UNIVERSITY
OF TRENTO

DEPARTMENT OF INFORMATION AND COMMUNICATION TECHNOLOGY

38050 Povo – Trento (Italy), Via Sommarive 14
http://www.dit.unitn.it

STRUCTURE PRESERVING SEMANTIC MATCHING

Fausto Giunchiglia, Mikalai Yatskevich and Fiona McNeill

May 2007

Technical Report # DIT-07-028

Structure preserving semantic matching

Fausto Giunchiglia[1], Mikalai Yatskevich[1], and Fiona McNeill[2]

[1] Dept. of Information and Communication Technology,
University of Trento,
38050 Povo, Trento, Italy
{fausto,mikalai.yatskevich}@dit.unitn.it
[2] School of Informatics,
University of Edinburgh,
EH8 9LE, Scotland
f.j.mcneill@ed.ac.uk

Abstract. The most common matching applications, e.g., ontology matching, focus on the computation of the correspondences holding between the nodes of graph structures (e.g., concepts in two ontologies). However there are applications, such as matching of web service descriptions, where matching may need to compute the correspondences holding among the full graph structures and preserve certain structural properties of the graphs being considered. The goal of this paper is to provide an implementation of a new matching operator, that we call *structure preserving match*. This operator takes two graph-like structures and produces a mapping between those nodes of the structures that correspond semantically to each other, (i) still preserving a set of structural properties of the graphs being matched, (ii) only in the case if the graphs *globally* correspond semantically to each other. We present an exact and an approximate structure matching algorithms. The latter is based on a formal theory of abstraction and builds upon the well known tree edit distance measures. We have implemented the algorithms and applied them to the web service matchmaking scenario. The evaluation results, though preliminary, show the efficiency and effectiveness of our approach.

1 Introduction

We are interested in the problem of location of web services on the basis of the capabilities that they provide. This problem is often referred as matchmaking problem, see [14, 15, 21] for some recent examples. Most previous solutions employ a single ontology approach, namely the web services are assumed to be described by the concepts taken from a shared ontology. This allows to reduce the matchmaking problem to reasoning within the shared ontology. In contrast to this work, as described in [19, 6], we assume that the web services are described taking the terms from different ontologies and that their behavior is described using complex terms, actually first order terms. This allows us to provide very detailed description of their input and output behavior. The problem becomes therefore that of matching two web service descriptions (that can be seen as graph structures) and the mapping is considered as successful only if the two graphs are *globally* similar (e.g., $tree_1$ is 0.7 similar to $tree_2$, according to some metric). A further requirement of these applications is that the mapping must preserve certain structural

properties of the graphs being considered. In particular the syntactic types and sorts have to be preserved (e.g., a function symbol must be mapped to a function symbol and a variable must be mapped to a variable). At the same time we would like to enable the matchmaking of the web service descriptions that match only approximately (see [6] for a detailed description). For instance, *get_Wine(Region, Country, Color, Price, Number_of_bottles)* can be (approximately) mapped to *get_Wine(Region(Country, Area), Colour, Cost, Year, Quantity)*.

In this paper we define an operator that we call *structure preserving match*. This operator takes two graph-like structures and produces a mapping between those nodes of the structures that correspond semantically to each other, (i) still preserving a set of structural properties of the graphs being matched, (ii) only in the case if the graphs *globally* correspond semantically to each other. Notice that this problem significantly differs from the ontology matching problem, as defined for instance in [8, 23], where (i) is only partially satisfied and (ii) is not even an argument. We present an exact and an approximate structure matching algorithms. The former solves the exact structure matching problem. It is designed to succeed on equivalent terms and to fail otherwise. The latter solves an approximate structure matching problem. It is based on the fusion of the ideas derived from the theory of abstraction [7] and tree edit distance algorithms [28, 3]. We have implemented the algorithms and evaluated them on the dataset constructed from different versions of the state of the art first order ontologies.

The rest of the paper is organized as follows. We present a motivating example in Section 2. Section 3 is devoted to the exact structure matching algorithm. In Section 4 we define the abstraction operations and introduce the correspondence between them and tree edit operations. In Section 5 we show how existing tree edit distance algorithms can be exploited for the computation of the global similarity between two web service descriptions. Section 6 is devoted to approximate structure matching algorithm. The evaluation results are presented in Section 7. These results, though preliminary, illustrate the high efficiency and effectiveness of our approach. Section 8 briefly reviews the related work and concludes the paper.

2 A Motivating Example

Figure 1 provides an example of exactly matched web service descriptions along with their tree representations (or term trees). Dashed lines stand for the correspondences holding among the nodes of the term trees.

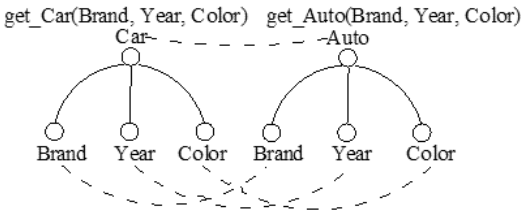

Fig. 1. Exactly matched web service descriptions and their tree representations.

In particular, in Figure 1 we have an exact match, namely the first of the services requires the second to return *Cars* of a given *Brand*, *Year* and *Color* while the other provides *Autos* of a given *Brand*, *Year* and *Color*. Notice that, there are no structural differences and that the only difference is in the function names.

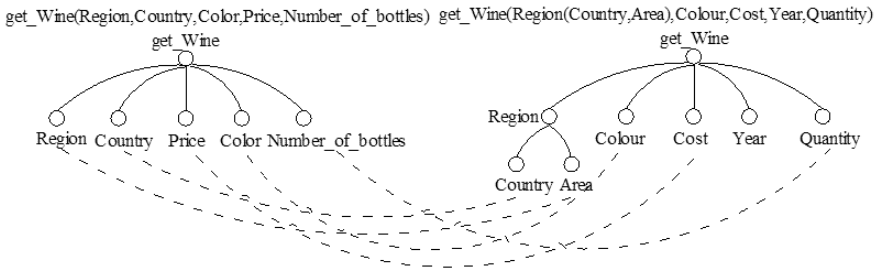

Fig. 2. Approximately matched web service descriptions and their tree representations.

Consider now Figure 2. It provides an example of an approximate match. In this case a more sophisticated data translation is required. For example, the first web service description requires that the fourth argument of *get_Wine* function (*Color*) to be mapped to the second argument (*Colour*) of *get_Wine* function in the second description. On the other hand *Region* on the right is defined as a function with two arguments (*Country* and *Area*) while on the left *Region* is an argument of *get_Wine*. Thus, *Region* in the first web service description must be passed to the second web service as the value of the *Area* argument of the *Region* function. Moreover *Year* on the right has no corresponding term on the left.

Therefore, in order to guarantee the successful data translation we are interested in the correspondences holding among the nodes of the term trees of the given web service descriptions only in the case when the web service descriptions themselves are "similar enough". At the same time the correspondences have to preserve the certain structural properties of the descriptions being matched. In particular we require the functions to be mapped to functions and variables to variables.

3 Exact structure semantic matching

There are two stages in the matching process:

- *Node matching*: solves the semantic heterogeneity problem by considering only labels at nodes and domain specific contextual information of the trees. In our approach we use semantic matching as extensively described in [8]. Notice that the result of this stage is the set of correspondences holding between the nodes of the trees.
- *Structural tree matching*: exploits the results of the node matching and the structure of the tree to find the correspondences holding between the trees themselves (e.g., $tree_1$ is 0.7 similar to $tree_2$).

Let us consider them in turn. The semantic node matching algorithm, as introduced in [8], takes as input two term trees and computes as output a set of correspondences holding among the nodes in the trees. This process is articulated in four macro steps:

Step 1. In this step we automatically translate natural language labels taken from the term tree elements into an internal logical language with boolean semantics. Thus, for example, the label *Number of bottles* is translated into $C_{Number\ of\ bottles} = C_{Number} \sqcap C_{bottles}$, where $C_{bottles} = \langle bottle, senses_{WN} \# 4 \rangle$ is taken to be the union of four WordNet senses, and similarly for *number*.

Step 2. Term trees are hierarchical structures where the path from the root to a node uniquely identifies that node (and also its meaning). We compute the logical formula associated with a node as a conjunction of concepts of the labels in the path from the given node to the root. For example, in Figure 2, the concept at node for the node *Area* is computed as follows: $C_{Area} = C_{get\_Wine} \sqcap C_{Region} \sqcap C_{Area}$.

Step 3 deals with acquisition of "world" knowledge. For example, from WordNet we can derive that *Region* and *Area* are synonyms, and therefore, $C_{Region} = C_{Area}$.

Step 4 deals with the computation of the semantic relations holding between two nodes. This is done by reducing this problem to a propositional satisfiability (SAT) problem and by exploiting state of the art SAT decider.

The exact structure matching algorithm exploits the results of the node matching algorithm. It is designed to succeed for equivalent terms and to fail otherwise. It expects the trees to have the same depth and the same number of children. More precisely we say that two trees T_1 and T_2 match iff for any node n_{11} in T_1 there is a node n_{21} in T_2 such that

- n_{11} semantically matches n_{21}, which in this case holds iff $c@n_{21}$ is equivalent to $c@n_{22}$ given the available background knowledge, where $c@n_1$ and $c@n_2$ are the concepts at nodes of n_1 and n_2;
- n_{11} and n_{21} reside on the same depth in T_1 and T_2, respectively;
- all ancestors of n_{11} are semantically matched to the ancestors of n_{21};

The pseudo code in Figure 3 illustrates an algorithm for exact structure matching. **exactStructureMatch** takes two trees of nodes *source* and *target* as an input. **exactStructureMatch** returns an array of *MappingElements* holding between the nodes of the trees if there is an exact match between them and null otherwise. The array of *MappingElements result* is created (line 12) and filled by **exactTreeMatch** (line 13). **allNodesMapped** checks whether all the nodes of *source* tree are mapped to the nodes of the *target* tree (line 14). If this is the case there is an exact structure match between the trees and the set of computed mappings is returned (line 15). **exactTreeMatch** takes two trees of nodes *source* and *target* and array of MappingElements *result* as an input. It recursively fills *result* with the mappings computed by **nodeMatch** (line 23). **exactTreeMatch** starts from obtaining the roots of *source* and *target* trees (lines 19-20). The semantic relation holding between them is computed by **nodeMatch** (line 21) implementing the node matching algorithm. If the relation is equivalence, the corresponding mapping is saved to *result* array (lines 22-23) and the children of the root nodes are obtained (line 26-27). Finally the loops on *sourceChildren* and *targetChildren* (lines 28-32) allow to call **exactTreeMatch** recursively for all pairs of sub trees rooted at *sourceChildren* and *targetChildren* elements.

```
1.Node struct of
2.  int nodeId;
3.  String label;
4.  String cLabel;
5.  String cNode;

6.MappingElement struct of
7.  int MappingElementId;
8.  Node source;
9.  Node target;
10. String relation;

11.MappingElement[] exactStructureMatch (Tree of Nodes source,target)
12. MappingElement[] result;
13. exactTreeMatch(source,target,result);
14. if (allNodesMapped(source,target,result))
15.  return result;
16. else
17.  return null;

18.void exactTreeMatch(Tree of Nodes source,target,MappingElement[] result)
19. Node sourceRoot=getRoot(source);
20. Node targetRoot=getRoot(target);
21. String relation= nodeMatch(sourceRoot,targetRoot);
22. if (relation=="=")
23.  addMapping(result,sourceRoot,targetRoot,"=");
24. else
25.  return;
26. Node[] sourceChildren=getChildren(sourceRoot);
27. Node[] targetChildren=getChildren(targetRoot);
28. For each sourceChild in sourceChildren
29.  Tree of Nodes sourceChildSubTree=getSubTree(sourceChild);
30.  For each targetNode in target
31.   Tree of Nodes targetChildSubTree=getSubTree(targetChild);
32.   exactTreeMatch(sourceChildSubTree, targetChildSubTree, nodesToMatch);
```

Fig. 3. Pseudo code for exact structure matching algorithm

4 Approximate matching via abstraction/refinement operations

In [7], Giunchiglia and Walsh categorize the various kinds of abstraction operations in a wide-ranging survey. They also introduce a new class of abstractions, called TI-abstractions (where TI means “Theorem Increasing”), which have the fundamental property of maintaining completeness, while losing correctness. In other words any fact which is true of the original term is also true of the abstract term, but not viceversa. And similarly, if a ground formula is true so is the abstract formula, but not vice versa. Dually, by taking the inverse of each abstraction operation, we can define a corresponding refinement operation which preserves correctness while loosing completeness. The second fundamental property of the abstraction operations is that they provide all and only the possible ways in which two first order terms can be made to differ by manipulations of their signature, still preserving completeness. In other words, this set of abstraction/refinement operations defines all and only the possible ways in which correctness and completeness are maintained when operating on first order terms and atomic formulas. This is the fundamental property which allows us to study and consequently quantify the semantic similarity (distance) between two first order terms. To this extent it is sufficient to determine which abstraction/refinement operations are necessary to convert one term into the other and to assign to each of them a cost that models the “semantic distance” associated to the operation.

Giunchiglia and Walsh's categories are as follows:

Predicate: Two or more predicates are merged, typically to the least general generalization in the predicate type hierarchy, e.g.,
– *Bottle(X) + Container(X)* $\mapsto$ *Container(X)*.
We call *Container(X)* a predicate abstraction of *Bottle(X)* or $Container(X) \sqsupseteq_{Pd} Bottle(X)$. Conversely we call *Bottle(X)* a predicate refinement of *Container(X)* or $Bottle(X) \sqsubseteq_{Pd} Container(X)$.

Domain: Two or more terms are merged, typically by moving the functions (or constants) to the least general generalization in the domain type hierarchy, e.g.,
– *Daughter(Me) + Child(Me)* $\mapsto$ *Child(Me)*.
– *Acura + Nissan* $\mapsto$ *Nissan*.
Similarly to the previous item we call *Child(Me)* and *Nissan* a domain abstractions of *Daughter(Me)* and *Acura* respectively or $Child(Me) \sqsupseteq_D Daughter(Me)$, $Nissan \sqsupseteq_D Acura$. Conversely we call *Daughter(Me)* and *Acura* a domain refinements of *Child(Me)* and *Nissan* or $Daughter(Me) \sqsubseteq_D Child(Me)$, $Acura \sqsubseteq_D Nissan$.

Propositional: One or more arguments are dropped, e.g.,
– *Bottle(A)* $\mapsto$ *Bottle*.
We call *Bottle* a propositional abstraction of *Bottle(A)* or $Bottle \sqsupseteq_P Bottle(A)$. Conversely *Bottle(A)* is a propositional refinement of *Bottle* or $Bottle(A) \sqsubseteq_P Bottle$.

Precondition: The precondition of a rule is dropped[3] , e.g.,
– *[Ticket(X)* $\rightarrow$ *Travel(X)]* $\mapsto$ *Travel(X)*.

Consider the following pair of first order terms *(Bottle A)* and *(Container)*. In this case there is no abstraction/refinement operation that make them equivalent. However consequent applications of propositional and predicate abstraction operations make the two terms equivalent:

$$(Bottle\ A) \mapsto^{\sqsubseteq_P} (Bottle) \mapsto^{\sqsubseteq_{Pd}} (Container) \quad (1)$$

In fact the relation holding among the terms is a composition of two refinement operations, namely $(Bottle\ A) \sqsubseteq_P (Bottle)$ and $(Bottle) \sqsubseteq_{Pd} (Container)$. We define an *abstraction mapping element (AME)* as a 5-tuple $\langle ID_{ij}, t_1, t_2, R, sim \rangle$, where ID_{ij} is a unique identifier of the given mapping element; t_1 and t_2 are first order terms; R specifies a relation for the given terms; and sim stands for a similarity coefficient in the range [0..1] quantifying the strength of the relation. In particular for the AMEs we allow the following semantic relations $\{\equiv, \sqsubseteq, \sqsupseteq\}$, where $\equiv$ stands for equivalence; $\sqsupseteq$ represents an abstraction relation and connects the precondition and the result of a composition of arbitrary number of predicate, domain and propositional abstraction operations; and $\sqsubseteq$ represents a refinement relation and connects the precondition and the result of a composition of arbitrary number of predicate, domain and propositional refinement operations.

[3] We do not consider precondition abstraction and refinement in the rest of this paper as we do not want to drop preconditions, because this would endanger the successful matchmaking of web services.

Therefore, the problem of AME computation becomes a problem of minimal cost composition of the abstraction/refinement operations allowed for the given relation R that are necessary to convert one term into the other. In order to solve this problem we propose to represent abstraction/refinement operations as tree edit distance operations applied to the term trees. This allows to redefine the problem of AME computation into a tree edit distance problem.

In its traditional formulation, the tree edit distance problem considers three operations: (i) vertex deletion, (ii) vertex insertion, and (iii) vertex replacement [25]. Often these operations are presented as rewriting rules:

$$(\mathrm{i})\upsilon \rightarrow \lambda;\ (\mathrm{ii})\lambda \rightarrow \upsilon;\ (\mathrm{iii})\upsilon \rightarrow \omega; \tag{2}$$

where υ and ω correspond to the labels of nodes in the trees while λ stands for the special blank symbol. Figure 4 illustrates two applications of delete and replace tree edit operations.

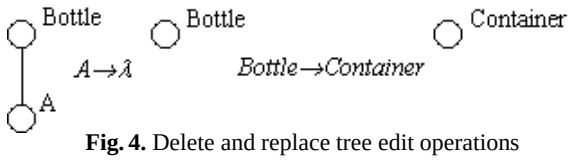

Fig. 4. Delete and replace tree edit operations

Our proposal is to restrict the formulation of the tree edit distance problem in order to reflect the semantics of the first order terms. In particular we propose to redefine the tree edit distance operations in such a way that will allow them to have one-to-one correspondence to the abstraction/refinement operations presented previously in this section. Table 1 illustrates the correspondence between abstraction/refinement and tree edit operations. The first column presents the abstraction/refinement operations. The second column lists corresponding tree edit operations. The third column describes the preconditions of the tree edit operation use. Consider, for example, the first line of Table 1. The predicate abstraction operation applied to first order term t_1 results with term t_2 ($t_1 \sqsupseteq_{Pd} t_2$). This abstraction operation corresponds to tree edit replacement operation applied to term tree of t_1 that replaces the node *a* with the node *b* ($a \rightarrow b$). Moreover the operation can be applied only in the case if (i) label *a* is a generalization of label *b* and (ii) both the nodes *a* and *b* in the term trees correspond to predicates in the first order terms.

Table 1. The correspondence between abstraction/refinement operations and tree edit operations.

Abstraction/ refinement operation	Tree edit operation	Preconditions of operation use
$t_1 \sqsupseteq_{Pd} t_2$	$a \rightarrow b$	$a \sqsupseteq b$; *a* and *b* correspond to predicates
$t_1 \sqsupseteq_{D} t_2$	$a \rightarrow b$	$a \sqsupseteq b$; *a* and *b* correspond to functions or constants
$t_1 \sqsupseteq_{P} t_2$	$\lambda \rightarrow a$	*a* corresponds to predicate, function or constant
$t_1 \sqsubseteq_{Pd} t_2$	$a \rightarrow b$	$a \sqsubseteq b$; *a* and *b* correspond to predicates
$t_1 \sqsubseteq_{D} t_2$	$a \rightarrow b$	$a \sqsubseteq b$; *a* and *b* correspond to functions or constants
$t_1 \sqsubseteq_{P} t_2$	$a \rightarrow \lambda$	*a* corresponds to predicate, function or constant

5 Computing the global similarity between two trees

Our goal now is to compute the similarity between two term trees. In order to perform this we need to compute the minimal cost composition of the abstraction/refinement operations that are necessary to convert one term tree/first order term into the other. The starting point is the traditional formulation of the tree edit distance problem.

$$Cost = \sum_{i \in S} n_i * Cost_i \tag{3}$$

The solution of the problem then becomes to minimize *Cost* in Eq. 3 and, therefore, to determine the minimal set of operations (i.e., the one with the minimum cost) which transforms one tree into another. In Eq. 3 S stands for the set of the allowed tree edit operations; n_i stands for the number of i-th operations necessary to convert one tree into the other and $Cost_i$ defines the cost of the i-th operation. Our goal is to define the $Cost_i$ in a way to model the semantic distance.

A possible uniform proposal is to assign the same unit cost to all tree edit operations that, as from Table 1, have their abstraction theoretic counterparts. Table 2 illustrates the costs of the abstraction/refinement (tree edit) operations, depending on the relation (equivalence, abstraction or refinement) being computed. Notice that the costs for estimating abstraction ($\sqsubseteq$) and refinement ($\sqsupseteq$) relations in AME have to be adjusted according to their definitions. In particular the tree edit operations corresponding to abstraction/refinement operations that are not allowed by the definition of the given relation have to be prohibited by assigning to them an infinite cost. Notice also that, we do not give any preference to a particular type of abstraction/refinement operations. Of course this strategy can be changed to satisfy certain domain specific requirements.

Table 2. Costs of the abstraction/refinement (tree edit) operations, exploited for computation of equivalence ($Cost_{\equiv}$), abstraction ($Cost_{\sqsubseteq}$) and refinement ($Cost_{\sqsupseteq}$) relations holding among the terms.

Abstraction/refinement (tree edit) operation	$Cost_{\equiv}$	$Cost_{\sqsubseteq}$	$Cost_{\sqsupseteq}$
$t_1 \sqsupseteq_{Pd} t_2$	1	∞	1
$t_1 \sqsupseteq_{D} t_2$	1	∞	1
$t_1 \sqsupseteq_{P} t_2$	1	∞	1
$t_1 \sqsubseteq_{Pd} t_2$	1	1	∞
$t_1 \sqsubseteq_{D} t_2$	1	1	∞
$t_1 \sqsubseteq_{P} t_2$	1	1	∞

Consider, for example, the first line in Table 2. The cost of the tree edit distance operation that correspond to the propositional abstraction ($t_1 \sqsupseteq_{Pd} t_2$) is equal to 1 when used for the computation of equivalence ($Cost_{\equiv}$) and abstraction ($Cost_{\sqsupseteq}$) relations in AME. It is equal to ∞ when used for the computation of refinement ($Cost_{\sqsubseteq}$) relation.

Eq. 3 can now be used for computation of the tree edit distance score. However, when comparing two web service descriptions we are interested rather in similarity than in distance. We exploit the following equation to convert the distance produced by

an edit distance algorithm into the similarity score:

$$sim = 1 - \frac{Cost}{max(number\_of\_nodes_1, number\_of\_nodes_2)} \quad (4)$$

where $number\_of\_nodes_1$ and $number\_of\_nodes_2$ stand for the number of nodes in the trees. Note that for the special case of *Cost* equal to ∞ the similarity score is estimated to 0.

Many existing tree edit distance algorithms allow to keep track of the nodes to which a replace operation is applied. Therefore, as a result they allow to obtain not only the minimal tree edit cost but also a minimal cost mapping among the nodes of the trees. According to [25] this minimal cost mapping is (i) one-to-one; (ii) horizontal order preserving between sibling nodes; and (iii) vertical order preserving. For example, the mapping depicted in Figure 1 complies to all these requirements while the mapping depicted in Figure 2 violates (ii). In particular the third sibling *Price* on the left tree is mapped to the third sibling *Cost* on the right tree while the fourth sibling *Color* on the right tree is mapped to the second sibling *Colour* on the left tree.

For the tree edit distance operations depicted in Table 1 we propose to keep track of nodes to which the tree edit operations derived from the replace operation are applied. In particular we consider the operations that correspond to predicate and domain abstraction/refinement ($t_1 \sqsupseteq_{Pd}$, $t_1 \sqsubseteq_{Pd}$, $t_1 \sqsupseteq_D$, $t_1 \sqsubseteq_D$). This allows us to obtain a mapping among the nodes of the term trees with the desired properties (i.e., there is only one-to-one correspondences in the mapping). Moreover it complies to the structure preserving matching requirements namely functions are mapped to functions and variables are mapped to variables. This is the case because (i) predicate and domain abstraction/refinement operations do not convert, for example, a function into a variable and (ii) the tree edit distance operations, as from Table 1, have a one-to-one correspondence with abstraction/refinement operations.

At the same time a mapping returned by a tree edit distance algorithm preserves the horizontal order among the sibling nodes, but this is not desirable property for the data translation purposes. This is the case because the correspondences that do not comply to the horizontal order preservation requirements, like the one holding between *Colour* and *Color* on Figure 2, are not included in the mapping. However, as from Table 1, the tree edit operations corresponding to predicate and domain abstraction/refinement ($t_1 \sqsupseteq_{Pd}$, $t_1 \sqsubseteq_{Pd}$, $t_1 \sqsupseteq_D$, $t_1 \sqsubseteq_D$) can be applied only to those nodes of the trees whose labels are either generalizations or specializations of each other, as computed by the node matching algorithm. Therefore, given the mapping produced by the node matching algorithm we can always recognize the cases when the horizontal order between sibling nodes is not preserved and change the ordering of the sibling nodes to make the mapping horizontal order preserving. For example, swapping the nodes *Cost* and *Colour* in the right tree depicted on Figure 2 does not change the meaning of the corresponding term while allows the correspondence holding between *Colour* and *Color* on Figure 2 to be included in the mapping produced by a tree edit distance algorthm.

6 The approximate structure matching algorithm

As from above our goal is to find 'good enough' services [9] if perfect are not available. We start by providing a definition of the approximate structure matching as the basis for the algorithm.

We say that two nodes n_1 and n_2 in the trees T_1 and T_2 approximately match iff $c@n_1$ R $c@n_2$ holds given the available background knowledge, where $c@n_1$ and $c@n_2$ are the concepts at nodes of n_1 and n_2, and where $R \in \{\equiv, \sqsubseteq, \sqsupseteq, \wedge, \perp, not\ related\}$.

We say that two trees T_1 and T_2 match iff there is at least one node n_{11} in T_1 and a node n_{21} in T_2 such that

- n_{11} approximately matches n_{21};
- all ancestors of n_{11} are approximately matched to the ancestors of n_{21};

The approximate structure matching algorithm exploits the node matching algorithm presented in Section 3. First the approximate structure matching algorithm estimates the similarity of two terms by application of a tree edit distance algorithm with the tree edit operations and costs modified as described in Sections 4 and 5. The similarity scores are computed for equivalence, abstraction and refinement relations. For each of these cases the tree edit distance operation costs are modified as depicted on Table 2. The relation with the highest similarity score are assumed to hold among the terms. If the similarity score exceeds a given threshold the mappings connecting the nodes of the term trees, as computed by the tree edit distance algorithm, are returned by the matching routine what allows for further data translation.

Figure 5 illustrates approximate structure matching algorithm.

```
AME struct of
  Tree of Nodes source;
  Tree of Nodes target;
  String relation;
  double approximationScore;

1.MappingElement[] approximateStructureMatch
        (Tree of Nodes source, target, double threshold)
2. MappingElement[] result;
3. approximateTreeMatch(source,target,result);
4. AME ame=analyzeMismatches(source,target,result);
5. if (getRelation(ame)=="=") or (getRelation(ame)=="<")
                            or (getRelation(ame)==">")
6.  if (getApproximationScore(ame)>threshold)
7.   return result;
8. return null;
```

Fig. 5. Pseudo code for approximate structure matching algorithm

approximateStructureMatch takes as input the *source* and *target* term trees and a *threshold* value. **approximateTreeMatch** fills the *result* array (line 3) which stores the mappings holding between the nodes of the trees. In contrast to **exactTreeMatch** in Figure 3 **approximateTreeMatch** considers the semantic relations other than equivalence. An AME *ame* is computed (line 4) by **analyzeMismatches**. If *ame* stands for equivalence, abstraction or refinement relations (line 5) and if an *approximationScore* exceeds *threshold* (line 6) the mappings calculated by **approximateTreeMatch** are returned (line 7). **analyzeMismatches** calculates the aggregate score of tree match quality by exploiting a tree edit distance algorithm as described in Section 5.

7 Evaluation

We have implemented the algorithms described in the previous sections in Java. In the implementation we have exploited a modification of simple tree edit distance algorithm from Valiente's work [27]. We have evaluated the matching quality of the algorithms on 132 pairs of first order logic terms. Half of the pairs were composed of the equivalent terms (e.g., *journal(periodical-publication)* and *magazine (periodical-publication)*) while the other half were composed from similar but not equivalent terms (e.g., *web-reference(publication-reference)* and *thesis-reference (publication-reference)*). The terms were extracted from different versions of the Standard Upper Merged Ontology (SUMO)[4] and the Advance Knowledge Transfer (AKT)[5] ontologies. We extracted all the differences between versions 1.50 and 1.51, and 1.51 and 1.52 of the SUMO ontology and between versions 1, 2.1 and 2.2 of the AKT-portal and AKT-support ontologies[6]. These are both first order ontologies, so many of these differences mapped well to the potential differences between terms that we are investigating. However, some of them were more complex, such as differences in inference rules, or consisted of ontological objects being added or removed rather than altered, and had no parallel in our work. These pairs of terms were discarded and our tests were run on all remaining differences between these ontologies. Therefore, we have simulated the situation when the service descriptions are defined exploiting the two versions of the same ontology.

In our evaluation we have exploited the commonly accepted measures of matching quality, namely precision, recall, and F-measure. Precision varies in the [0,1] range; the higher the value, the smaller the set of incorrect correspondences (false positives) which have been computed. Precision is a correctness measure. Recall varies in the [0,1] range; the higher the value, the smaller the set of correct correspondences (true positives) which have not found. Recall is a completeness measure. F-measure varies in the [0,1] range. The version computed here is the harmonic mean of precision and recall. It is a global measure of the matching quality, increasing as the matching quality improves. While computing precision and recall we have considered the correspondences holding among first order terms rather than the nodes of the term trees. Thus, for instance, *journal(periodical-publication*$_1$*)=magazine(periodical-publication*$_2$*)* was considered as single correspondence rather than two correspondences, namely *journal=magazine* and *periodical-publication*$_1$*=periodical-publication*$_2$. The evaluation was performed on a Pentium 4 computer with 512 Mb of RAM.

Interestingly enough our exact structure matching algorithm was able to find 36 correct correspondences what stands for 54% of Recall with 100% Precision. All mismatches (or correct correspondences not found by the algorithm) corresponded to structural differences among first order terms which exact structure matching algorithm is unable to capture. The examples of correctly found correspondences are given below:

```
meeting-attendees(has-other-agents-involved)
meeting-attendee(has-other-agents-involved)
```

[4] `http://ontology.teknowledge.com/`

[5] `http://www.aktors.org`

[6] see `http://dream.inf.ed.ac.uk/projects/dor/` for full versions of these ontologies and analysis of their differences

```
r&d-institute(Learning-centred-organization)
r-and-d-institute(Learning-centred-organization)

piece(Pure2,Mixture)
part(Pure2,Mixture)

has-affiliatied-people(Affiliated-person)
has-affililated-person(affiliated-person)
```

The first and the second example illustrate the minor syntactic differences among the terms, while the third and fourth examples illustrate the semantic heterogeneity in the various versions of the ontologies.

Figure 6 presents the matching quality measures depending on the cut-off threshold value for approximate structure preserving matching algorithm. As from Figure 6, the

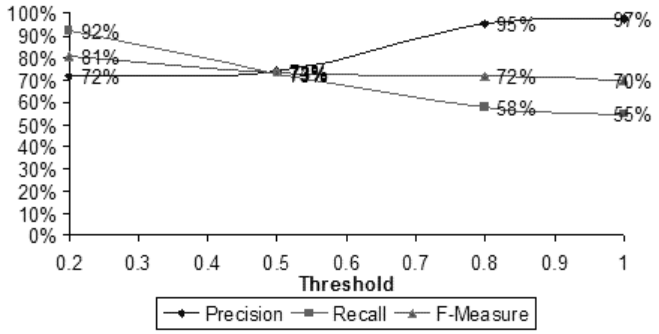

Fig. 6. The matching quality measures depending on threshold value for approximate structure matching algorithm

algorithm demonstrates high matching quality on the wide range of threshold values. In particular, F-Measure values exceed 70% for the given range. Table 3 summarizes the time performance of the matching algorithm. It presents the average time taken by

Table 3. Time performance of approximate structure matching algorithm (average on 132 term matching tasks)

	Node matching Step 1 and 2	Node matching Step 3 and 4	Structure matching
Time, ms	134.1	3.3	0.9

the various steps of the algorithm on 132 term matching tasks. As from the table, Step 1 and 2 of the node matching algorithm significantly slow down the whole process. However these steps correspond to the linguistic preprocessing that can be performed once offline [8]. Given that the terms can be automatically annotated with the linguistic preprocessing results [8] once when changed, the overall runtime is reduced to 4.2 ms, which corresponds roughly to 240 term matching tasks per second.

8 Conclusion and Related Work

We have presented an exact and an approximate structure matching algorithms that implement the *structure preserving match* operator. We have implemented the algorithms and applied them to the web service matchmaking scenario. The evaluation results, though preliminary, show the efficiency and effectiveness of our approach.

Future work includes further investigations on the cost assignment for the abstraction/refinement operations. In the version of the algorithm presented in the paper no preference is given to the particular abstraction/refinement operation and all allowed operations are assigned a unit cost. One may argue, for example, that the semantic distance between *cat* and *mammal* is less then the semantic distance between *cat* and *animal*. Therefore, the operation abstracting *cat* to *mammal* have to be less costly than the operation abstracting *cat* to *animal*.

The problem of location of web services on the basis of the capabilities that they provide (often referred as matchmaking problem) recently has received a considerable attention. Most of the approaches to the matchmaking problem so far employed a single ontology approach (i.e., the web services are assumed to be described by the concepts taken from the shared ontology). See [14, 15, 21] for example. Probably the most similar to ours is the approach taken in METEOR-S [1] and in [20] where the services are assumed to be annotated with the concepts taken from various ontologies. Then the matchmaking problem is solved by the application of the matching algorithm. The algorithm combines the results of atomic matchers that roughly correspond to the element level matchers exploited in the Step 3 of the node matching algorithm in Section 3. In contrast to this work we exploit a more sophisticated matching technique that allows us to utilize the context provided by the first order term.

Many diverse solutions to the ontology matching problem have been proposed so far. See [23] for a comprehensive survey and [5, 18, 16, 4, 22, 10, 2, 12, 17, 24] for individual solutions. However most of efforts were devoted to computation of the correspondences holding among the classes of description logic ontologies. Recently several approaches allowed computation of correspondences holding among the object properties (or binary predicates) [13, 26]. The approach taken in [11] allows to find correspondences holding among parts of description logic ontologies or subgraphs extracted from the ontology graphs. In contrast to these approaches we allow the computation of correspondences holding among first order terms.

References

1. R. Aggarwal, K. Verma, J. Miller, and W. Milnor. Constraint driven web service composition in METEOR-S. In *Proceedings of IEEE International Conference on Services Computing*, 2004.
2. S. Bergamaschi, S. Castano, and M. Vincini. Semantic integration of semistructured and structured data sources. *SIGMOD Record*, 28(1):54–59, 1999.
3. W. Chen. New algorithm for ordered tree-to-tree correction problem. *J. Algorithms*, 40(2):135–158, 2001.
4. M. Ehrig, S. Staab, and Y. Sure. Bootstrapping ontology alignment methods with APFEL. In *Proceedings of the International Semantic Web Conference (ISWC)*, pages 186–200, 2005.

5. J. Euzenat and P. Valtchev. Similarity-based ontology alignment in OWL-lite. In *Proceedings of European Conference on Artificial Intelligence (ECAI)*, pages 333–337, 2004.
6. F. Giunchiglia, F. McNeill and M. Yatskevich: Web Service Composition via Semantic Matching of Interaction Specifications. Technical Report DIT-06-080. November 2006.
7. F. Giunchiglia and T. Walsh. A theory of abstraction. *Artificial Intelligence*, 57(2-3):323–390, 1992.
8. F. Giunchiglia, M. Yatskevich, and P. Shvaiko. Semantic matching: Algorithms and implementation. *Journal on Data Semantics*, IX, 2007.
9. F. Giunchiglia and I. Zaihrayeu. Making peer databases interact - a vision for an architecture supporting data coordination. In *Proceedings of the International Workshop on Cooperative Information Agents (CIA)*, pages 18–35, 2002.
10. R. Gligorov, Z. Aleksovski, W. ten Kate, and F. van Harmelen. Using google distance to weight approximate ontology matches. In *Proceedings of the seventeenth World Wide Web conference WWW'17*, Korea, May 2007.
11. W. Hu and Y. Qu. Block matching for ontologies. In *Proceedings of ISWC*, 2006.
12. Y. Kalfoglou and M. Schorlemmer. If-map: an ontology mapping method based on information flow theory. *Journal on Data Semantics*, 1(1):98–127, October 2003.
13. J. Kim, M. Jang, Y. Ha, J. Sohn, and S. Lee. MoA: OWL ontology merging and alignment tool for the semantic web. In *Proceedings of the IEA/AIE*, pages 722 – 731, 2005.
14. M. Klusch, B. Fries, and K. Sycara. Automated semantic web service discovery with owls-mx. In *AAMAS*, pages 915–922, 2006.
15. L. Li and I. Horrocks. A software framework for matchmaking based on semantic web technology. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 331–339, New York, NY, USA, 2003. ACM Press.
16. V. Lopez, M. Sabou, and E. Motta. Powermap: Mapping the real semantic web on the fly. In *The Semantic Web - ISWC 2006*, pages 414–427, 2006.
17. P. Mitra, N. Noy, and A. Jaiswal. Ontology mapping discovery with uncertainty. In *Proceedings of the International Semantic Web Conference (ISWC)*, pages 537–547, 2005.
18. N. Noy and M. Musen. The PROMPT suite: interactive tools for ontology merging and mapping. *Int. J. Hum.-Comput. Stud.*, 59(6):983–1024, 2003.
19. OpenKnowledge Manifesto. http://www.openk.org/, 2006.
20. S. Oundhakar, K. Verma, K. Sivashanugam, A. Sheth, and J. Miller. Discovery of web services in a multi-ontology and federated registry environment. *International Journal of Web Services Research (JWSR)*, 2(3):1–32, 2005.
21. M. Paolucci, T. Kawamura, T. Payne, and K. Sycara. Semantic matching of web services capabilities. In *Proceedings of ISWC*, pages 333–347, 2002.
22. Y. Qu, W. Hu, and G. Cheng. Constructing virtual documents for ontology matching. In *Proceedings of WWW '06*, pages 23–31, New York, NY, USA, 2006. ACM Press.
23. P. Shvaiko and J. Euzenat. A survey of schema-based matching approaches. *Journal on Data Semantics*, IV, 2005.
24. U. Straccia and R. Troncy. oMAP: Combining classifiers for aligning automatically owl ontologies. In *Proceedings of the International Conference on Web Information Systems Engineering (WISE)*, pages 133–147, 2005.
25. K. Tai. The tree-to-tree correction problem. *J. ACM*, 26(3):422–433, 1979.
26. J. Tang, J. Li, B. Liang, X. Huang, Y. Li, and K. Wang. Using bayesian decision for ontology mapping. *Web Semant.*, 4(4):243–262, 2006.
27. G. Valiente. *Algorithms on Trees and Graphs*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.
28. J. Wang, B. Shapiro, D. Shasha, K. Zhang, and K. Currey. An algorithm for finding the largest approximately common substructures of two trees. *IEEE Trans. Pattern Anal. Mach. Intell.*, 20(8):889–895, 1998.