U.: Web Service Modeling Ontology (WSMO). WSMO Final Draft 10 (2005)
12. Shen, Z., Su, J.: Web service discovery based on behavior signatures. In: Proc. of International Conference on Services Computing (SCC) 1, 279–286 (2005)
13. Dong, X., Halevy, A.Y., Madhavan, J., Nemes, E., Zhang, J.: Simlarity Search for Web Services. In: Proc. of International Conference on Very Large Data Bases (VLDB). pp. 372–383 (2004)
14. Hao, Y., Zhang, Y.: Web Services Discovery Based on Schema Matching. In: Proceedings of Australasian Computer Science Conference (ACSC) (2007)
15. Pu, K., Hristidis, V., Koudas, N.: Syntactic Rule Based Approach to Web Service Composition. In: Proc. of International Conference on Data Engineering (ICDE). vol.31 (2006)
16. Hristidis, V., Gravano, L., Papakonstantinou, Y.: Efficient IR-Style Keyword Search over Relational Databases. In: Proc. of VLDB. pp.850–861 (2003)
17. Hristidis, V., Papakonstantinou, Y.: DISCOVER: Keyword Search in Relational Databases. In: Proc. of VLDB). pp.670–681 (2002)
18. Agrawal, S., Chaudhuri, S., Das, G.: DBXplorer: A System for Keyword-Based Search over Relational Databases. In: Proc. of International Conference on Data Engineering (ICDE) (2002)
19. Salton, G.: The SMART Retrieval System - Experiments in Automatic Document Processing. Prentice-Hall, Inc, Upper Saddle River, NJ, USA (1971)
20. Salton, G., Wong, A., Yang, C.S.: A Vector Space Model for Automatic Indexing. Communications of the ACM (CACM) 18(11), 613–620 (1975)
21. Reis, D.D.C., Golgher, P.B., d.Silva, A.S., Laender, A.H.F.: Automatic web news extraction using tree edit distance. In: Proc. of WWW Conference. pp.502–511 (2004)
22. Zhang, K., Shasha, D.: Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems. SIAM Journal on Computing 18(6), 1245–1262 (1989)
23. Xie, T., Sha, C., Wang, X., Zhou, A.: Approximate Top-k Structural Similarity Search over XML Documents. In: Proc. of Asia-Pacific Web. Conference (APWeb) 3841, 319–330 (2006)
24. Kaufman, L., Rousseeuw, P.J.: Finding Groups in Data: An Introduction to Cluster Analysis. John Wiley, New York (1990)
25. http://www.xmethods.org. (XMethod)
26. http://www.bindingpoint.com. (BindingPoint)
27. http://www.webservicelist.com. (WebServiceList)