View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by El Servicio de Difusión de la Creación Intelectual



# A Semi-Automatic Method for Ontology Mapping *

Laura Perez, Agustina Buccella, and Alejandra Cechich

GIISCO Research Group
Departamento de Ciencias de la Computación
Universidad Nacional del Comahue
Neuquen, Argentina

**Abstract.** Ontology mapping involves the task of finding similarities among overlapping sources by using ontologies. In a Federated System in which distributed, autonomous and heterogeneous information sources must be integrated, ontologies have emerged as tools to solve semantic heterogeneity problems. In this paper we propose a three-level approach that provides a semi-automatic method to ontology mapping. It performs some tasks automatically and guides the user in performing other tasks for which his/her intervention is required. Finally, a plug-in of the ontology editor, Protégé, is presented showing how the method is implemented through a case study.

## 1 Introduction

A Federated System [7] refers to the data integration of distributed, autonomous and heterogeneous information sources. In general it is implemented by using a 4-layer architecture [3], in which the *federation layer* is the core of the system. In this layer, each information source is described by its own ontology (domain ontologies), and all of them converge in one global shared vocabulary. This shared vocabulary contains basic terms (the primitives) of the whole domain. In this way, an hybrid ontology approach [18] is followed.

Several proposals on ontology mapping have emerged in the last years [9]. Among them, we can cite [13, 12, 11, 14]. For example, [13] and [12] propose two similar ontology-merging tools. On one hand, the PROMPT tool described in [13] proposes an interactive tool that guides the user through the merging process. However the main problem with the PROMPT tool is that it is highly dependent on the names of the concepts in the ontology. On the other hand, Chimarea [12] provides support for merging of ontological terms from different sources, checking the coverage and correctness of ontologies and maintaining ontologies over time. Except for several situations referring to structural aspects of the ontologies, Chimarea does not make any suggestion to the user; and the only relation that Chimarea considers is the subclass/superclass relation. Another proposal for semantic matching is introduced in [11], where a lexical and a conceptual layer are used to find similarities. At the lexical level, the method uses a lexical function called lexical similarity measure (SM). At the conceptual level, concepts (classes and properties) are compared taking into account the taxonomies in which they appear. However, some types of properties are not considered by this method. Finally, the proposal of Rodriguez et.al [14] presents a combination of two different approaches to similarity assessment – the feature matching process [17] and the semantic distance. Common features increase the similarity value and distinct features decrease it. The main disadvantage with this method is that the similarity values cannot be calculated neither automatically nor semi-automatically due to the high dependence on natural language descriptions.

In previous work [5, 4], we have proposed a three-level approach that allows us to build similarities expressed as mappings. In this work we improve this method taking into account

* This work is partially supported by the UNComa project 04/E059 (Mejora del Proceso de Desarrollo de Software Basado en Componentes).

cycles in the ontologies and implementing a tool as a plug-in of the ontology editor Protégé [1]. This extension allows us to introduce a semi-automatic process for ontology mapping.

This paper is organized as follows: Section 2 shows the main steps of our three level approach describing changes performed to detect and solve cyclic ontologies. Then, in Section 3 the structure of components for our plug-in is described. An example describing how our tool works is shown in Section 4. We discuss future work and conclusions afterwards.

## 2 A Three-Level Approach to Ontology Mapping

In previous work, we have proposed a three-level method to ontology merging, taken into account information ontologies provide [5]. In this way, concepts of an ontology are compared using three comparison levels: *syntactic*, *semantic* and *user* level. Figure 1 shows our approach graphically, where the levels are part of the process.

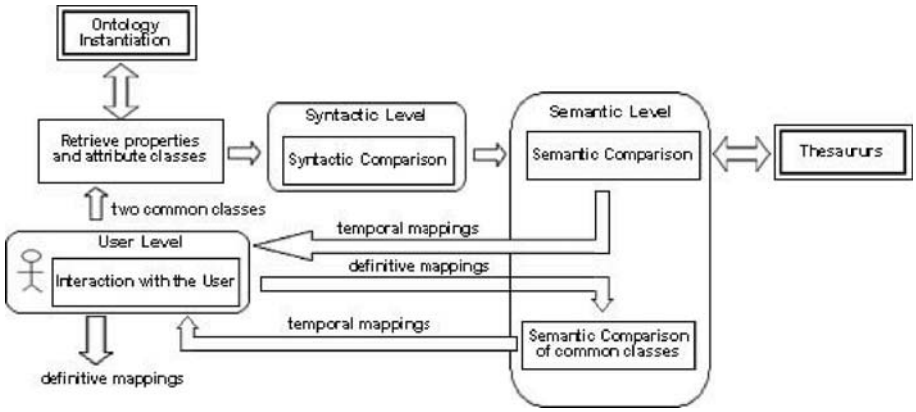


**Fig. 1.** Approach for searching similarities

*Ontology Instantiation* and *Thesaurus* are the two external modules. The former, obtains the object structure from ontologies described in some ontology language. Figure 2 shows how the different elements of an ontology are divided. The first division refers to two different elements. On one branch we have the *classes* and on the other branch the *properties*. Firstly we analyze the *classes* branch, which is also divided into two new branches: *common classes* and *attribute classes*. Both are classes defined in the ontology to represent things about the world. The specific role defined in the ontology is the difference between them. The *common classes* have the role of representing things about the domain and the *attribute classes* have the role of representing information about a common class. Both roles exist because some concepts of the ontologies act as attributes. For example, an ontology can have the *Animal* class as a common class and the *Organ* class as an attribute class because *Organ* exists to describe a characteristic about a common class. The *Organ* class has no properties.

On the other branch, Figure 2 shows the *properties* branch which is also divided into two new branches: *datatype properties* and *special properties*. A property is a set of tuples that represents a relationship among objects in the universe of discourse. Each tuple is a finite, ordered sequence (i.e., list) of objects. The properties have restrictions to denote functions, cardinality, domain, range, etc. The *datatype properties* are properties relating a class or a set of classes with a data type. For example, the animal name is a common property between the

*Animal* class and the *String* data type. On the other hand, the *special properties* are properties relating classes. For example, the relationship between the *Animal* class and the *Organ* class to denote the organs of an animal. Thus, a common class has both datatype properties and special properties, and attribute classes do not have properties.

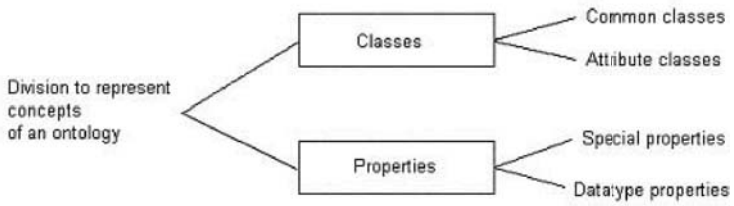


**Fig. 2.** Proposed division to represent the ontology

The latter, the *Thesaurus* module, uses thesauruses to search for synonyms, which are detected by the module through the use of a similarity function.

In Figure 1, two common classes (of different ontologies) are indicated by the user. These two classes are inputs of the *Retrieve Properties and Attribute Classes* module, which retrieves the attribute classes and special and datatype properties of each class by using the object structure of each ontology. This retrieved information enters the *Syntactic Comparison* module, which analyses syntactically classes and properties relating with the concepts. A set of syntactic functions [5] are used here.

Then, the *Semantic Comparison* module compares the classes and properties semantically. To do so, we extract semantic information from the Thesaurus module in order to find synonym relationships. Using the results of the syntactic level functions, we construct functions that combine these values together with the thesaurus information.

In the *Interaction with the User* module all the mappings that exceed a threshold are shown to the user, and he decides if the mappings are correct. The accepted mappings are classified as definitive mappings.

The *Semantic Comparison for Common Classes* module receives the definitive mappings and compares the common classes of the two ontologies. It uses the mappings added by the comparison of properties in order to denote the set of similar attributes (properties) of both classes.

One more time, in the *Interaction with the User* module, all mappings are displayed to the user and he decides if these mappings must be added permanently.

## 2.1 Improving the Method: Detecting Cycles

Our approach analyzes the ontologies as graphs, taking into consideration both taxonomic and non taxonomic relationships among terms.

Figure 3 describes our basic method for searching similarities. The method has a series of steps depending on the different elements of the ontology (see Figure 2).

Firstly, a user must indicate the first mapping, for example between the *Animal* class of one ontology and the *Creature* class of the other ontology. If the classes are common classes, the system compares firstly the datatype properties of both classes. The *cleaning_process* in the method

**Similarity(O1,O2)**

the user enter to similar classes (c1,c2)
if (c1 and c2 are common classes)
for each datatype property $dtp_i \in c1$ and $dtp_j \in c2$
$cleanning\_process(dtp_i, dtp_j)$
$sim1_{thesaurus}(dtp_i, dtp_j) = search\_on\_thesaurus(dtp_i, dtp_j)$
$sim1_{sint}(dtp_i, dtp_j) = w_{ed} * sim_{ed}(dtp_i, dtp_j) + w_{tri} *$
$sim_{tri}(dtp_i, dtp_j) + w_{dtc} * sim_{dtc}(range\_of(dtp_i), range\_of(dtp_j)) +$
$w_{thesaurus} * sim1_{thesaurus}(dtp_i, dtp_j)$
if $sim1_{sint}(dtp_i, dtp_j) \geq th_{accept}$
$add\_mapping(dtp_i, dtp_j)$
for each special property sp_i $\in c1 and sp_j \in c2$
$cleanning\_process(sp_i, sp_j)$
$sim2_{thesaurus}(sp_i, sp_j) = search\_on\_thesaurus(sp_i, sp_j)$
$sim_{rest}(sp_i, sp_j) = check\_restrictions(sp_i, sp_j)$
$sim2_{sint}(sp_i, sp_j) = w_{ed} * sim_{ed}(sp_i, sp_j) + w_{tri} * sim_{tri}(sp_i, sp_j) +$
$w_{thesaurus} * sim2_{thesaurus}(sp_i, sp_j) + w_{rest} * sim_{rest}(sp_i, sp_j)$
$if(c1, c2) \notin Visited$
*add_visited(c1,c2)*
if $(range\_of(sp_i), range\_of(sp_j)) \notin Mapped$
$sim_{total}(sp_i, sp_j) =$
$calculate\_all\_the\_process\_for(range\_of(sp_i), range\_of(sp_j))$
if $(range\_of(sp_i), range\_of(sp_j)) \in Mapped$
$sim_{total}(sp_i, sp_j) = get\_value(range\_of(sp_i), range\_of(sp_j))$
$sim_{sp}(sp_i, sp_j) = w_{sint} * sim2_{sint}(sp_i, sp_j) + w_{total} * sim_{total}(sp_i, sp_j)$
if $sim_{sp}(sp_i, sp_j) \geq th_{accept}$
$add\_mapping(sp_i, sp_j)$
*remove_visited(c1,c2)*
using the added mappings
$cleanning\_process(c1, c2)$
$sim3_{thesaurus}(c1, c2) = search\_on\_thesaurus(c1, c2)$
$sim3_{sint}(c1, c2) = w_{ed} * sim_{ed}(c1, c2) + w_{tri} * sim_{tri}(c1, c2) +$
$w_{thesaurus} * sim3_{thesaurus}(c1, c2)$
if c1 and c2 are attribute classes
$sim_{class}(c1, c2) = sim3_{sint}(c1, c2)$
if c1 and c2 are common classes
$sim_{class}(c1, c2) = w_{sint} * sim3_{sint}(c1, c2) + w_{propiedadtes\_especiales}$
$* sim_{propiedadtes\_especiales} + w_{propiedadtes\_tipo\_de\_dato} * sim_{propiedadtes\_tipo\_de\_dato}$
$if(c1, c2) \notin Visited$
if $sim_{class}(c1, c2) \geq th_{accept}$
$add\_mapping(c1, c2)$
$if(c1, c2) \in Visited$
if $sim_{class}(c1, c2) \geq (th_{accept} - (th_{accept} * w_{propiedadtes\_especiales}))$
$add\_mapping(c1, c2)$

**Fig. 3.** Steps for searching similarities

denotes the process of elimination of articles, prepositions and non-relevant characters (_,:,-, etc.). Next, thesauruses are used to search for synonymies. The function $sim1_{thesaurus}(dtp_i, dtp_j)$ is equal to 1 if a synonym relationship is found for the two datatype properties and it is equal to 0 otherwise.

Then, the $sim1_{sint}(dtp_i, dtp_j)$ function is calculated using four syntactic functions. The *edit distance* function, which considers the number of changes that must be done to turn one string into the other, and weights the number of these changes with respect to the length of the shortest string. The *trigram* function [10], which is based on the number of different trigrams in two concepts or strings. The *data type compatibility* function, $(sim_{dtc}(range\_of(dtp_i), range\_of(dtp_j)))$ which compares the datatype of the ranges. And the result of applying the *thesaurus* function, aforementioned. The $sim1_{sint}(dtp_i, dtp_j)$ function returns a value between 0 and 1; and the sum of weights, the $w$ values ($w_{ed}, w_{tri}, w_{dt}$ and $w_{thesaurus}$), is equal to 1.

Finally, if the result of the function exceeds a threshold ($th_{accept}$), a temporal mapping is added.

The $sim_{rest}(sp_i, sp_j)$ function checks special property restrictions [16] such as functional, symmetric, allValuesFrom, someValuesFrom, cardinality, etc. That is, it compares the constrains applied to the properties. Only when both properties have the same restrictions, the function returns 1; otherwise it returns a percentage according to the number of restrictions that are the same.

Then, a temporal mapping is added when the $sim_{sp}(sp_i, sp_j)$ function exceeds the threshold.

Following, the method compares the special properties included in the common classes. The comparison is similar to the previous case, but the *datatype compatibility* function is not calculated.

The $sim_{total}(sp_i, sp_j)$ function makes all the similarity process taking into account the range of the special properties. Therefore, this is a recursive method that will stop when the ranges are attribute classes (because they do not have properties). Once again, thesauruses are used to find synonymies relationships.

We have detected the presence of cycles in the similarity search graph. There is a major cause for cycles, the way special properties and classes can be combined are not acyclic graphs themselves. As a result, the cycles existing in the ontology become cycles in the similarity search graph, it means, a descendant of a concept could be simultaneously an ancestor of this concept. For instance, in our similarity searching method, the similarity value of a pair of classes $A$ and $B$ depends on the resolution of the similarity value of a pair of classes $C$ and $D$ which depends on the similarity values of $A$ and $B$.

Because the ontology graph contains cycles, precautions must be taken for avoiding loops in the graph traversal. It is necessary for the method to eliminate cycles. In addition, once a cycle is found with a graph's node a partial similarity value has to be calculated in order to go on with the analysis. First of all, our approach for detecting cycles is to mark as visited the nodes while covering the graph in order to avoid visiting twice the same node. This is a very simple and effective approach. So, in Figure 3 the sentence $if(c1, c2) \notin Visited$ verifies whether the node is in the analysis path. If this condition is true, then the algorithm takes places again for the range classes with the purpose of finding the $sim_{total}$ value. On the other hand, $if(c1, c2) \in Visited$ means that the node has been visited so that no special properties will be analyzed.

Although a cycle may be detected in a node graph, the similarity search algorithm must still go ahead and return a partial value for that node. In this case the similarity value, will

be obtained only from the information available on the node itself. For instance, semantic and syntactic analysis over the concepts will be carried out but no structural analysis over the special properties will be done.

Finally, the method compares the classes. This comparison is made using the syntactic functions for common and attribute classes and the semantic function for common classes. The semantic function uses the mappings added by the property comparisons in order to denote the set of similar attributes of both classes. A temporal mapping is added if the final function exceeds the threshold. Notice that in the final similarity value for any pair of classes $sim_{class}(c1, c2)$, the operand $w_{propiedadtes\_especiales} * sim_{propiedadtes\_especiales}$ ought to be null if a cycle was detected on $(c1, c2)$. Thus, the similarity threshold is lower in this case with the intention of giving more chances to the user to decide over the mapping of this classes.

Once all similarity values are obtained for two classes, the temporal mappings are displayed to the user and he/she must decide if these mappings must be added permanently. Thus, the user makes the final decision.

One last thing to point out over the method is the check done before the process starts again over the range classes. Taking into account that either of the ontologies having a lot of classes and properties will generate a large graph, a question whether a pair classes are already in the defined mappings set is stated with the $if(range\_of(sp_i), range\_of(sp_j)) \in Mapped$ sentence. If they have been mapped, then the found value is taken to avoid doing the same analysis again.

## 3 Architecting a Supporting Tool

As an implementation of our three-level approach for ontology integration, we built a plug-in for the Protégé ontology editor [1]. Basically, at its core, Protégé implements a rich set of knowledge-modelling structures and actions that support the creation, visualization, and manipulation of ontologies in various representation formats. Further, Protégé can be extended by way of a plug-in architecture and a Java-based Application Programming Interface (API) for building knowledge-based tools and applications. The Protégé platform supports two main ways of modelling ontologies through the following editors *Protégé-Frames* y *Protégé-OWL*. In this section, we will describe the design of the Protégé plug-in called OWLSim.

During the design process, the Responsibility Driven Design (RDD) [20, 19] model was used to keep our focus on the behavior of our software. This methodology helps us to identify the application's responsibilities and to divide them into collaborative objects.

The plug-in's architecture design is based on two architectural styles the model-view-controller (MVC) and its successor the presentation-application-control PAC [2, 6]. We describe how software objects are organized, this means how objects are located in components. Each component contains characteristic object roles that are located according to both the application components' functionality and the object's roles.

Figure 4 shows our plug-in's architecture. In the first place, the Transactions (*Control and business logic*) component includes the objects that are responsible for the control and business logic. Furthermore, this component mediates the interaction between Domain Model and Presentation in order to avoid direct dependency between them. Secondly, the Domain Model (*Abstraction*) component contains all those objects that represent the domain concepts. Lastly, the User interaction component (*Presentation*) is structured into objects that provide window, menu, and dialog functionality. They manage the inputs and translate them into service requirements.

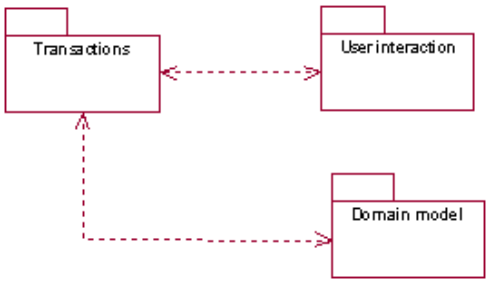


**Fig. 4.** Plug-in's architecture

### 3.1 Designing classes

One of the most important activities in Object-Oriented design is to identify object classes. Thus, we move from the requirements and descriptions – the method specification explained in previous sections (behavior that the plug-in must accomplish), the definition of OWL ontologies and the Protégé model specification – to find and describe the most important classes.

**Domain and application-specific objects** Domain objects represent concepts in a specific field of interest. In our domain, the ontologies and their elements, the mappings between them, and the similarity method.

Figure 5 shows part of the domain class diagram using UML notation [8] – the part where the ontology and both their elements and mappings are modelled. There are classes that model the most important components of an OWL [16] ontology, such as classes, properties and restrictions. In addition, as described in section 2, we also take into account the division of the ontology elements that the similarity method embodies (*Atributte_Class*, *Common_Class*, *Datatype_Property* and *Special_Property*). As shown in the diagram, both classes *Atributte_Class* and *Common_Class* were modelled as a class *Class* specialization. Because of their differences, the method gives different treatment to each of them. In addition, the *has_superclass* relation represents taxonomic relations in an ontology. Regarding class *Property*, it has a specialization into two classes: *Datatype_Property* and *Special_Property* in accordance to the method's division on property elements, as only *Special_Property* relates the *classes*. Moreover, both special properties and datatype properties ranges are different, so two distinct relations *has_classrange* and *has_XMLDatatyperange* were modelled to associate them to the range classes *XMLDatatype* and *Class*. Further, the *has_compatibility* association shows that each data type might be compatible with other data types. Unlike *Common classes*, that might have both types o properties, the *Attribute classes* have not gotten any property; thus, the association *has_property*is between the subclass *Common_Class* and the superclass *Property*. Finally, properties restrictions are also modelled.

Following, the mappings found by the method are contained in the *Mapping* class. The *Property_Mapping* and *Class_Mapping* classes are its subclasses. The former class involves classes using the *has_classes* and the latter class involves properties using the *has_properties*.

As we shift our view from modelling ontology concepts to the three-level approach method, we find the *Similarity-Searcher* class. This, is an abstraction from the similarity searcher method. It is subclassified into three subclasses *DataTypePropertyAnalysis*, *SpecialProperty-Analysis*, *ClassAnalysis*, each of them representing the analysis method part over the following elements identified in an ontology (Figure 2).

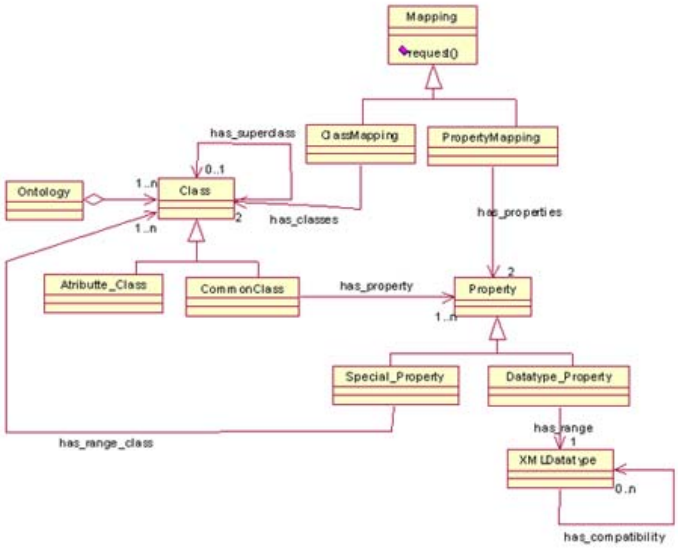


**Fig. 5.** Plug-in class design diagram part

Similarly, there are in the design plenty of other classes that are needed objects to translate the computer's user inputs to commands to the right objects in the application. For brevity reason we only explained the Similarity Searcher class which is the core of our method.

### 3.2 Similarity searching

As a result of being too complex to be implemented by a single object, the *SimilaritySearcher* class main responsibility is divided into subresponsibilities reassigned to collaborating objects. Each object implements a quite different similarity search method depending on three elements as above mentioned. The *SimilaritySearcher* object coordinates these collaborating objects, as Figure 6 shows. If the ontology classes to be analyzed are *Common_Class* then datatype and special property analysis would take place, following the class analysis is carried out.

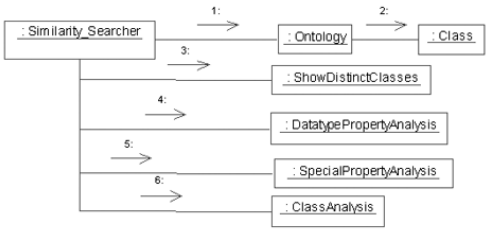


**Fig. 6.** Symilary_Searcher collaborates with other objects to find similarities.

Those specialized objects, which implement the subordinated responsibilities that the *Similarity Searcher* object delegates as well as coordinates are: *DataType Property Analysis*, *Special Property Analysis*, *Class Analysis*.

Each of these classes implements the different parts of our method. For instance, the DataType Property Analysis component implements the comparison between datatype properties applying all the functions (syntactic and semantic) described in Figure 3.

## 4 A Case Study - Using the Plugin

Now, we will present a case study that lets us show both the plug-in interface and how the method works. In addition, it let us point out some interesting results depending on the chosed ontologies for mapping. The following two ontologies were selected and created with the Protégé editor, they are shown graphically in Figures 7 and 8.

- The first ontology is called *"Travel Ontolgy"* [1]; it models flights, air agencies, car rental, hotels among other concepts. It has around 40 classes with many properties in order to describe their semantic and structure – they may be special properties or data type properties. From this ontology we only show the *Airport* class and their taxonomical relationships, and properties and restrictions that apply to the related entities, the ones involved in the example.
- The second ontology is called *"Location Ontology"* [2] which possess five classes and a number of properties, both special and datatype properties, to represent a location domain.

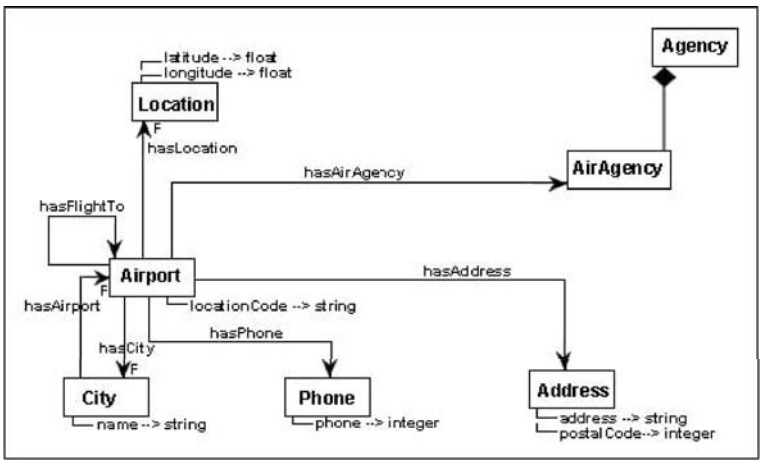


**Fig. 7.** *"Travel ontology part."*

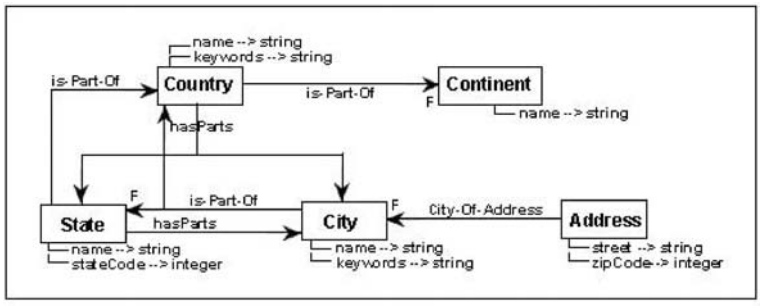


**Fig. 8.** *"Location ontology part."*

There are some points to notice in the two ontologies. Firstly, in both *Travel* and *Location* ontology graphs there are cycles. It means, that there is a pair of special properties that made

[1] www.ilby.net/travel.owl
[2] http://www.liacs.nl/CS/DLT/pickups/sjoerd/for%20Protege/Science.zip

some class both range and domain class from each other at the same time. In the former, the *hasFlightTo*, *hasAirport* and *hasCity* special properties involving the classes *Airport* and *City* as their ranges and domains. In the other one, for instance, the *hasParts* and *is_Part_Of* involves the classes City and Country as ranges and domains classes from one each other. Nevertheless, our method will detect this and find out a similarity value too. Secondly, regarding to the *City* and *Address* classes modelled in each of the ontologies, they are represented quite different. So the similarity values will not be high in spite of been the same entities. Later we will discuss examples on this points.

Let us now look at the plug-in's interface where we choose the .owl files containing the mentioned ontologies. As Figure 9 shows, this form lets the user to indicate the ontologies to be compared in OWL language.

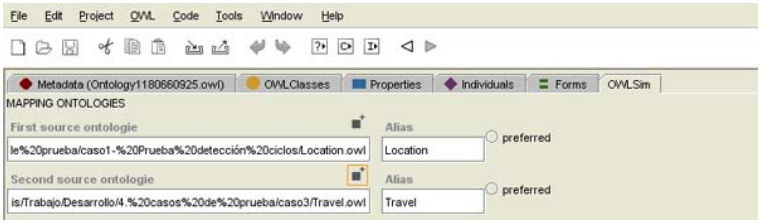


**Fig. 9.** *"OWLSim plug-in interface which lets us select the owl files to map."*

Following, if both owl files are loaded successfully, the mapping layout screen appears (see Figure 10). It is divided into two main panels. On the left side, there is the *select source classes form* that holds each of the selected ontologies class hierarchy, so that a pair of classes to be compared could be choosen. On the other side, the right one, there is the *show class and property mappings form*. On this window, the mappings between classes and properties found by the procces are shown.

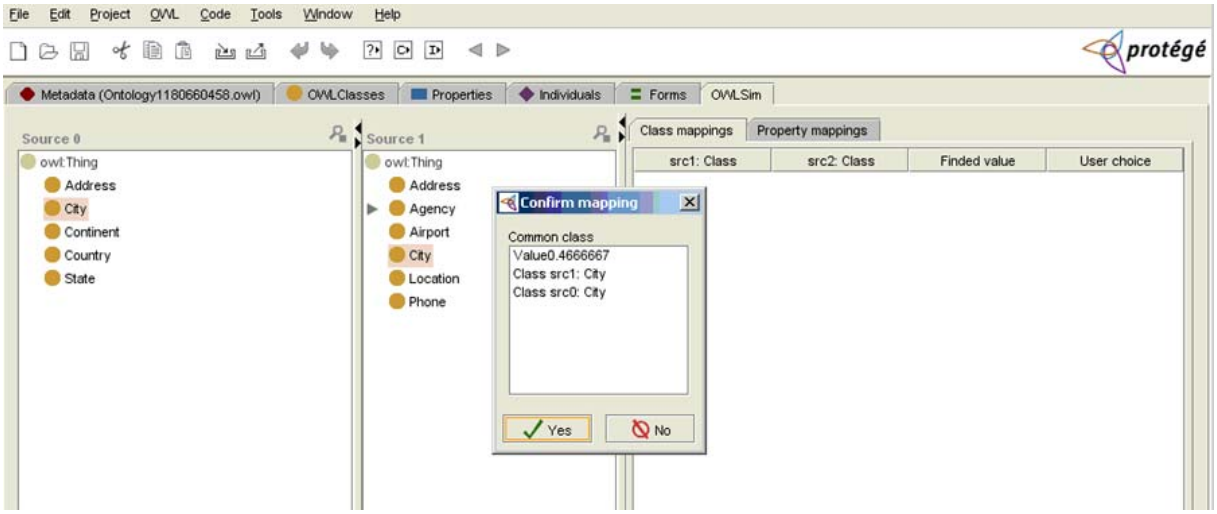


**Fig. 10.** *"OWLSim plug-in interface shown while the mapping process takes place."*

Once two classes are selected to be compared, the analysis takes place over those selected concepts by clicking the *"Mapp selected"* button. As a result, for any mapping that is found,

a confirmation is required from the user through the confirmation window as shown previously in Figure 10.

Let us now look at two examples, which show the similarity values found when the two points described before occur while using the plug-in. First, when the *Location.City* and *Travel.City* classes are selected to be compared the values shown in Figure 11 will be obtained as the method goes through the special properties graph. As the similarity searching progresses it will find a cycle when the City classes are visited again. The method detects that the node is in the searching path. As a result of the special treatment to this node, we can see that the similarities between the data type properties *cityKeyWords*, *name* y *name_2* (stand for Location.City.name) are calculated. The similarity value from the two last ones only is higher than the acceptance threshold. Moreover, the syntactic and semantic values are equal to one. Thus, the structural comparison regarding the data type properties, *_simAttDtatypeProperties*, is quite low. Moreover, because of being a cycle node, the *_simAttSpecialProperties* is equal to cero. From this, the final value just barely exceeds the acceptance threshold.

```
Go on with the next child: City - City
32. Visited node [City vs City ]
dtp> sim value: cityKeyWords-name==>0.26666668
dtp> sim value: name_2-name==>0.7083334
32. CYCLE IN: City - City
_simThesaurus: 1.0
_simSint: 1.0
_simAttDatatypeProperties : 0.26666668
_simAttSpecialProperties: 0.0
ca-common> sim value: City-City===>0.53333336
```

**Fig. 11.** *"Similarity values obtained by comparing* City *concepts from both ontologies."*

In the second place, Figure 12 shows the results obtained from the analysis of both *Address* classes. They match syntactically as well as semantically; consequently, the *_simThesaurus* and *_simSint* values are equal to one. However, they are represented quite differently. On the one hand, the *Location.Address* class has two data type properties, *street* and *zipCode* in addition to a special property *city-of-address*. On the other hand, the *Travel.Address* class is represented with two data type properties, *address* and *postalCode*. Although, the address and street properties meant to represent the same information about classes, they are represented rather different. Hence, it will afect the final result. Futher, the comparison of the *zipCode* and *postalCode* properties is the only one that results in a grater value. Consequently, there is just one common attribute between these classes. The final similarity value is higher than the acceptance threshold so a *temporal* mapping is found, and this result will be prompted to the user. Then, if the user confirms this mapping it will be changed to *permanent*.

```
1. Visited node [Address vs Address ]
dtp> sim value: zipCode-postalCode==>0.4625
dtp> sim value: zipCode-address==>0.14583333
dtp> sim value: street-postalCode==>0.14285715
dtp> sim value: street-address==>0.31666666
_simThesaurus: 1.0
_simSint: 1.0
_simAttDatatypeProperties : 0.2
_simAttSpecialProperties: 0.0
ca-common> sim value: Address-Address===>0.47500002
```

**Fig. 12.** *"Similarity values obtained by comparing* Address *concepts from both ontologies."*

## 5 Conclusions and Future works

In this paper, we have presented our three level approach for searching mappings between two OWL ontologies, and aspects such as cycles in the ontologies and performance have been considered to avoid possible problems in the application of the method. An implementation of it as a Protégé plug-in has been presented and it will be soon available to be downloaded from Internet [3].

However, currently our work is in a development stage for a number of tasks that are still being developed. Since our current method only deals with one-to-one relationships, we are improving the similarity functions in order to consider many-to-many relationships. Some efforts has been presented in [15].

## References


1. Protégé. http://protege.stanford.edu/doc/users_guide/index.html, 2000.
2. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley Longman Publishing Co., Inc., Masachusetts, USA, 1998.
3. A. Buccella, A. Cechich, and N. Brisaboa. Ontology-based data integration: Different approaches and common features. In L. Rivero, J. Doorn, and V. Ferraggine, editors, *Encyclopedia of Database Technologies and Applications*. Idea Group, 2005.
4. A. Buccella, A. Cechich, and N. R. Brisaboa. Ontology-based identification of similarity among heterogeneous sources. *Journal of Computer Science and Technology*, 6(1):62–68, 2003.
5. A. Buccella, A. Cechich, and N. R. Brisaboa. A federated layer to integrate heterogeneous knowledge. In *VODCA'04 First International Workshop on Views on Designing Complex Architectures*, number 142 in Electronic Notes in Theoretical Computer Science, Elsevier Science B.V, pages 79–97, Bertinoro, Italy, September 2004.
6. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerland, and M. Stal. *Pattern Oriented Software Arquitecture - A systema of Patterns*. Wiley and Sons Ltd., West Sussex, England, 1996.
7. S. Busse, R. Kutsche, U. Leser, and H. Weber. Federated information systems: Concepts, terminology and architectures. Technical Report Nr. 99-9, Technical University of Berlin, 1999.
8. Martin Fowler and Kendall Scott. *UML distilled (2nd ed.): a brief guide to the standard object modeling language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
9. Y. Kalfoglou and M. Schorlemmer. Ontology mapping: the state of the art. *The Knowledge Engineering Review*, 18(1):1–31, 2003.
10. D. Lin. An information-theoretic definition of similarity. In *Proceedings of the Fifteenth International Conference on Machine Learning*, pages 296–304, July 1998.
11. A. Maedche and S. Staab. Measuring similarity between ontologies. In *Proceedings of the EKAW'02 13th International Conference on Knowledge Engineering and Knowledge Management. Ontologies and the Semantic Web*, pages 251–263, London, UK, 2002. Springer-Verlag.
12. D. McGuinness, R. Fikes, J. Rice, and S. Wilder. An environment for merging and testing large ontologies. In *Proceedings of the KR*, pages 483–493, 2000.
13. Natalya Fridman Noy and Mark A. Musen. Prompt: Algorithm and tool for automated ontology merging and alignment. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 450–455. AAAI Press / The MIT Press, 2000.
14. M.A. Rodríguez and M.J. Egenhofer. Comparing geospatial entity classes: An asymmetric and context-dependent similarity measure. *International Journal of Geographical Information Science*, 18(3):229–256, 2004.
15. S. Roger, A. Buccella, A. Cechich, and M. S. Palomar. Asematch: A semantic matching method. In *TSD'06: Ninth International Conference on Text, Speech and Dialogue*, pages 229–235, Brno, Czech Republic, September 11-15 2006.
16. M. K. Smith, C. Welty, and D. McGuinness. Owl web ontology language guide. W3C, February 2004.
17. A. Tversky. Features of similarity. *Psychological Review*, 84(4):327–352, 1977.
18. H. Wache, T. Vögele, U. Visser, H. Stuckenschmidt, G. Schuster, H. Neumann, and S. Hbner. Ontology-based integration of information - a survey of existing approaches. In *Proceedings of the IJCAI-01 Workshop: Ontologies and Information Sharing*, pages 108–117, Seattle, WA, 2001.
19. R. Wirfs-Brock and A. McKean. *Object Design: Roles, Responsibilities, and Collaborations*. Addison-Wesley 2003, 2003.
20. R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software*. Prentice Hall 1990, 1990.


[3] http://protege.cim3.net/cgi-bin/wiki.pl