A Federated Layer to Integrate Heterogeneous Knowledge

Agustina Buccella\textsuperscript{1}  Alejandra Cechich\textsuperscript{2}

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

Nieves R. Brisaboa\textsuperscript{3}

Departamento de Computación
Universidade de A Coruña
A Coruña, Spain

Abstract
The way developers define architecture, execute architectural strategy, and record the results make a critical difference in the ability to deal with information and knowledge. In this context, integrating databases is very important indeed, but the different semantics they possibly have usually complicates administration. Therefore, recovering information through a common semantics becomes crucial in order to realise the full knowledge contained in the databases. In this paper, we describe and illustrate a proposal on the use of layered architectures to integrate knowledge from heterogeneous sources. We illustrate how the process might be facilitated by applying ontology-based comparisons as part of the components’ behaviour.

Keywords: Federated Databases, Ontology, Semantics Heterogeneity, Contexts.

\textsuperscript{*} This work is partially supported by the CyTED project VII-J-RITOS2 and the UNCOMA project 04/E048.

\textsuperscript{1} Email: abuccel@uncoma.edu.ar
\textsuperscript{2} Email: acechich@uncoma.edu.ar
\textsuperscript{3} Email: brisaboa@udc.es
1 Introduction

Software system developers are increasingly recognizing the importance of exploiting existing knowledge in the engineering of federated databases and federated information systems. One way to do this is to define an architecture that determines a partitioning of system design elements and rules for their composition. For example, the architecture proposed in [6] (based on [23]), introduces three main layers: the “wrapper layer”, which involves a number of modules to retrieve data from underlying sources hiding their data organization; the “interface layer”, which allows users to access the federated system; and the “federated layer”, which is in charge of solving problems related to semantics heterogeneity.

These problems are classified in [9] as: (1) aspects dealing with semantically equivalent concepts – different models use different terms to refer to the same concept, e.g. synonyms, or properties are modelled differently by different systems; (2) aspects dealing with semantically unrelated concepts, e.g. the same term may be used by different systems to denote completely different concepts; and (3) aspects dealing with semantically related concepts by using generalization/specialization, etc. A similar classification of semantics heterogeneity can be found in [7].

So far, different approaches have been used to model the federated layer. They are so diverse as complementary in some cases, and can involve different perspectives such as the use of ontologies [5], the use of metadata [6,16,22], etc. For example, the work in [18] presents a characterization of the ontology integration process focusing on the analysis of the quality of the ontologies as a way of obtaining a reusable resulting ontology. As another example, the Chimarea approach [15] provides support for merging ontological terms of different sources, checking the coverage and correctness of ontologies, and maintaining ontologies over time. As a last example, the FCA-MERGE method [26] describes a bottom-up technique for merging ontologies based on a set of documents.

Our work is focused on the ontological heterogeneity problem [29] when modelling the federated layer. This problem appears when the mapping between the source ontologies and the shared vocabularies must be performed. The ontological heterogeneity has a series of inherent problems because each ontology corresponds only to one information source created independently.

To address these problems, we have proposed a federated architecture based on a hybrid ontology approach [8], and we have also defined a method to populate the architecture [2,3]. Our method contains three main stages:
building the source ontologies, building the mappings among source ontologies and building the shared vocabulary. Each of these stages serves as a guideline to create the architectural components.

In this paper, we focus on the second stage, building the mappings among source ontologies, explaining its steps in order to understand how architectural components behave to deal with ontological heterogeneity problems. Particularly, we describe how similarity of concepts might be calculated.

This paper is organized as follows: Section 2 presents a summary of our architecture, especially describing the component in charge of calculating similarity of concepts. Then, Section 3 presents an example in OWL illustrating how the component works. Future work and conclusions are addressed afterwards.

2 Components to Solve Semantics Heterogeneity

Figure 1 shows the main components of our federated system architecture. As the wrapper layer and the interface layer have the same functionality as presented in [6], in this section we only focus on the federated layer presented in [2]. This layer is composed of three main components: source ontologies, OCM (Ontology and Context Mapping) and shared vocabulary.

Let us briefly clarify these concepts. For each information source within the federated system, one source ontology and a specific context are specified. Also, a set of contexts is defined within each ontology describing the different roles of one database. For example, the use cases of a UML specification [10] might be the source to obtain some of the contexts.

The second component, OCM, deals with the relationships among the contexts and concepts of the different source ontologies. These relationships are equality, inclusion, intersection, etc. Therefore, the OCM deals with the information flow between the source ontologies and the shared vocabulary.

Finally, the shared vocabulary is the component in which all source ontologies converge. This component is composed of the generic concepts and the context that will be used to query the system. Users use this vocabulary to query and get answers from the system. Thereby, the system gets access to the information sources to produce the output data.

In [2,3], we have also proposed a method to create the three components of our federated architecture. This method contains three main stages: building the source ontologies, building the mappings among source ontologies and building the shared vocabulary.

Particularly, the second stage, building the mappings among source ontolo-
gies, contains three main steps: defining the mapping, searching for similarities and building the equality axioms. The first step implies defining the relationships among the contexts of the source ontologies built in the previous stage. As the contexts are defined globally, this is a straightforward step. The second step, search for similarities, is the most important step and the main focus of this paper. It will be described in Section 2.2. Finally, the last step, building the equality axioms, also is a straightforward step because high similarity values must be looked for in the related context.

2.1 Architectural components of the federated layer

In this section, we briefly describe the structure of software components used to implement the federated layer, and particularly the calculation of similarities. Following, some components are described in terms of their interfaces and sub-components, and others are only mentioned for brevity reasons. We refer the reader to [4] for a more detailed description. Figure 2 presents a diagram showing the components and their dependencies, where the structure is represented using the UML notation [10].
• The Coordinator Component: The intent of this component is to coordinate all the processes accordingly by using each component at a time. Once the ontology is loaded by the user (in OWL Language [24]), the Coordinator calls the Parser and Instantiation Component to obtain an object structure (representing an instantiation of the Ontology Model Component) as a result. In this way the whole ontology, its common and attribute classes and its special and datatype properties, will be objects of the Ontology Model. In order to calculate the similarity values among the concepts included in the related contexts, the Similarity Searcher Component is invoked.

• The Parser and Instantiation Component: The component should parse the OWL code loaded by the user in order to create an object structure which represents a valid instantiation of the Ontology Model Component. Besides, error codes generated during the parsing process or the creation of an instance are returned to the Coordinator Component. Users should use some ontology editor such as Protégé [25] to avoid syntactic problems.

• The Ontology Model Component: This component corresponds to the Java translation [27] of the Ontology. We use the syntax of OWL, where an ontology contains classes and properties according to our classification (Section 2.2).

4 Other similar languages might be also used.
Figure 3 shows the class diagram used in the ontology instantiation. Each user can create his own ontology involving the user and ontology classes. The “ontology class” class involves common and attribute classes (using the “has attributes” association) and super/subclasses (using the “has superclass” association). The properties are contained in the property class. The special and datatype properties are subclasses of property class. Note that ranges of these subclasses are different. The special property has one or more “ontology class” as range and the datatype property has one “XML datatype” as range. Besides, the “ontology class instance” class has the instances of the classes and the “i values” subclass has the instances of the properties.

A dependence relationship exists between this component and the Context Model Component because one association with the “ontology class” class within the first component is required. This association indicates the classes contained in the contexts.

- The Context Creator Component: This component has the responsibility of creating the object structure which represents a valid instantiation of the Context Model Component corresponding to the context and its relationships loaded by the users.
- The Context Model Component: This component corresponds to the Java translation of the context defined by the users. It has the classes and properties included in each context together with the relationship among the contexts.
- The Similarity Searcher Component: This component has the task of calculating the similarity values within two related contexts. It uses the Ontology Model Component in order to obtain the common and attribute classes, and the special and datatype properties, and to use them in our similarity method (Section 2.2 – Figure 5). The Context Model Component is used to obtain the concepts included in the related contexts.

2.2 The Similarity method

Finding similarities is a very complex activity because in general it is not possible to determine fully automatically all mappings between two ontologies – primarily because of the ontological heterogeneity problems (synonyms, different classifications, etc.). The similarity functions we propose in this paper

---

5 In this context the terms “ontology class” and “ontology class instantiation” denote the classes and the instances of the ontologies. They do not denote the instances of a class of an Object Oriented paradigm but similar terms with different meanings.

6 In this paper, instances are not used for comparison.
should therefore only determine mapping candidates, which a user can accept, reject or change. Furthermore, the user should be able to specify mappings for concepts for which the system was unable to find satisfactory match candidates.

We propose to use several similarity functions depending on the ontology’s elements. Following, we describe all the functions used to compare concepts. Then, we show our similarity method in which these functions are applied.

The concepts of an ontology can be compared using two comparison levels: syntactic and semantic. Within the syntactic level we use three similarity functions: the edit distance function [12,14], the trigram function [13] and the data type function. The edit function \( (1) \) returns a degree of similarity between 0 and 1, where 1 stands for perfect match and 0 for bad match. Given two strings, it considers the number of changes that must be made to turn one string into the other and weighs the number of these changes against the length of the shortest string.

\[
\text{sim}_{ed} = \max \left( 0, \frac{\min(|x|, |y|) - ed(x, y)}{\min(|x|, |y|)} \right) \in [0, 1]
\]

For example, \( ed(\text{animal}, \text{animals}) = 1 \), because one insert operation changes
the string “animal” into “animals”, min(|animal|, |animals|) = min(6,7) = 6, therefore \( sim_{ed}(\text{animal}, \text{animals}) = \max(0, 5/6) = 5/6 \)

The trigram function (2) is based on the number of different trigrams in the two concepts or strings:

\[
\text{(2)} \quad sim_{tri}(x, y) = \frac{1}{1 + |\text{tri}(x)| + |\text{tri}(y)| - 2 \times |\text{tri}(x) \cap \text{tri}(y)|}
\]

where \( \text{tri}(x) \) is the set of trigrams in \( x \). For example,

\( \text{tri}(\text{animal}) = \{\text{ani, nim, ima, mal}\} \) and

\( \text{tri}(\text{animals}) = \{\text{ani, nim, ima, mal, als}\} \), so

\( \text{tri}(\text{animal}) \cap \text{tri}(\text{animals}) = \text{tri}(\text{animal}) \), and therefore

\( sim_{tri}(\text{animal}, \text{animals}) = 1/(1 + 4 + 5 - 2 \times 4) = 1/2. \)

Finally, the data type function or datatype compatibility (3) is a straightforward function because it only compares the data types of the two concepts. For example: string to string or string to integer. If there exists a logical conversion [1] of a data type into another the result of the function is equal to 1, otherwise it is equal to 0.

\[
\text{(3)} \quad sim_{dtc}(dt_1, dt_2) = \begin{cases} 1 & \text{if } \text{datatype compatibility}(dt_1, dt_2) \\ 0 & \text{otherwise} \end{cases}
\]

At the semantic level, to compare two concepts based on their attributes, we use the similarity function described in [2]. Also, we use the information thesaurus provided in order to find synonymy and hyponymy relationships. Then, the similarity function (4) to compare two concepts semantically is [2,21]:

\[
\text{(4)} \quad sim_{att}(x, y) = \frac{|X \cap Y|}{|X \cap Y| + |X/Y| \cdot \frac{1 - \alpha(x, y)}{\alpha(x, y)} \cdot \frac{|Y/X|}{|Y/X|}}
\]

for \( 0 \leq \alpha \leq 1 \)

where \( x \) and \( y \) are concepts and \( X \) and \( Y \) correspond to description sets of \( x \) and \( y \), in this case “attributes”. Function (4) is based on Tversky’s model [28], in which the function \( \alpha \) identifies the most common superclass between
two concepts and calculates their depth in a hierarchy. If the depth of two concepts is the same, the value of this function is equal to 0.5.

For example, if the \textit{animal} concept is described by three attributes (\textit{color}, \textit{weight} and \textit{age}), and the \textit{animals} concept is also described by three attributes (\textit{color}, \textit{age} and \textit{mammal}), the terms of function (4) might be calculated as:

\[
|X \cap Y| = |\{\text{color, weight, age}\} \cap \{\text{color, age, mammal}\}| = 2;
\]
\[
|X / Y| = |\{\text{color, weight, age}\} / \{\text{color, age, mammal}\}| = 1;
\]
\[
|Y / X| = |\{\text{color, age, mammal}\} / \{\text{color, weight, age}\}| = 1;
\]

and in this case \(\alpha(\text{animal}, \text{animals}) = 0.5\). Therefore,

\[
sim_{att}(\text{animal}, \text{animals}) = \frac{2}{3}.
\]

In order to begin comparing concepts of two ontologies, we firstly classify the different concepts. Figure 4 shows how the different elements of an ontology are divided. The first division refers to two different elements. On one branch we have the \textit{classes} and on the other branch the \textit{properties}. Firstly we analyze the \textit{classes} branch, which is also divided into two new branches: \textit{common classes} and \textit{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 \textit{common classes} have the role of representing things about the domain and the \textit{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 \textit{Animal} class as a common class and the \textit{Organ} class as an attribute class because \textit{Organ} exists to describe a characteristic about a common class. The \textit{Organ} class has no properties.

On the other branch, Figure 4 shows the \textit{properties} branch which is also divided into two new branches: \textit{datatype properties} and \textit{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 \textit{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 \textit{Animal} class and the \textit{String} data type. On the other hand, the \textit{special properties} are properties relating classes. For example, the relationship between the \textit{Animal} class and the \textit{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.

Figure 5 describes our basic method for searching similarities. The method depends on the similarity functions described previously and on the different elements of the ontology as shown in Figure 4. 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 denotes the process of elimination of articles, prepositions and non-relevant characters (.,;,-, etc.). Thesauruses are used to search for synonymies. The function \( \text{sim1}_\text{thesaurus}(dtp_i, dtp_j) \) is equal to 1 if a synonym relationship is found for the two datatype properties and it is equal to 0 otherwise. Then, the \( \text{sim1}_\text{sint}(dtp_i, dtp_j) \) function is calculated using the similarity functions (1), (2) between the names of these properties, the data type compatibility \( \text{sim}_\text{dtc}(\text{range}(dtp_i), \text{range}(dtp_j)) \) and the result of the thesaurus. The \( \text{sim1}_\text{sint}(dtp_i, dtp_j) \) function returns a value between 0 and 1; and the sum of weights, the \( w \) values \( (w_{ed}, w_{tri}, w_{dt} \text{ and } w_{thesaurus}) \), is equal to 1. Finally, if the result of the function exceeds a threshold \( (t_{\text{accept}}) \), a temporal mapping is added.

Then, we must compare the special properties included in the common classes. The comparison is similar to the previous case, but the datatype compatibility is not calculated.

The \( \text{sim}_\text{total}(sp_i, sp_j) \) 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). Again, thesauruses are used to find synonymies relationships.

The \( \text{sim}_\text{rest}(sp_i, sp_j) \) function checks special property restrictions \( [24] \) such as functional, symmetric, allValuesFrom, someValuesFrom, cardinality, etc. Then, a temporal mapping is added when the \( \text{sim}_\text{sp}(sp_i, sp_j) \) function exceeds
Similarity\((O_1, O_2)\)
the user enters two similar classes \((c_1, c_2)\)
if \(c_1\) and \(c_2\) are common classes
for each datatype property \(dtp_i \in c_1\) and \(dtp_j \in c_2\)
cleaning_process\((dtp_i, dtp_j)\)
\(\text{sim}_{1\text{thesaurus}}(dtp_i, dtp_j) = \text{search_on_thesaurus}(dtp_i, dtp_j)\)
\(\text{sim}_{1\text{int}}(dtp_i, dtp_j) = w_{\text{ed}} \times \text{sim}_{\text{ed}}(dtp_i, dtp_j) + w_{\text{tri}} \times \text{sim}_{\text{tri}}(dtp_i, dtp_j) + w_{\text{dtc}} \times \text{sim}_{\text{dtc}}(\text{range_of}(dtp_i), \text{range_of}(dtp_j)) + w_{\text{thesaurus}} \times \text{sim}_{1\text{thesaurus}}(dtp_i, dtp_j)\)
if \(\text{sim}_{1\text{int}}(dtp_i, dtp_j) \geq \text{thaccept}\)
add_mapping\((dtp_i, dtp_j)\)
for each special property \(sp_i \in c_1\) and \(sp_j \in c_2\)
cleaning_process\((sp_i, sp_j)\)
\(\text{sim}_{2\text{thesaurus}}(sp_i, sp_j) = \text{search_on_thesaurus}(sp_i, sp_j)\)
\(\text{sim}_{\text{rest}}(sp_i, sp_j) = \text{check_restrictions}(sp_i, sp_j)\)
\(\text{sim}_{2\text{int}}(sp_i, sp_j) = w_{\text{ed}} \times \text{sim}_{\text{ed}}(sp_i, sp_j) + w_{\text{tri}} \times \text{sim}_{\text{tri}}(sp_i, sp_j) + w_{\text{thesaurus}} \times \text{sim}_{2\text{thesaurus}}(sp_i, sp_j) + w_{\text{rest}} \times \text{sim}_{\text{rest}}(sp_i, sp_j)\)
\(\text{sim}_{\text{total}}(sp_i, sp_j) = \text{calculate all the process for (range_of(sp_i), range_of(sp_j))}\)
\(\text{sim}_{\text{sp}}(sp_i, sp_j) = w_{\text{sint}} \times \text{sim}_{2\text{int}}(sp_i, sp_j) + w_{\text{total}} \times \text{sim}_{\text{total}}(sp_i, sp_j)\)
if \(\text{sim}_{\text{sp}}(sp_i, sp_j) \geq \text{thaccept}\)
add_mapping\((sp_i, sp_j)\)
using the added mappings \(\Rightarrow\)
cleaning_process\((c_1, c_2)\)
\(\text{sim}_{3\text{thesaurus}}(c_1, c_2) = \text{search_on_thesaurus}(c_1, c_2)\)
\(\text{sim}_{3\text{int}}(c_1, c_2) = w_{\text{ed}} \times \text{sim}_{\text{ed}}(c_1, c_2) + w_{\text{tri}} \times \text{sim}_{\text{tri}}(c_1, c_2) + w_{\text{thesaurus}} \times \text{sim}_{3\text{thesaurus}}(c_1, c_2)\)
if \(c_1\) and \(c_2\) are attribute classes
\(\text{sim}_{\text{class}}(c_1, c_2) = \text{sim}_{3\text{int}}(c_1, c_2)\)
if \(c_1\) and \(c_2\) are common classes
\(\text{sim}_{\text{class}}(c_1, c_2) = w_{\text{sint}} \times \text{sim}_{3\text{int}}(c_1, c_2) + w_{\text{att}} \times \text{sim}_{\text{att}}(c_1, c_2)\)
if \(\text{sim}_{\text{class}}(c_1, c_2) \geq \text{thaccept}\)
add_mapping\((c_1, c_2)\)

Fig. 5. Method for searching similarities

the threshold.

Finally, we must compare 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.

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

Related Work

Several similarity measures are found in the literature, each of them applied to a specific situation. For example, several similarity measures, which are used in applications such as information retrieval or word sense disambiguation, are based on the content of information of each term [11,13,19]. This content is defined as the number of occurrences of a term, or any child term, in the same hierarchy in a corpus. Therefore, the formulas proposed by these approaches are based on probabilistic values. For example, in [13] the similarity measure is not defined directly by a formula; rather, it is derived from a set of assumptions about similarity.

Other works that do not use corpus can be found in [14,17,20,21]. Particularly, the work in [21] presents a combination of two different approaches for similarity assessment: the feature-matching process and the semantic distance. This model uses three elements to compare concepts: parts, functions and attributes. The parts are structural elements of a concept (or class), such as “roof” and “floor” of a building; the functions represent the purpose of a concept; and the attributes correspond to additional characteristics of a concept. Function 4 and the following function are used by this approach:

\[
S(a^{O_1}, b^{O_2}) = w_p \cdot S_p(a^{O_1}, b^{O_2}) + w_f \cdot S_f(a^{O_1}, b^{O_2}) + w_a \cdot S_a(a^{O_1}, b^{O_2})
\]

for \( w_p, w_f, w_a \geq 0 \) and \( w_p + w_f + w_a = 1 \)

Function 5 is a sum of products, where \( w \) represents parts, functions and attributes (\( w_p, w_f, \) and \( w_a \) respectively). Each \( S_x(a, b) \) for \( x=p \) or \( x=f \) or \( x=a \), is compared using function 4 explained previously.

This approach cannot be mapped directly onto our approach because concepts are compared without thinking of underlying ontologies. Elements to calculate similarities can be obtained from other information sources such as WordNet [20]; however, similarity cannot be calculated automatically since functions associated to a concept are written in natural language. Another problem with this approach is that the ontologies have properties which do not either have parts or attributes, therefore functions associated to properties are the only available element for comparison (and here we face again the natural language problem).

The work in [30] presents an approach to define verbs based on a set of
shared semantic domains. Within one conceptual domain, similarity of two concepts is defined by how closely they are related in the hierarchy, i.e., their structural relations. Conceptual similarity between two concepts C1 and C2 is expressed as:

\[ ConSim(C1, C2) = \frac{2 \times N3}{N1 + N2 + 2 \times N3} \]

where C3 is the least common superconcept of C1 and C2; N1 is the number of nodes on the path from C1 to C3; N2 is the number of nodes on the path from C2 to C3; and N3 is the number of nodes on the path from C3 to the root.

The problem with this formula is that a concept within an ontology may not contain any superclass. Then, the common superclass between this concept and another one will be the root, and the result of the function 6 will be 0. Besides, like the last method presented, there is no way of comparing properties.

The work presented in [17] describes a semi-automatic method for the integration of heterogeneous database systems. This method consists of several phases that work on the creation of three dictionaries: a synonym dictionary, an homonym dictionary, and an object cluster similarity dictionary. The method uses background knowledge about concept names stored in a Lexical Synonym Property Dictionary – LSPD. In order to obtain this dictionary, the method first constructs a set of synonym pairs by intersecting a set of pairs of a standard thesaurus and a set of names of input schemes. Then, plausibility coefficients are added for each synonym pair found in the last step. These coefficients are defined by domain experts.

The method builds one graph for each database in the integration and analyzes the neighborhoods for detecting similarity of two concepts. Thus, this method works iteratively based on the similarities found on each phase. As it is defined for databases, several factors included in these types of information systems are used; for example, data type compatibility of the attributes, attribute domains, and attribute semantic relevance. This last factor refers to the contribution of an attribute in characterizing the concept in which it belongs to, for example non-primary key attributes are more specific to the semantics expressed by the concept than primary key attributes, because they are generally codes or numbers used to identify instances. As we can see, several of these factors are specific for database systems and cannot be applied to ontologies.
Finally, the work in [14] is similar to our method—a lexical and a conceptual layer are used to find similarities—but thesaurus or other sources of semantic information are not used. At the lexical level, the method uses the function 1 called \textit{lexical similarity measure} (SM). At the conceptual level, concepts are compared taking into account the taxonomies in which they appear. Authors use the concept of \textit{Semantic Cotopy} (SC) to define all the super- and subconcepts of a specific concept within a taxonomy. The following function is used to compare two concepts:

$$\text{TO}^1(L, O_1, O_2) = \frac{F_{1}^{-1}(SC(F(\{L\}), H_1)) \cap F_{2}^{-1}(SC(F(\{L\}), H_2))}{F_{1}^{-1}(SC(F(\{L\}), H_1)) \cup F_{2}^{-1}(SC(F(\{L\}), H_2))}$$

where $H_1$ and $H_2$ are taxonomies, and $F_{1}^{-1}(SC(F(\{L\}), H_1))$ and $F_{2}^{-1}(SC(F(\{L\}), H_2))$ are all the super- and subconcepts of the $L$ concept in both taxonomies.

A different function is used when the properties or relations must be compared. To do so, authors compare domains and ranges of the properties by using another concept called \textit{Upwards Cotopy} (UC), which defines all the superconcepts of a specific concept:

$$\text{CM}(C_1, O_1, C_2, O_2) = \frac{F_{1}^{-1}(UC(C_1, H_1)) \cap F_{2}^{-1}(UC(C_2, H_2))}{F_{1}^{-1}(UC(C_1, H_1)) \cup F_{2}^{-1}(UC(C_2, H_2))}$$

where $F_{1}^{-1}(UC(C_1, H_1))$ and $F_{2}^{-1}(UC(C_2, H_2))$ are superconcepts of $C_1$ and $C_2$ respectively.

Datatype properties are not considered by this method. As an improvement, our method compares these properties by analyzing datatype compatibility and syntactic similarity (an example of this comparison is shown in Table 1).

Special properties are compared in a very similar way to our method because domain and range are analyzed separately. However, we also take into account their syntaxes, look for thesauruses and consider property’s restrictions. All of these factors have influences when searching similarities because two ontologies may have two properties with the same domain and range but with different meanings.

Our method emerges as a combination of several proposals in the literature. Our proposal modifies some similarity functions to adequate to the information...
the ontologies provide. Then we merge these functions in a basic method in order to find the most suitable mappings. As we will show in the following section, our method allows a user to find several correct mappings, but it is not suitable for dealing with many-to-one mappings.

3 A Motivating Example

In order to illustrate how the Ontology Model component works, we will describe part of a case study on which we are currently working. Figure 6 shows its two ontologies represented by diagrams (for simplicity, we do not show the OWL code here). Arrows in the figure represent special properties of common classes (a source arrow represents the domain and a target arrow represents the range). The datatype properties are represented as attributes within the class definition, such as \( \text{name\_of\_property} \rightarrow \text{datatype} \). The first element is the name property and the second is the data type. Therefore, the domain of a datatype property is the class in which it is in and the range is the datatype element.

Both ontologies are modelling a library domain. As we can see, each ontology uses its own vocabulary to represent things in the domain. In both ontologies, \text{Country} and \text{Place} classes are attribute classes because they do not have special properties and they exist to describe characteristics of a common class.

\text{Book}, \text{Volume}, \text{Author} and \text{Authors}, are common classes. \text{Written\_by}, \text{nationality} and \text{origin} are special properties and \text{isbn}, \text{name}, \text{first\_name}, etc. are datatype properties.

Following, we show the results of comparing \text{Book} and \text{Volume} classes.
Table 1 shows the results of applying the similarity method (Figure 5) on the datatype properties of the \textit{Book} and \textit{Volume} classes.

<table>
<thead>
<tr>
<th>( x \rightarrow y )</th>
<th>( \text{sim}_{\text{sint}}(x, y) )</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{book}.isbn \rightarrow \text{volume}.isbn )</td>
<td>1</td>
<td>\text{add_mapping}(x, y)</td>
</tr>
<tr>
<td>( \text{book}.name \rightarrow \text{volume}.name )</td>
<td>1</td>
<td>\text{add_mapping}(x, y)</td>
</tr>
<tr>
<td>( \text{book}.number_of_edition \rightarrow \text{volume}.edition_number )</td>
<td>0.78</td>
<td>\text{add_mapping}(x, y)</td>
</tr>
<tr>
<td>( \text{book}.pages \rightarrow \text{volume}.number_of_pages )</td>
<td>0.29</td>
<td>None</td>
</tr>
</tbody>
</table>

Table 1
Similarities for data type properties

Then, the special properties are compared. In our example, there is only one special property to be compared (\textit{written\_by}). This comparison also generates the comparison of the \textit{Author} and \textit{Authors} classes and \textit{Place} and \textit{Country} classes together with their properties.

Thus, as \( \text{sim}_{\text{sp}}(\text{book}.\text{written\_by}, \text{volume}.\text{written\_by}) = 0.835 \), the mapping is added.

Finally, we must compare \textit{Book} and \textit{Volume} classes. This process uses the mappings added in the previous steps because properties adding a mapping are considered equal. Then, \( \text{sim}_{\text{att}}(\text{book}, \text{volume}) = 0.75 \) and \( \text{sim}_{\text{class}}(\text{book}, \text{volume}) = 0.565 \).

In the example above, we considered that weights (\( w \) values) in the similarity functions are evenly distributed and the threshold value is equal to 0.5. Only in the case of attribute classes we increased the weight of the thesaurus function (that looks for synonymy words) because these classes were only compared syntactically. For example, if we compare the \textit{Country} class with the \textit{Place} class using evenly distributed weights, the similarity function \( \text{sim}_{\text{sint}}(\text{country}, \text{place}) \) returns 0.37. Note that this result is very low taking into account that these classes are semantically equivalent. Therefore, to compare these types of classes we assign 0.5 to \( w_{\text{thesaurus}} \) and 0.25 to \( w_{\text{ed}} \) and \( w_{\text{tri}} \) in the \( \text{sim}_{\text{sint}}(c1, c2) \) function. Now, the result of the function \( \text{sim}_{\text{class}}(\text{country}, \text{place}) \) is 0.53 and a temporal mapping is added.

Table 2 shows the mappings generated by the similarity method taking into account the last modification, that is, increasing the weight of the thesaurus function (\( w_{\text{thesaurus}} \)) for the \( \text{sim}_{\text{sint}}(c1, c2) \) function when the classes are attribute classes. Note that the mapping between name and first name or last name was not found because our method generates only one-to-one mappings.
Data type properties | Special properties | Common and attribute classes
---|---|---
`book.isbn` → `volume.isbn` | `book.written_by` → `volume.written_by` | `Book` → `Volume`
`book.name` → `volume.name` | `author.origin` → `authors.nationality` | `Author` → `Authors`
`author.birthdate` → `authors.date_of_birth` | | |

Table 2
Mappings found by the similarity method

4 Conclusion and Future Work

We have introduced a layered architecture to deal with semantic heterogeneity problems. To do so, we have proposed to model a federated layer using ontologies and contexts, and we have described a particular component – the Ontology Model Component. It implements comparisons between heterogeneous sources by applying information the ontologies provide. The way comparisons are made allows a user to make decisions on similarity of concepts, based on a more complete and accurate description of the sources.

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 and hyponym relationships. We should also take advantage of information on the use of instances and consider other problems about ontological heterogeneity, such as aggregation-level mismatches.

Finally, our architecture needs to be empirically validated, so much research must still be done to demonstrate the applicability of our proposal.

References


[27] Sun, Java SE Platform. URL http://java.sun.com

