Object-Oriented Similarity Measures for Semantic Web Service Matchmaking

Georgios Meditskos and Nick Bassiliades
Aristotle University of Thessaloniki, Greece
{gmeditsk,nbassili}@csd.auth.gr

Abstract

The semantic annotation of Web services capabilities with ontological information aims at providing the necessary infrastructure for facilitating efficient and accurate service discovery. The main idea is to apply reasoning techniques over semantically enhanced Web service requests and advertisements in order to determine Web services that meet certain requirements. In this paper we present our work for introducing similarity measures inspired from the domain of Object-Oriented paradigm for ontology concept matching. Our work focuses on the utilization of such measures over an Object-Oriented schema that is created through mapping rules of OWL constructs and semantics into the Object-Oriented model. The goal of the approach is to combine the Object-Oriented representation of the information and the reasoning over OWL semantics in order to enhance the retrieval of semantically relevant, to some criteria, Web services.

1. Introduction

While the number of the deployed Web services on the Internet increases, the ability of automatically and accurately discovering one with a specific functionality becomes more and more important. The Web service discovery process incorporates not only the location of a single Web service that satisfies specific user's requirements but also it is a vital procedure during composition, i.e. the combination of more than one Web services in order to create a new service with enhanced functionality. In this case, the discovery process is performed in each step of the composition plan in order to determine candidate Web services that meet sub-requirements.

The combination of Semantic Web techniques with Web services seems the best way to give the appropriate semantic notion to Web services in order to achieve the desirable level of automation, leading to the notion of the semantic Web services. Languages and frameworks such as the OWL-S [15], WSMO [22] and WSDL-S [21] use ontologies for the description of Web services capabilities, enabling the utilization of ontology reasoning engines to process this information and to derive conclusions. In that way, queries for a specific functionality are semantically matched with Web services capabilities based on the language semantics, e.g. OWL [14], leading to semantically accurate results.

However, the matchmaking procedure based only on logic-based reasoning is restricted to the determination of the subsumption relationships among the concepts of the ontology. Thus, although the matchmaking results capture semantic information stemming from the ontology, ignore structural information of the schema which may enhance the retrieval of relevant concepts, unable to be captured by the reasoning process.

In this paper we define a methodology for the discovery of Web services by utilizing similarity measures inspired from the domain of the Object-Oriented (OO) paradigm. The main idea is to introduce techniques which are able not only to handle the subsumption relationships that derive from reasoning on OWL ontologies but also to capture useful hierarchical relationships, such as the indirect relationship between two sibling classes, in order to augment the matchmaking process.

The rest of the paper is organized as follows: in section 2 we describe the notion of similarity in the OO paradigm. In section 3 we present shortly the methodology of transforming OWL ontologies into the OO model. In section 4 we analyze the potential utilization of OO similarity measures during Web services matchmaking and we describe in detail our methodology. In sections 5 and 6 we give a feeling of the application of our methodology in simple use cases. In section 7 we give related work on the field of semantic Web service matchmaking and finally in section 8 we conclude giving future directions.

2. Similarity in the OO Paradigm

The notion of similarity in the OO paradigm is used mainly for determining fragments of duplicated code (clone detection), emphasizing on the reuse that is responsible for the increase in software quality and development productivity [8][13][18]. The algorithms are based on the definition of similarity measures that characterize class definitions at the level of source code, such as properties, packages, methods, lines of code (LOC) etc. and by using clustering or tree algorithms they manage to identify similar classes. Such measures have been also used in the domain of the case-based reasoning for determining similar solutions to similar problems [1][3][10].

To the best of our knowledge, only [1] defines object similarity measures that are stemmed directly from the class hierarchy. The goal is to determine the similarity between two objects, i.e. one object representing the case and one object representing the query, a paradigm very similar to the discovery of a Web service, where the "query object" is the query for a service with a specific functionality and the "case objects" are the advertisements of Web services. In their approach, they introduce the notion of the *intra-class* and *inter-class* similarity. The first is based on the common properties of the two objects whereas the latter on the positions of the object classes in the hierarchy. The overall similarity between a query object and a case object can be computed by defining an equation over the inter- and intra-class similarity.

Generally, the intra-class similarity between two objects is defined over the values of their common properties, i.e. it is computed over the properties of the most specific common superclass of the examined objects. This ensures that the properties being examined are inherited to both objects. During this procedure, a matching algorithm is applied in both *simple attributes*, i.e. integers or strings, and *relational attributes*, i.e. attributes that take objects as values. The overall intra-class similarity for two objects is derived from the aggregation of both similarities. Let k be the number of properties of the common class of two objects a and b with S_n and R_m being the simple and relational properties respectively ($n + m = k$), f_s the simple and f_r the relational property matching algorithm and A an aggregation function. Then, the intra-class similarity Sim_{intra} of the two objects can be defined as:

$$Sim_{intra}(a,b) = A\left(\mathop{\mathrm{H}}_{1}^{n} f_s\left(S_{n,a}, S_{n,b}\right), \mathop{\Theta}_{1}^{m} f_r\left(R_{m,a}, R_{m,b}\right) \right) \quad (1)$$

where H and Θ are aggregation functions for each matching function f_s and f_r respectively.

Contrary to the intra-class similarity, the inter-class similarity is defined upon the class hierarchy of the OO schema, without taking into account class properties. The idea is to define a similarity value that represents the hierarchical relationship between the classes of two objects, independently of the property values of the objects. The proposed method for defining the inter-class similarity in [1] distinguishes inner from leaf nodes. More specifically, it assigns a weight S_i to each inner class C_i such that $\forall C_i, C_k \mid C_k \sqsubseteq C_i : S_i \le S_k$ and the inter-class similarity for two objects of leaf classes is defined as:

$$Sim_{inter}(C_i, C_k) = \begin{cases} 1, & \text{if } C_i = C_k \\ S_c, & \text{otherwise} \end{cases} \quad (2)$$

where S_c is the S weight for the most specific common superclass of C_i and C_k. If one (or both) of the examined object classes is an inner class, then there are different cases, concerning class similarity. To give an example, if the query object belongs to a leaf class C and the case object to an inner class I, then the similarity is defined as:

$$Sim_{inter}(Q, I) = \begin{cases} 1, & \text{if } Q \sqsubseteq I \\ S_c, & \text{otherwise} \end{cases} \quad (3)$$

where S_c is the weight S for the most specific common superclass of the Q and I classes.

In that way, the overall similarity of two objects a and b can be defined by an aggregation function Φ over the intra- and inter-class similarities:

$$Sim(a,b) = \Phi\left(Sim_{intra}(a,b), Sim_{inter}(class(a), class(b))\right) \quad (4)$$

3. Transforming OWL to Object Model

The idea of applying OO similarity measures in the domain of Web service matchmaking is motivated by the OO OWL reasoning methodology we define in [12]. We have developed O-DEVICE[1], an extension of the well known CLIPS production rule engine [2] that handles a superset of OWL Lite ontologies close to OWL DL, following the OO model (as realized by the CLIPS OO Language COOL). In that way (a) we enable the implementation of OO rule programs over OWL, allowing complex systems to be modeled as modular components and (b) we use the resulting OO model as a simplistic form of indexing; we can have hierarchical relationships and property values of an object in one step by exploiting the functions and the message passing mechanism of CLIPS.

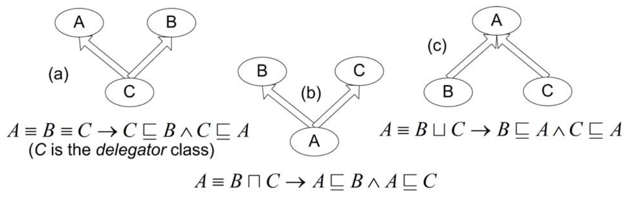
In an OWL ontology, classes and properties are defined as instances of appropriate built-in classes, e.g. `owl:Class` or `owl:ObjectProperty`. The system creates the objects that correspond to these OWL instances, which we call *meta-objects*, and it uses the information stored in their slots in order to create the classes and the properties in the OO model.

Each concept C of an ontology is mapped into a class in the OO model and each TBOX assertion $C \sqsubseteq D$ is mapped into a subclass relationship. A class with no explicit superclasses is defined as subclass of the `owl:Thing` class, the superclasses of all classes according to the OWL axioms. In that way we can use the OO environment's inheritance mechanisms in order to determine subsumption relationships.

The `owl:intersectionOf` construct is treated by defining multiple concurrent subclass relationships. If there is a class C defined as $C \equiv A_1 \sqcap A_2 \sqcap \dots A_n$, then we define $C \sqsubseteq A_k$, where $1 \le k \le n$, i.e. each A_k class becomes a direct superclass of class C and every object of class C is simultaneously an object of all A_k classes (Fig. 1 (b)). The `owl:unionOf` construct is also treated by defining subclass relationships. If there is a class C defined as $C \equiv A_1 \sqcup A_2 \sqcup \dots A_n$, then we define $A_k \sqsubseteq C$, where $1 \le k \le n$, i.e. each A_k class becomes a direct subclass of class C (Fig. 1

[1] http://lpis.csd.auth.gr/systems/o-device/o-device.html

(c)). Moreover, let there be a set of n equivalent classes C_n (n>1). The system selects randomly one of the n classes, e.g. class C_d to become the *delegator* class and defines it as a subclass of the rest of the classes, i.e. $C_d \sqsubseteq C_n$ where $n \neq d$ (Fig. 1 (a)). The complete semantics of these transformations can be found in [12].



(a)
A
B
C
$A \equiv B \equiv C \rightarrow C \sqsubseteq B \wedge C \sqsubseteq A$
(C is the delegator class)
(b)
B
C
A
$A \equiv B \sqcap C \rightarrow A \sqsubseteq B \wedge A \sqsubseteq C$
(c)
A
B
C
$A \equiv B \sqcup C \rightarrow B \sqsubseteq A \wedge C \sqsubseteq A$

Fig. 1. Class equivalence (a), intersection (b) and union (c).

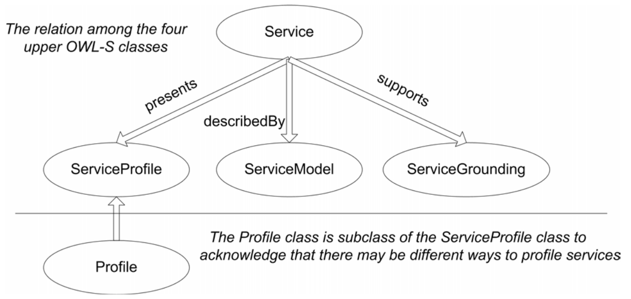
4. Applicability of OO Similarity Measures in the Domain of Web Service Discovery

In order to exploit the level of the object similarity knowledge that derives from the OO representation of the ontology information, we have investigated the possibility of applying the similarity measures defined in [1] in the domain of Web service discovery based on the OWL-S descriptions of services inputs and outputs.

Based on the transformation methodology we mentioned in the previous section, we create a native OO model of classes, properties and instances that reflects the class and property hierarchy that stems from the semantics of the OWL language. In the case of OWL-S Web service descriptions, both Web service advertisements and queries are defined as OWL instances of the *Profile* class, an OWL-S class that is used for defining Web services advertisements (Fig. 2). Thus, each Profile instance is mapped into an object of the corresponding OO Profile class and the matchmaking process can be performed by checking the similarity of the input and output property values of the profile objects. The goal is to perform matchmaking not based only on the subsumption relationships that are derived from the underlying rule reasoner but also to exploit the knowledge of similarity that stems from the OO representation.

One distinctive characteristic of the values of the I/O properties of the Profile class is that they denote the types of the I/O values of a query or a Web service and not the values themselves. This is realized by giving a URI that denotes an ontology class or a datatype (although the latter is not recommended since it does not give semantic information). Thus, the process of determining the similarity between two profile objects requires matching algorithms between datatypes or classes and not between actual values, e.g. numeric or object values, as the intra-class measure defines.

Moreover, since all the objects of our interest belong always to the same class (Profile), the inter-class measure cannot give useful information about class similarity.



The relation among the four upper OWL-S classes
Service
presents
describedBy
supports
ServiceProfile
ServiceModel
ServiceGrounding
Profile
The Profile class is subclass of the ServiceProfile class to acknowledge that there may be different ways to profile services

Fig. 2. The OWL-S upper ontology.

From equation (2), the inter-class similarity of two profile objects would always be 1, since $\forall C_i, C_k : C_i = C_k$ and therefore the equation (4) would only be based on the intra-class similarity of the objects.

Having identified the differences that restrict the direct application of the OO similarity measures in the domain of Web service discovery based on *direct* objects of the profile class, we change the definition of the intra- and inter-class similarity measures to fit our case. In the following, we use the term "simple" to refer to I/Os that denote datatypes and the term "relational" to refer to I/Os that denote class types.

We define the similarity S of two profiles as:

$$S(p_Q, p_W) = \Phi\left(f_s(IO_{Q,s}, IO_{W,s}), f_r(IO_{Q,r}, IO_{W,r}) \right) \quad (5)$$

where f_s is a function that computes the similarity over query and Web service simple I/O parameters and f_r a function that computes the similarity between query and Web service relational I/O parameters.

Equation (5) is analogous to equation (3) that defines the intra-class similarity. The profile similarity is defined by checking the I/O property values of a query and a Web service profile object and according to the referred values, the simple or relational similarity function is used. However, in order to take into account hierarchical information in the similarity function, we consider as f_r the inter-class similarity of the relational parameters. Intuitively, the similarity of two profile objects is determined by an "intra-class" measure that is calculated based also on the hierarchical information (inter-class) in order to determine the similarity of the classes referred in the I/Os.

In the following we assume that $I_{Q,s}$, $O_{Q,s}$, $I_{W,s}$ and $O_{W,s}$ are the sets of query and Web service simple I/Os and $I_{Q,r}$, $O_{Q,r}$, $I_{W,r}$ and $O_{W,r}$ are the sets of query and Web service relational I/Os, respectively.

4.1 Simple Property Similarity

The similarity of I/O parameters that refer to datatypes can be determined by checking directly the URIs. In our analysis, we consider only simple numerical datatypes, i.e. `xsd:int`, `xsd:float`, etc., Boolean and string datatypes. We define three levels of similarity:

- *exact match*: Two datatypes are matched exactly if they refer to the same type.

- *numerical type match*: Two numerical datatypes have always a type match, e.g. `xsd:int` and `xsd:float` are type matched.
- *mismatch*: Two datatypes have a mismatch if there is not an exact or numerical match, e.g. between the `xsd:int` and the `xsd:boolean` datatypes.

Algorithm 1 describes the f_s function that performs datatype checking between a query and an advertisement simple input parameters. The same function can be used for simple output values by reversing the order of the parameters, i.e. $f_s(O_{W,s}, O_{Q,s})$. The f_s function determines the number of matches (exact or numerical) and it gives a rank value by aggregating the counts of matches giving different weights to different types.

Algorithm 1: The f_s function for simple inputs.
Inputs: The $I_{Q,s}$ and $I_{W,s}$ sets.
Output: A rank value.

```
function $f_s(I_{Q,s}, I_{W,s})$ {
  var $exM := numM := 0$
  var $N :=$ size($I_{W,s}$), $A := 1$, $B := A / 2$
  if $N = 0$ then return 1
  for each $d \in I_{W,s}$ do
    if $\exists d' \in I_{Q,s} \wedge$ exact($d$, $d'$) then
      $exM := exM + 1$
    else if $\exists d' \in I_{Q,s} \wedge$ numerical($d$, $d'$) then
        $numM := numM + 1$
      else return 0 //mismatch
  return $(A * exM + B * numM) / N$
}
```

The algorithm uses a simple aggregation function that calculates the rank value as the weighted sum of the matches divided by the number of advertisement inputs. If there are only exact matches among all inputs, then the returned value is 1 since $exM = N$ and $numM = 0$. If there are only numerical matches then $exM = 0$ and $numM = N / 2$ and the similarity equals to 0.5. In any other case, the rank is a value between 0.5 and 1. The rank value equals to 0 whenever a mismatch is detected.

4.2 Relational Property Similarity

The similarity of I/O parameters that refer to class types can be determined using the inter-class similarity measure. In the OO paradigm, the inter-class measure is used in order to determine the similarity of two classes according to their position in the hierarchy. Thus, it is feasible to use such a measure in order to find the similarity of the classes referred by the I/O profile properties.

In order to realize the inter-class algorithm, each inner class of the hierarchy should be assigned with an S weight, as we have mentioned in section 2. However, the algorithm for weight assignment and for determining the inter-class similarity proposed in [1] cannot constitute an absolute measure for defining class similarity. For example, consider the class hierarchy of Fig. 3. According to equation (3), the similarity among a query object of the class B and a case object of the class C should be S_A since A is the most specific common superclass of B and C. Furthermore, the similarity between class B and of every subclass of D should be again S_A for the same reason. But intuitively, the similarity between B and C should be greater than the similarity between B and, for example, F. The problem is that the algorithm does not take into account the level of the examined classes in relation to their most specific common superclass.

In our approach, we do not assign a weight to each inner class but we implement the notion of the distance d between two classes C_i and C_k which we define as the number of classes that exist in the (shortest) path from C_i to C_k (including also in the sum the C_i and C_k), e.g. the distance between two classes with a direct subclass relationship is 2. However, the distance should be carefully defined, taking into account the semantics of the hierarchy, since subclass relationships do not necessarily mean subsumption relationships in our OO model.

In section 3 we describe the basic principles of the transformation procedure of an OWL ontology into the OO model. Subclass relationships of ontology classes are mapped into subclass relationships in the OO class hierarchy, implementing in that way property and class inheritance. We follow the same procedure in the case of class equivalence, where we define special subclass relationships among the corresponding classes, as Fig. 1 depicts. However, by considering class equivalence as actual subclass relationships, a problem emerges concerning the accuracy of a distance function. A distance algorithm that would be based directly on the class hierarchy would lead in inaccurate results, since the distance between class A and B or C in Fig. 1 (a) would be equal to 2, denoting subclass relationships. However, the actual distance is 1, due to class equivalence.

We define Algorithm 2 which computes the minimal distance between two classes not only based on the class hierarchy but also taking into account class equivalence. The algorithm actually implements the *Branch and Bound*

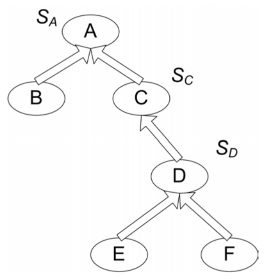


Fig. 3. A simple class hierarchy with inner class weights.

search strategy: starting from the most specific class of the two input classes (*subC*), expands the frontier set in each recursion by retrieving the direct superclasses of each class. The frontier set contains pairs of the form <*class*, *distance*> where *class* is the current class and *distance* is the calculated distance from the beginning. These pairs are created by the *parentList* function which decreases the *distance* value by one when a class equivalence relationship is found between a class and its superclass. The class equivalence between C_i and C_k ($C_i \equiv C_k$) can be determined by checking the values of the `owl:equivalentClass` property of the corresponding class *meta-objects* $[C_i]$ and $[C_k]$:

$$C_i \equiv C_k \rightarrow [C_i] \in [C_k].owl{:}equivalentClass \vee [C_k] \in [C_i].owl{:}equivalentClass$$

If the two classes C_i and C_k do not have hierarchical relationship, then the algorithm returns -1. It is worth mentioning that the determination of subclass relationships, such as $C_i \sqsubseteq C_k$, or the retrieval of the superclasses of a specific class, i.e. the *directSuperClasses* function, are common and optimized procedures in any OO programming environment, including CLIPS and thus, the *dist* function can be easily implemented.

Additionally, we define the *classSimilarity* function (Algorithm 3) for determining the degree of similarity between two classes based on their position in the hierarchy. If two classes have hierarchical relationship, then their similarity equals to their *distance*, as we have defined it in section 4.2. Otherwise, the function returns the sum of the distances of each class from their most specific common superclass minus 1. If there is not such a common superclass, the function returns 0. Moreover, the function returns 0 if the two examined classes are disjoint. This can be determined by examining the `owl:disjointWith` property of class meta-objects:

$$C_i \sqcap C_k \sqsubseteq \perp \rightarrow [C_i] \in [C_k].owl{:}disjointWith \vee [C_k] \in [C_i].owl{:}disjointWith$$

Algorithm 2: Distance between two classes.
Inputs: Two classes C_i and C_k.
Output: Distance value.

```
function dist($C_i$, $C_k$) {
  var closedSet := ∅, frontier := ∅
  var curD, cur, subC, supC, minD, dist
  if $C_i \sqsubseteq C_k$ then
    subC := $C_i$
    supC := $C_k$
  else if $C_k \sqsubseteq C_i$ then
    subC := $C_k$
    supC := $C_i$
  else return -1 //not hierarchically related classes
  closedSet := ∅
  frontier := {<subC, 1>}
  minD := ∞
  while frontier ≠ ∅ do
    <cur, dist> := first(frontier)
    frontier := rest(frontier)
    curD := dist + 1
    if cur ∉ closedSet then
      if (cur = supC ∨ cur ≡ supC) ∧ curD < minD then
        minD := curD
      else if curD < minD then
        frontier:= parentList(cur, curD) ∪ frontier
        closedSet := closedSet ∪ {cur}
  return minD
}

function parentList(C, currentDist) {
  var list := ∅, superClasses := ∅
  superClasses := directSuperClasses(C)
  for each x ∈ superClasses do
    if x ≡ C then list := list ∪ {<x, currentDist - 1>}
    else list := list ∪ {<x, currentDist>}
  return list
}
```

Algorithm 3: Similarity between two classes.
Inputs: Two classes C_i and C_j.
Output: Similarity score.

```
function classSimilarity($C_i$, $C_k$) {
  var distance, comSuperClass, $d_i$, $d_k$
  if $C_i \sqcap C_k \sqsubseteq \perp$ then //disjoint classes
    return ∞
  distance := dist($C_i$, $C_k$)
  if distance = -1 then
    comSuperClass := mostSpecificSuperClass($C_i$, $C_k$)
    if comSuperClass = ∅ then
      return ∞
    else
      $d_i$ := dist($C_i$, comSuperClass)
      $d_k$ := dist($C_j$, comSuperClass)
      return $d_i + d_k - 1$
  else return distance
}

function mostSpecificSuperClass($C_i$, $C_j$) {
  var comClasses := ∅, cur
  comClasses := superClasses($C_i$) ∩ superClasses($C_j$)
  comClasses := comClasses - {owl:Thing}
  for each c ∈ comClasses do
    if ∄c′ ∈ comClasses | c′ ⊑ c then return c
  return ∅
}
```

Having defined the distance and the similarity between two classes, we present Algorithm 4 which computes the overall similarity between a query profile ($I_{Q,r}$) and an advertisement profile relational inputs ($I_{W,r}$). The algorithm traverses every advertisement relational input and checks if there is one in the query that matches. The threshold a represents the maximum distance we want to exist between two sibling classes during the matching procedure. The returned value describes the sum of the weights for each type match divided by the number of the advertisement inputs. There are four matches: (a) *exact* with a standard weight of 4, (b) *plug-in* with weight 2+2/d, (c) *subsume* with weight 1+2/d and (d) *sibling* with weight 2/d. Notice that these are example weights that can be configured appropriately. The same algorithm can be used for matching relational outputs, i.e. $f_r(O_{W,r}, O_{Q,r}, a)$.

Algorithm 4: The f_r function for relational inputs.
Inputs: The $I_{Q,r}$ and $I_{W,r}$ sets and the threshold a.
Output: The similarity score.

```
function $f_r(I_{Q,r}, I_{W,r}, a)$ {
  var $sim := 0, N :=$ size($I_{W,r}$)
  if $N = 0$ then return 1
  for each $c \in I_{W,r}$ do
    if $\exists c' \in I_{Q,r}$ | classSimilarity($c, c'$) $\neq \infty$ then
      if classSimilarity($c, c'$) = 1 then //exact match
        $sim := sim + 4$
      else if $c' \sqsubseteq c$ then //plug-in match
          $sim := sim + 2 + (2/$classSimilarity$(c, c'))$
        else if $c \sqsubseteq c'$ then //subsume match
            $sim := sim + 1 + (2/$classSimilarity$(c, c'))$
          else if classSimilarity($c, c'$) $\leq$ a then
              $sim := sim + 2/$classSimilarity$(c, c')$
            else return 0
    else return 0
  return $sim/N$
}
```

4.3 Overall Profile Similarity

Based on the f_s and f_r functions, we define the similarity S of two profile objects p_Q and p_A. The returned value represents the matching score of the two profile objects and it can be used in order to rank the results of a request against multiple advertisements. We define also the *matchmaking* function (Algorithm 6) which examines all the profile objects of the KB against a query and returns the results in descending order according to the score of each match. A threshold N can be used in order to define the maximum number of the results.

Algorithm 5: Similarity S of two profile objects.
Inputs: Two profile objects p_Q and p_A and the sibling distance threshold a.
Output: The similarity score.

```
function $S(p_Q, p_A, a)$ {
  var $I_{Q,s}, O_{Q,s}, I_{A,s}, O_{A,s}, I_{Q,r}, O_{Q,r}, I_{A,r}, O_{A,r}, S_s, S_r$
  for each $i \in$ inputs($p_Q$) do
    if datatype($i$) then $I_{Q,s} := I_{Q,s} \cup \{i\}$
    else $I_{Q,r} := I_{Q,r} \cup \{i\}$
  for each $o \in$ outputs($p_Q$) do
    if datatype($i$) then $O_{Q,s} := O_{Q,s} \cup \{o\}$
    else $O_{Q,r} := O_{Q,r} \cup \{o\}$
  for each $i \in$ inputs($p_A$) do
    if datatype($i$) then $I_{A,s} := I_{A,s} \cup \{i\}$
    else $I_{A,r} := I_{A,r} \cup \{i\}$
  for each $o \in$ outputs($p_A$) do
    if datatype($i$) then $O_{A,s} := O_{A,s} \cup \{o\}$
    else $O_{A,r} := O_{A,r} \cup \{o\}$
  $S_s := \mathbf{f_s}(I_{Q,s}, I_{A,s}) * \mathbf{f_s}(O_{A,s}, O_{Q,s})$
  $S_r := \mathbf{f_r}(I_{Q,r}, I_{A,r}, a) * \mathbf{f_r}(O_{A,r}, O_{Q,r}, a)$
  return $S_s * S_r$
}
```

Algorithm 6: Matchmaking results.
Inputs: A query profile object p_Q, the sibling class distance threshold a and the maximum number of results N
Output: Similar advertisement profile objects to a query in descending order.

```
function $matchmaking(p_Q, a, N)$ {
  var $match = \emptyset, rank, result$
  for each $p : Profile$ do
    $rank :=$ S($p_Q, p, a$)
    if $rank \neq 0$ then $match := match \cup \{<p, rank>\}$
  $\mathbf{sort}_{\mathbf{desc,rank}}(match)$
  while $N > 0$ do
    $result := result \cup$ first($match$)
    $match :=$ rest($match$)
    $N := N - 1$
  return $result$
}
```

5. A Simple Use Case

In this section we give a simple example presenting the rationale of our methodology. Assume the simple ontology of Fig. 4 which is already processed by the underlying OWL rule reasoner, generating the corresponding class hierarchy into the OO KB.

We also assume that there are 4 Web services and a query with the following characteristics:

- **WS1(in:{*Title*}, out:{*Publisher*})**: takes as input a book title and returns its publisher.
- **WS2(in:{*Title*}, out:{*Person*})**: takes as input a book title and returns instances of any person of the ontology.
- **WS3(in:{*Title*}, out:{*Author*})**: takes as input a book title and returns its author.
- **WS4(in:{*Title*}, out:{*Author*})**: takes as input an article title and returns its author.
- **query(in:{*Title*}, out:{*Author*})**: find the author based on the book title.

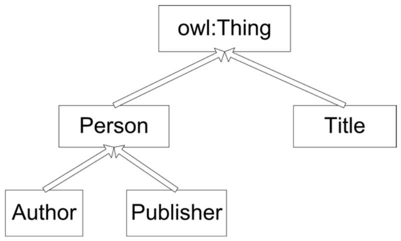


Fig. 4. A sample ontology

For each Web service there is a profile instance that corresponds to the semantic description of the I/O parameters. Moreover, there is a profile object representing the query for a specific Web service. These instances constitute the profile objects of our OO KB and are presented in the simplified UML object diagram of Fig. 5.

In this simple example, the input parameters of all Web services and the input parameter of the query match exactly, since they refer to the same *Title* concept of the ontology. Thus, according to Algorithm 4 we have: $f_r(I_{query,r}, I_{WS1,r}, a) = f_r(I_{query,r}, I_{WS2,r}, a) = f_r(I_{query,r}, I_{WS3,r}, a) = f_r(I_{query,r}, I_{WS4,r}, a) = 4$, for every value of a since there are only exact matches. For the output parameters there are three cases:

- There is an exact match between the output concepts of the query and WS3 and WS4, thus $f_r(O_{W3,r}, O_{query,r}, a) = f_r(O_{W4,r}, O_{query,r}, a) = 4$ and $S(p_{query}, p_{WS3}, a) = S(p_{query}, p_{WS4}, a) = 4*4 = 16$.
- There is a subsume match between the output *Author* concept of the query and the output concept *Person* of the WS2, thus $f_r(O_{W2,r}, O_{query,r}, a) = 2$, since d = **classSimilarity**(*Author*, *Person*) = 2 and the weight for the subsume match is 1+2/d = 2. Thus, $S(p_{query}, p_{WS2}, a) = 4*2 = 8$.
- There is a sibling relation between the output concept *Author* of the query and the output *Publisher* concept of WS1, thus $f_s(O_{W1,r}, O_{query,r}, a) = 2/3$ since d = **classSimilarity**(*Author*, *Publisher*) = 3 and the weight for the sibling match is 2/d = 2/3. Thus, $S(p_{query}, p_{WS1}, a) = 4*2/3 = 2.66$. The algorithm needs an appropriate a value in order to take into account the sibling match. For our example, such a value should be greater or equal to 3 in order to capture the sibling relation of the example where the distance between the sibling classes is 3. Intuitively, this is the minimum value of the threshold a, i.e. the distance between the direct subclasses of a common superclass. An a value less than 3 means that we do not want sibling matches during matchmaking.

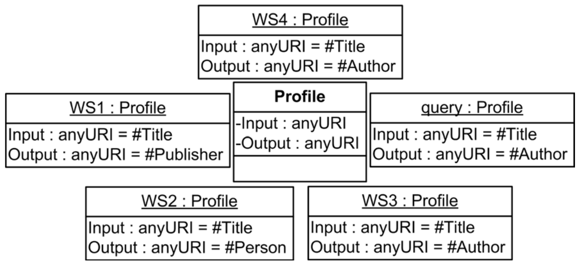


Fig. 5. UML object diagram for the profile objects.

In that way, the matchmaking procedure will return the four services in a descending order: WS4=WS3> WS2>WS1, assuming that $N \geq 4$. In this example, the sibling class relationship is of minor importance since there are exact and subsume matches. However, the sibling class relationships could be proved very useful especially in cases where the matchmaking process does not return any result (due to the absence of exact, plug in or subsume matches) and there is a need to relax the matchmaking criterion. Furthermore, although WS4 is semantically irrelevant to the query, it is returned since its I/Os are satisfied. Such situations can be circumvented by using a profile taxonomy, as we describe in the next section.

6. Taxonomy-based Profile Matchmaking

So far we have described how object similarity measures can be applied in the domain of Web service matchmaking of profile objects that belong directly to the Profile class. However, Web service profiles can be classified into taxonomies of profiles according to their functionality [17]. In this case, each profile object belongs to a class defined as subclass of the Profile class, creating in that way a profile taxonomy.

Intuitively, hierarchically related profiles denote services with potentially the "same" functionality. To this end, queries could be also classified in the taxonomy according to the requested service. In that way, we could use our inter-class similarity measure in order to *filter* profiles during the matchmaking procedure by pruning WS profiles with no hierarchical relationship with the query object class or by relaxing the matchmaking criterion in order to take into consideration sibling classes.

The filtering of profile objects during the matchmaking procedure can be realized with Algorithm 7, where *class*(o) is the class of the object o.

```
Algorithm 7: Relevant profile object to a query
Inputs: The query object q, a profile object p and the distance threshold b for sibling profile class relations.
Output: Profile is relevant (TRUE) or not (FALSE)

function relevantProfile(q, p, b) {
  var temp = classSimilarity(class(q), class(p))
  if temp = ∞ then return FALSE
  if temp = 1 then return TRUE
  if class(q) ⊑ class(p) ∨ class(p) ⊑ class(q) then
    return TRUE
  if temp ≤ b then return TRUE
  return FALSE
}
```

In that way, Algorithm 6 can be extended in order to perform matchmaking based on "relevant" profile objects. We use a threshold *b* that defines the sibling profile class distance threshold in order to be able to declare different relaxing policies between concepts and profiles.

```
Algorithm 6b: Extended matchmaking algorithm.
Inputs: A query profile object p_Q, the sibling concept distance threshold a, the sibling profile class distance threshold b and the maximum number of results N.
Output: Similar advertisement profile objects to a query in descending order.

function matchmaking(p_Q, a, b, N) {

  for each p : Profile | relevantProfile (p_Q, p, b)=TRUE
    do …
}
```

To give an example, consider the profile hierarchy of Fig. 6. Assume that the four Web service profile objects of section 0 have been classified accordingly to their functionality. Let also be a variety of Web service profile objects (WSN) been classified into different classes.

A query object defined to belong directly in the Profile class would lead to the examination of all the profile objects of the hierarchy, since $\forall p$: *Profile*, $class(p)=class(q) \vee class(p) \sqsubseteq class(q)$ (the similar case to section 5) and thus Algorithm 7 would always return TRUE. However, by defining the query to be an object of an appropriate class, according to the requested service, we give the opportunity to filter out Web services. Assuming that $b<3$:

- If q : *InformationServices*, then the matchmaking procedure will be performed $\forall p \in C \mid C \sqsubseteq$ *InformationServices* $\vee$ *InformationServices* $\sqsubseteq C$. For example, profiles that are direct or indirect objects of the *E-commerce* class will be pruned.

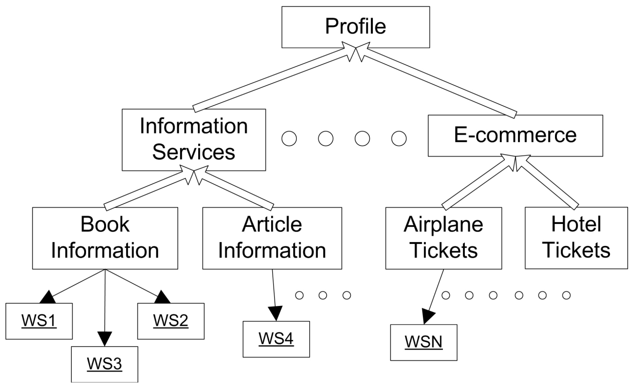


Fig. 6. Example profile hierarchy.

- If q : *BookInformation,* then additionally to the previous case, the profile objects of the *ArticleInformation* class will be pruned. In that case, WS4 will not be returned since it is an irrelevant Web service to the book domain.

By setting a *b* value greater or equal to 3, we let profile objects of sibling classes to participate also in the matchmaking procedure in an analogous manner as we have described in section 0.

7. Related Work

Many research efforts have been focused on the field of Web service discovery. In this section we briefly present some of these approaches.

In [19] the authors propose a prototype for semi-automating Web service composition. Users create a workflow of services by presenting the available choices at each step. Web services descriptions are defined in DAML-S and through an OWL Prolog reasoner, the system inferences and selects matching services based on subsumption relationships. Services are also filtered based on constraints which the user may specify.

The authors in [20] describe the implementation of the DAML-S/UDDI Matchmaker that expands on UDDI by providing semantic capability matching. They provide an extensive description on the theoretical framework underlying the use of DAML-S for Web service discovery, interaction and composition. Advertisements and requests refer to DAML concepts and the matching process can perform inferences on the subsumption hierarchy leading to the recognition of semantic matches.

In [9] the authors describe a service matchmaking prototype which uses a DAML-S based ontology and the Racer [4] reasoner. The reasoner checks the satisfiability of the request with each advertisement by computing the subsumption relationships between advertisements and requests.

In [6] the authors present a logical framework for automated Web service discovery which is based on the

WSMO conceptual model. They have implemented their approach in the F-Logic reasoning engine Flora2.

The above approaches are based on subsumption relationships that are derived from ontology descriptions of Web services. In [7], an approach to hybrid semantic Web service matching is presented, called OWLS-MX, that utilizes both logic-based reasoning and content-based information retrieval techniques for services specified in OWL-S. The authors show that hybrid approaches to semantic matching that exploit both formal and implicit semantics may improve the retrieval performance of semantic service matching over purely logic-based ones. The implementation is based on the combination of syntactic similarity metrics, such as the extended Jacquard similarity coefficient or the cosine similarity value, and on the semantic derivations of Pellet DL reasoner.

WSMO-MX [5] is also an approach to hybrid semantic web service matching based on both logic programming and syntactic similarity measurement in order to retrieve WSMO-oriented service descriptions that are semantically relevant to a given query. They use the notion of the derivative and perform matching of a goal derivative with a service description derivative based on type matching, logical constraint, relation and syntactic matching.

In our approach we introduce a new methodology of matching ontology concepts: we exploit the OO schema that emerge after the transformation of OWL ontologies into a native OO model by customizing appropriately object similarity measures inspired from the domain of OO programming. The resulting methodology is able to match concepts based not only on the subsumption relationships that are handled by the inheritance mechanism of the underlying OO rule reasoner (CLIPS) but also on class relationships stemming from the OO schema, such as relationships among sibling classes. Such relations can be proved very useful and should not be totally ignored during the discovery of Web services [5][7].

8. Conclusions and Future Work

The efficient discovery of semantic Web services demands sophisticated frameworks and algorithms able to handle the semantic nature of their description. In this paper we approach the problem of semantic Web services matchmaking from an OO perspective.

In O-DEVICE [12], we have shown that the OO characteristics of OWL, such as the classes, properties, instances, class and property inheritance, etc., as well as OWL semantics can be easily mapped into rules and constructs of the OO model. Motivated by the results of this work, we have extended our OO methodology to the domain of the OWL-S profile descriptions of Web services. Since the profiles and requests in our framework are represented as objects, we apply object similarity measures in order to realize the matchmaking procedure. The utilization of such similarity measures, gives us the opportunity to perform Web service discovery based on hierarchical relationships that cannot be captured by the logic-based subsumption relation. The proposed methodology can be implemented also in a non OO environment by the utilization of an OWL ontology reasoner for the determination of hierarchical relationships. However, the generation of an OO model from OWL ontologies following the methodology in [12] gives great advantage since hierarchical relationships can be determined at once by the OO environment, without performing runtime inferencing.

In section 4 we explain the reasons for which we are not able to apply directly object similarity measures on profile objects and how they can be customized for the domain of OWL-S Web services matchmaking. However, with a more sophisticated web Services profile management policy, the initial object similarity measures can be proved very useful.

By assuming that each profile object belongs directly to the Profile class, we use the inter-class similarity measure (as we have defined it) in order to determine the similarity of the classes that are referred in the I/O profile parameters and not the similarity of the profile objects themselves. However, Web service profiles can be categorized into profile taxonomies according to their functionality, as the OWL-S example of the Congo Web services refers [16]. In that case, we can use the inter-class similarity in order to filter a priori profile objects that are not “similar” to a particular query, according to a minimum distance, since the profile objects would not necessarily belong directly to the same (Profile) class.

Such filtering can be performed by examining also non-functional properties of the profile objects which they inherit due to their classification into a class hierarchy, such as price or location. In that case, the profile objects would contain actual values in these specific non-functional properties, enabling the direct application of the intra-class similarity measure, as it is defined in [1], determining similarities based on actual values and not based on datatypes or classes.

For the future, we plan to enrich our methodology with services pre and post conditions. An interesting work relevant to this field is presented in [23] where the authors describe a variety of relaxed matches of software components based on what a component “requires” and what “ensures”, in a similar manner to Web services pre and post conditions. We plan also to use our methodology in [11] where we perform discovery based only on subsumption relationships that are derived by the underlying OO rule reasoner. Finally, we investigate the combination of our methodology with machine learning techniques, such as text mining or clustering in order to enrich even more the Web services retrieval process with non-logic based approaches.

Acknowledgments. This work was partially supported by a PENED program (EPAN M.8.3.1, No. 03EΔ73), jointly funded by the European Union and the Greek Government (General Secretariat of Research and Technology/GSRT) and by a NON-EUROPE project (GSRT - 05 NON EU 423).

References

[1] Bergmann, R., Stahl, A. "Similarity Measures for Object-Oriented Case Representations", In Proc. of the 4th European Workshop on Case-Based Reasoning, 1998.

[2] CLIPS, http://www.ghg.net/clips/CLIPS.html

[3] Gomes, P., Pereira, F.C., Paiva, P., Seco, N., Carreiro, P., Ferreira, J.L., Bento, C. "Experiments on Case-Based Retrieval of Software Designs", In Proc. of the European Conference on Case-Based Reasoning (ECCBR'02)

[4] Haarslev, V., Möller, R. "Racer: A Core Inference Engine for the Semantic Web", 2nd Int. Workshop on Evaluation of Ontology-based Tools (EON2003), Sanibel Island, Florida, USA, pp. 27–36, 2003

[5] Kaufer, F., Klusch, M. "WSMO-MX: A Logic Programming Based Hybrid Service Matchmaker", In Proc. of the 4th IEEE ECOWS, IEEE, Zurich, Switzerland, 2006

[6] Kifer, M., Lara, R., Polleres, A., Zhao, C., Keller, U., Lausen, H., Fensel, D. "A Logical Framework for Web Service Discovery", In SWS Workshop at ISWC, Hiroshima, Japan, November 2004

[7] Klusch, M., Fries, B., Sycara, K. "Automated Semantic Web Service Discovery with OWLS-MX", In Proc. of 5th Int. Conference on Autonomous Agents and Multi-Agent Systems (AAMAS), Hakodate, Japan, 2006

[8] Kontogiannis, K., Mori, R.D., Bernstein, R., Galler, M., Merlo, E. "Pattern matching for clone and concept detection", Journal of Automated Software Engineering, 1996

[9] Li, L., Horrock, I. "A software framework for matchmaking based on semantic web technology", In Proc. 12th Int World Wide Web Conference Workshop on E-Services and the Semantic Web (ESSW 2003)

[10] Maiden, N., Sutcliffe, A. "Case-based reasoning in software engineering", Case-Based Reasoning, IEE Colloquium on, Vol., Iss., 12 Feb 1993

[11] Meditskos G., Bassiliades N. "A Semantic Web Service Discovery and Composition Prototype Framework Using Production Rules", OWL-S: Experiences and Future Developments workshop in conjunction with the 4th ESWC, Innsbruck, Austria, 6 June, 2007

[12] Meditskos, G., Bassiliades, N. "Towards an Object-Oriented Reasoning System for OWL", Proc. Workshop OWL: Experiences and Directions, Galway, Ireland, CEUR, 2005.

[13] Merlo, E., Antoniol, G., Di Penta, M., Rollo, V.F. "Linear Complexity Object-Oriented Similarity for Clone Detection and Software Evolution Analyses," icsm, pp. 412-416, 20th IEEE Int. Conference on Software Maintenance

[14] OWL Web Ontology Language Overview,http://www.w3.org/TR/owl-features/

[15] OWL-S: Semantic Markup for Web Services, http://www.w3.org/Submission/OWL-S/

[16] OWL-S 1.1 Release: Examples, http://www.daml.org/services/owl-s/1.1/examples.html

[17] Profile-based class hierarchies, http://www.daml.org /services/owl-s/1.1/ProfileHierarchy.html

[18] Sager, T., Bernstein, A., Pinzger, M., Kiefer, C. "Detecting similar Java classes using tree algorithms", In Proc. of the 2006 international workshop on Mining software repositories, May 22-23, 2006, Shanghai, China

[19] Sirin, E., Hendler, J., Parsia, B. "Semi-automatic composition of Web services using semantic descriptions", In Proc. of Web Services: Modeling, Architecture and Infrastructure workshop in conjunction with ICEIS2003

[20] Sycara, K., Paolucci, M., Anolekar, A., Srinivasan, N. "Automated discovery, interaction and composition of semantic web services", Web Semantics, 1(1), Elsevier, 2003

[21] WSDL-S, http://www.w3.org/Submission/WSDL-S/

[22] WSMO: http://www.wsmo.org/TR/d2/v1.3/.

[23] Zaremski A.M. and Wing J.M. "Specification matching of software components", ACM Transactions on Software Engineering and Methodology, 6(4), pp 333-369, 1997.