Managing an Evolving Shared Knowledge Repository: The Upgrade Process via Semantic Mediation

Saravanan Muthaiyah
George Mason University
E-Center for E-Business
Department of Computer Science
4400 University Dr
Fairfax, VA 22030, USA
smuthaiy@gmu.edu

Marcel Barbulescu
George Mason University
Learning Agents Center
Department of Computer Science
4400 University Dr
Fairfax, VA 22030, USA
mbarbulescu@gmu.edu

Larry Kerschberg
George Mason University
E-Center for E-Business
Department of Computer Science
4400 University Dr
Fairfax, VA 22030, USA
kersch@gmu.edu

Abstract

This research focuses on the problem of ontology change management in the context of evolving shared hierarchical knowledge repositories. We propose a framework for dealing with ontology evolution via semantic mediation. In particular we present a Multi-Agent System (MAS) architecture that would be deployed to manage changes in shared ontologies such as update, deletion and renaming of classes in a dynamic environment. The proposed mediation process will be a mixed-initiative effort involving both intelligent agents and domain experts. The goal of this research is to extend the currently ongoing work in ontology evolution by including hierarchical ontology structures using semantic mediation. To validate this approach and to provide proof-of-concept, several tools have been used such as Jena and Protégé. We demonstrate how this multi-agent approach can enable ontology evolution and change management by using the JADE platform.

Key-Words: - Ontology Evolution, Shared Knowledge Repository, Change Management, Semantic Mediation

1. Overview

Research in the area of ontology engineering has reached a considerable level of maturity and stability. With the advent of editing tools such as Protégé and knowledge representation models, this progress has grown exponentially. However, very little work has been done for the evolutionary aspects of ontologies. The Semantic Web is a distributed and collaborative environment where ontologies will naturally evolve and co-evolve as such research on ontology evolution becomes crucial. A lot of work has been done in the area of evolution of single ontologies, but very little has been done for the evolution of shared inter-organizational ontologies.

2. Shared Knowledge Repository

In order to facilitate knowledge reuse and in the same time to allow experts to express their knowledge, even if they don't completely agree on it, we propose a hierarchical repository structure. The knowledge will be organized at different levels, each ontology inheriting the knowledge from one or more parent ontologies. An ontology will

completely inherit the knowledge from its parent ontologies (e.g. no partial inheritance), so for multiple inheritance relationships it is very important that the knowledge inherited from parents to be consistent (i.e. no naming clashes).

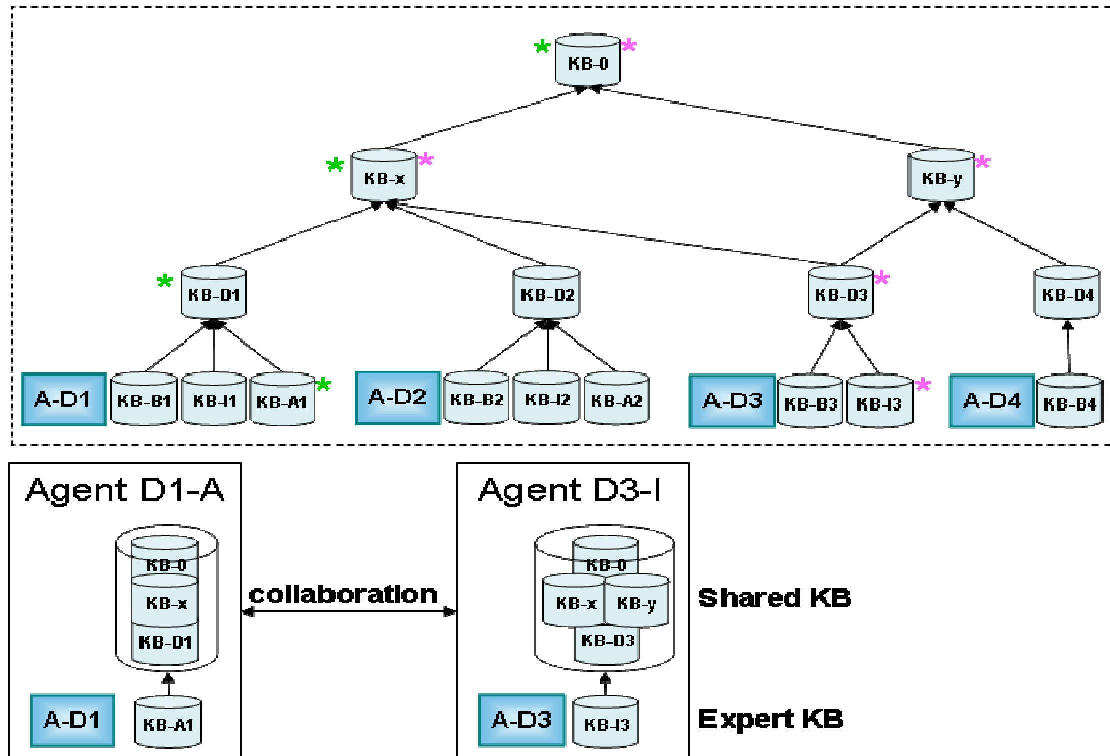


Figure 1. Shared Hierarchical Knowledge Repository

Figure 1 illustrates such a shared knowledge repository. There are four agents, (A-D1, A-D2, A-D3 and A-D4) that use parts of the shared knowledge and also have their own local developed knowledge. A-D1 has three local ontologies that incorporate different level of expertise: basic expertise in KB-B1, intermediate expertise in KB-I1 and advance expertise in KB-A1. All its local ontologies incorporate the knowledge from KB-D1, KB-x and KB-a. Agent A-D3 has local ontologies (KB-B3 and KB-I3) that incorporate knowledge from KB-D3, KB-x, KB-y and KB-a. When agent A-D1 and A-D3 collaborate, they know for sure that at least part of their knowledge is similar (e.g. KB-x and KB-a).

The reuse of knowledge is much easier with such a setup since when creating a new ontology we have a choice of parent ontologies that we can basically inherit from and start encoding knowledge on top of what already exists. Also, different contradictory pieces of knowledge can be encoded in different ontologies which are not in an inheriting relation (e.g. one is not a parent of the other). In the same time, the communication between agents is easier this way since they share common knowledge, so at least partially, they “speak” the same language.

2.1 Shared Distributed Knowledge Repository

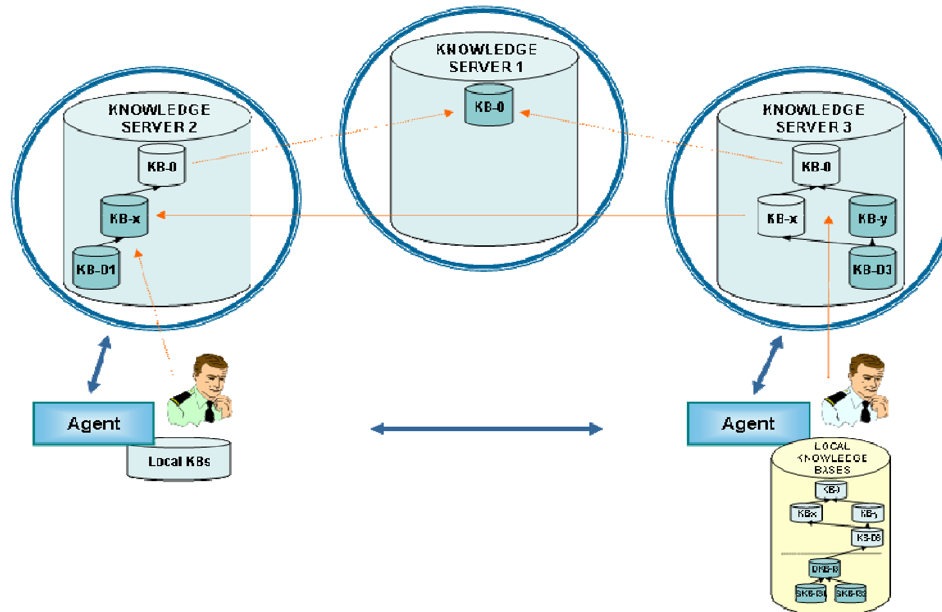


Figure 2. Shared Distributed Knowledge Repository

A hierarchical repository structure would help with the knowledge reuse, but nowadays is not realistic to consider that all the developed ontologies are under a central control and available at all times to everybody. Because of that, a distributed model of the hierarchical repository is more appropriate. Each ontology will have its own creator/maintainer and will be able to inherit knowledge from the ontologies created by others. To solve the availability problem, when a new ontology is inheriting knowledge from another ontology, a copy of the inherited knowledge will be available locally (Figure 2). This way the parent ontology do not have to be available online at all times in order to have all their children ontologies functioning properly.

2.2 Knowledge Evolution

An important part of the problem is how to handle the evolution of the individual ontologies. We introduce a versioning mechanism which enables sharing of static ontology versions. This way, when knowledge is inherited, it's actually the knowledge from a certain version (snapshot) of that ontology, which doesn't change over time. The maintainers can update their ontologies locally and release new versions when they are ready. At the same time, an ontology maintainer can decide to upgrade the inherited parent ontologies to newer versions.

2.2.1 Ontology Versioning

Given the hierarchical ontology repository from Figure 3, let's consider that we have multiple versions available for the following ontologies:

Table 1. Ontology Versions

Ontology	Versions (Inherit from)
KB-0	v1, v2
KB-x	v1 (KB-0 v1), v2 (KB-0 v1)
KB-y	v1 (KB-0 v1), v2 (KB-0 v2)
KB-D3	v1 (KB-x v1, KB-y v1)

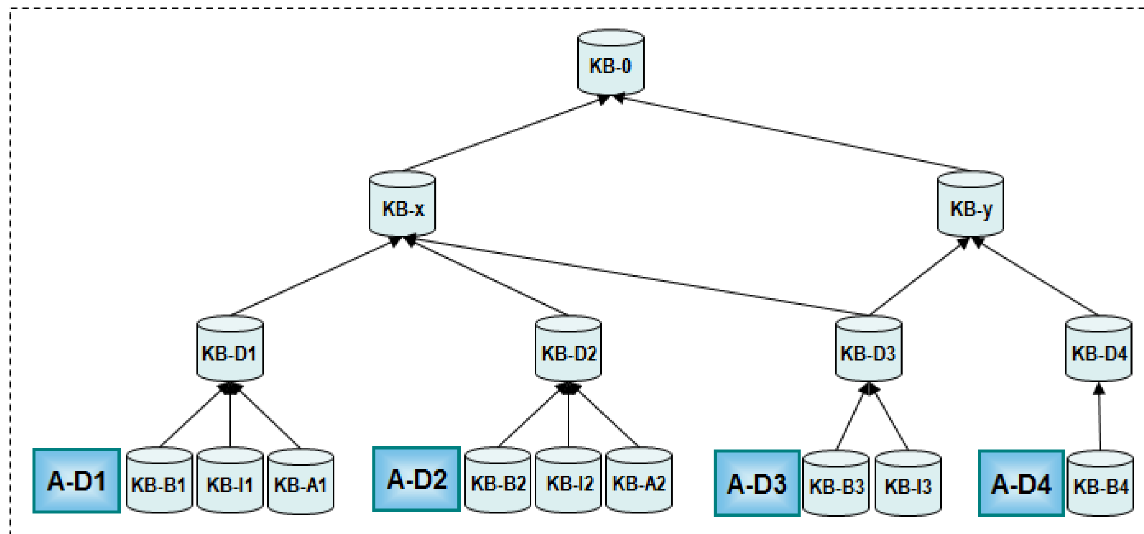


Figure 3. Example of a hierarchical knowledge repository

The direct parents of KB-D3 v1 are KB-x v1 and KB-y v1. If the maintainer of the KB-D3 decides to upgrade one of his parent ontologies, he may have to upgrade it to inherit from newer other parent ontologies as well, because of the version dependencies. Given the example above, the maintainer can update to inherit knowledge from KB-x v2 but cannot update to inherit from KB-y v2, since that version inherits from KB-0 v2 and there is no version of KB-x that inherits knowledge from KB-0 v2. Only one version of an ontology can exist at one time in such a hierarchy of inherited knowledge (i.e. KB-0 v1 & v2 cannot be inherited in the same time, directly or indirectly, by the same ontology).

After the maintainer selects a new set of versions of ontologies that he wants to inherit knowledge from, the next step would be to try to put together all the inherited knowledge. When one tries to upgrade to new versions of inherited ontologies, it may happen that more than one new version defines the same piece of knowledge (i.e. naming clash). This would make the two ontologies "incompatible" and make impossible to inherit from both of them in the same time. If the operation of putting together the inherited knowledge succeeds, the next step consists of updating the local defined ontology to accommodate the new inherited knowledge.

2.2.2 Evolution Log

In order to keep up with the changes that occurred from the previous versions, an evolution log can be very useful. Such a log would explicitly record the changes made to

an ontology so the changes will be easier to track in the ontology upgrading process. One example would be element renames. If the links between elements are made using their names instead of some unique id, changing names can be fatal to other ontologies inheriting that element. The new name of the element can be identified using such an evolution log so we can translate the local encoded knowledge into the new "world" without problems. In the literature review section we present the Change and Annotation Ontology (CHAO) [9] approach where the changes themselves are also represented in an ontology.

The changes in the new versions of the ontology can be of the following types:

I. New elements

The only problem introduced by new elements in a new version of an ontology is if their names/ids collide with elements locally defined somewhere else. If the multiple definitions are inherited, there is nothing that can be done; the inherited knowledge is "incompatible". In case the inherited knowledge collide with the local knowledge, there are two possibilities: the two elements are semantically different so the only option is to have the local element renamed since we cannot alter the inherited element; the other case is when a local element was added later to a newer version of the inherited knowledge so we don't need to define it locally anymore, since it's inherited from the new version.

II. Deleted elements

If inherited knowledge elements were used locally and deleted from the parent ontologies, a straight forward solution is to recreate locally the same piece of knowledge; another approach is to treat the operation the same way as if a local element is deleted and to adapt the local ontology.

III. Modified elements

This is a complicated situation and can be treated the same way modifications to local elements are treated, as if the modified piece of knowledge would be local.

2.2.3 Knowledge Reorganization

Imagine the case when there is limited inherited knowledge available and two or more ontologies start to develop the same kind of knowledge. One can suggest a piece of knowledge to be incorporated in an upper ontology and to become inherited knowledge for the others; a "sliced piece" of knowledge can be suggested to the maintainer of the upper ontology to be incorporated [2]; the maintainer may accept it, integrate (import) it and release a new version of the ontology. After this, the other ontologies that inherit knowledge from the original version can be upgraded.

2.3 Knowledge Import

An existing knowledge repository might need to be augmented with knowledge from other existing monolithic ontologies or even with knowledge from different systems using a different knowledge representation model. In the latter case, an extra step is required to perform a knowledge translation operation into your own knowledge

representation space. One way to start importing knowledge is to build a list of terms that you want to import knowledge about (for example things like "government" and "weapon"). In general, knowledge elements are highly connected through semantic links; one can build an algorithm to extract the desired terms and related knowledge by "slicing" it, by following the links as far as a set maximum depth [2].

If the maximum depth is not set, we're probably prone to import the whole ontology, since there is a big probability that everything is linked to everything else, a way or another. An example would be to look for the "weapon" class, take all the direct super-classes, direct sub-classes, instances and import them into the desired ontology, which is already connected to the shared knowledge repository. If knowledge elements are used in the definitions and cannot be found in the shared knowledge repository they will be generalized to existing elements. Figure 4, shows a concrete example of how an importing mechanism import knowledge from Cyc [6] and any OKBC server into Disciple [11], that uses a frame-like knowledge representation for its ontology.

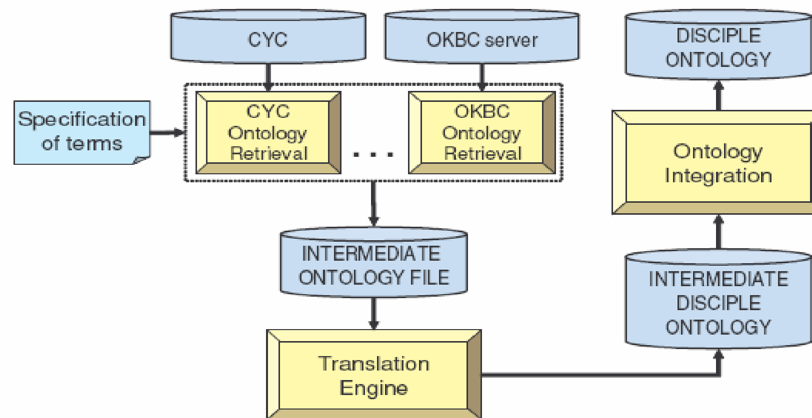


Figure 4. Disciple's Ontology Import Framework

First, the knowledge is sliced into an intermediate ontology format. This step can be performed by various plug-ins that can use different sources of knowledge. There is a need for an intermediate ontology format since the destination frame-like knowledge representation might be quite restrictive and implementing a translation engine into each plug-in would duplicate a lot of work. This way we'll have the same translation engine performing the work on knowledge extracted from any known source. It will deal with constructing a stand-alone ontology which captures as much as possible knowledge from the intermediate ontology file into the knowledge representation of the destination. For example, the Disciple knowledge representation doesn't allow cycles in the ontology hierarchies as well as not allowing an object to be both an instance and a class, things that are allowed in Cyc. In those cases, the translation engine will have to take decisions of how to break the cycles and to define an object as a class or as an instance. The result of the translation step is an ontology that is represented in the same knowledge representation and that can be easier dealt with during the integration process. The integration process itself is usually quite similar with the classic approach for building an ontology: build the class hierarchy, the property hierarchy and in the end populate the ontology with instances and facts. The property hierarchy is built after the class

hierarchy, since it uses classes in defining the property restrictions (e.g. domain and range). Also, facts use both instances and properties, so they are usually the last to be created. This is not a one-time linear process, but it's usually performed in multiple iterations. For example, each time a new fact is required to be added, if the property is not yet in the ontology it will be created. But previous to that, we have to make sure that we have the classes that are used to define the property restrictions.

3. Problem definition

In dynamic environments, underlying ontologies continue to evolve. A new version of a shared ontology of two separate platforms will cause inconsistencies in knowledge and therefore does require some form of update or upgrade in order to work with the most current version of knowledge. This is primarily caused by the introduction of new data classes or instances. For example if the camera manufacturer Nikon were to release new camera models and features, the old shared Nikon camera ontology would now have to be updated with this new knowledge. This phenomenon has brought ontology mapping new challenges in the context of data heterogeneity due to ontology evolution. Manually updating ontology is error-prone. The introduction of multi-agent systems (MAS) has been significant towards ontology management. We believe that the MAS approach is a natural and intuitive way in handling dynamics for ontology evolution.

4. Motivation

When underlying ontologies continue to evolve, the problem of consistency arises between shared ontologies and local ontologies. This is especially true for decision support systems that require knowledge from multiple domains. For example to respond to emergency situations effectively there is a need for rapid ontology development through reuse of knowledge from existing ontologies. It is always easier to build a new system by inheriting knowledge from an existing shared hierarchical repository and this expedites development time. There is a need for support of parallel development, especially in the process of developing systems that require knowledge from multiple domains.

This is addressed by the fact that each agent can extend its own ontology independently, without interrupting the repository knowledge consistency with the help of an ontology versioning system. As the shared ontologies continue to evolve the ontology development has to be consistent with newly made changes to the shared ontology. This is a fairly new problem and is a significant one for ontology mediation researchers as well. There are currently no clear solutions for resolving this problem yet. This also motivates us to provide a reasonable and practical contribution to this domain of knowledge.

5. Literature Review

Once a specific ontology is developed it is never a static entity. New versions arise, updates are performed and general inconsistencies develop as definitions change in the real world. Ontologies continuously evolve especially in a domain with many participants that communicate and contribute to new knowledge in a shared knowledge environment.

This causes inconsistencies and requires new knowledge to be updated in some fashion. We surveyed several works in this area and the following are the ones that are closest to our research goal.

The first approach examines ontology change management by addressing four major change environments such as consistent ontology evolution, repairing inconsistencies, reasoning with inconsistent ontologies and ontology versioning [10]. The authors also address the specifics by emphasizing on syntactic vs. semantic discrepancies, language dependent vs. language independent and functional or non-functional change. Consistent ontology evolution occurs where a developer has a good ontology and maintains consistency as updates occur. The developer must constantly be aware of new versions of ontology or changes in definitions which may cause substantial work on his part.

Repairing inconsistencies deals with an ontology which started out with no inconsistencies and later became inconsistent based on definition changes. The developer in this case is aware of current definitions and resolves inconsistencies with a reasoning system i.e. Processing Inconsistent ONtologies (PION) which is a system implemented in XDIG, an extended description logic interface [10]. Reasoning with inconsistent ontologies is dealt with PION by verifying the relevance of the returned query and extending the consistent sub-ontology for further reasoning. If a query does not return consistent results rather than the developer will try to reason with the inconsistency and get meaningful answers. Finally ontology versioning deals different versions of an ontology and transforms an original version of an ontology into an updated version without loosing prior versions of data that the ontologist wishes to keep. The limitation of this paper was that it did not provide a methodology of how the hierarchical ontology classes were actually matched. We will demonstrate this in our similarity matching methodology.

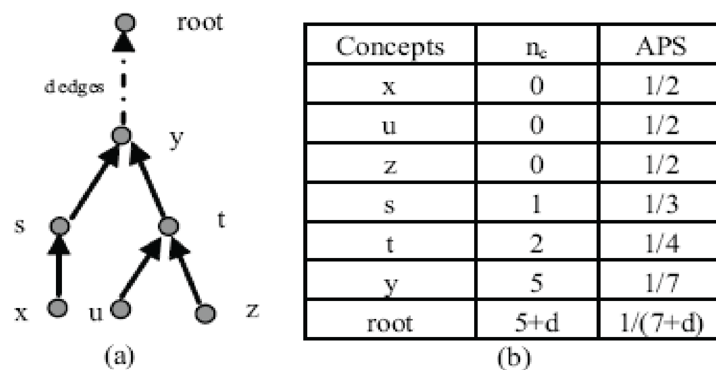
Only until recently have multi-agent systems (MAS) been used in ontology management which direct not only how to use ontology but also how to manage ontology. A multi-agent system (MAS) architecture and an algorithm for multiple agents are proposed for managing and deploying ontologies in a dynamic environment [7]. This was the only paper that proposed ontology evolution and integration using the JADE platform [3]. The MAS approach is believed to be a natural and intuitive way in handling dynamics. A three layered architecture is presented in this paper. The ontology-based application architecture includes three parts ontology-based applications (App1 and App2) for specific ontologies (Onto1 and Onto2), relevant knowledge bases (KB1 and KB2), agent system for the mapping module and the refining module. Agents also offer functionalities such as learning and rule generation to refine ontologies and map between multiple ontologies. However, this paper did not quite demonstrate how agents could be specified in JADE. We will show this in our implementation.

Change and Annotation Ontology (CHAO) was developed by the creators of Protégé to keep track of the changes that happens in the ontology [8]. Changes are represented in the ontology as subclasses of *Change* class and *Annotation* class. There is also a class that is used to group multiple changes into higher-order ones. Changes performed by users are represented as instances of corresponding subclass of *Change* and contain information describing the change and the class, the property or individual to which the change is

applied. The change ontologies are populated either by the ontology tools during the editing episodes or generated by specialized tools that compare two ontologies and extract structural changes (e.g. Prompt) [8, 9]. Change ontologies have multiple uses such as aiding visualization of changes between versions, aiding curators in accepting/rejecting modifications, keeping track of history for individual ontology elements and generating audit information for instance what class was edited and what changes were made.

The instance-based matching technique or “Minimum Similarity” (MS) of hierarchical ontology structures was developed at the University of Leipzip [5]. This approach looks at is-a relations between classes in a hierarchy. The authors propose to derive the similarity between classes from the similarity of the associated instances. Their similarity metrics takes two classes (e.g. C1 and C2) and determines how close they match. Their proof-of-concept was tested on a software and games catalog of Amazon and Softunity. The instance match was determined by the European Article Number (EAN). Even though this is useful in ontology structures with different sized cardinalities, it still lacks the advantage of actually upgrading the ontology. For example, if Amazon published it’s concepts to a shared ontology, Softunity could upgrade it’s version of products to match Amazon. The approach taken in our contribution takes into account that Amazon and Softunity would benefit from sharing the same ontology versions, and thus able to provide better searches to the user.

“Semantic Similarity” (SS) uses actual ontology structures to virtually mesh with other ontologies via three steps [12]. It first, starts by scoring the class from a to b, then evaluates the amount of transfer between each class and finally scores the transfer amount to get a distance value i.e. D (a, b). The initial score reflects the amount of attention or specialty a particular class is given. The goal is to score based on the events that closely correlate to real world selection. Second, a transfer score or “*A-prior*” score (APS) reflects the expected “next move” the average user would make, but without using any user information. By assuming the amount of transfer between certain classes, one can deduce the expected propagation through the ontology structure.



Concepts	n_c	APS
x	0	1/2
u	0	1/2
z	0	1/2
s	1	1/3
t	2	1/4
y	5	1/7
root	5+d	1/(7+d)

Figure 5. (a) a simple β ontology and its AFSs(b)

Finally, an inferring score is developed from the previous steps by showing the lowest common ancestor, or the closest upward reachable node, between the two classes. To test the approach they used the WordNet ontology. They are able to achieve 91% correlation with real user ratings on how close one word-pair matched another. However, this

technique is useful in ontology structures that do not have to be upgraded. Once an ontology is upgraded the user ratings for portions of the new version ontology would be non-existent since there would be no prior knowledge of how much a particular class was chosen. Also the assumption here is that the new version has been published for use and has acquired a selection score. There was also no indication as to how the methodology works and where it had been tested. We provide a better approach that uses WordNet and Latent Semantic Analysis (LSA) that semantically matches classes of a shared ontology with higher reliability and precision.

The third approach deals with Approximate Query Reformation (AQR) where a query represented in one ontology is reformulated approximately into a query represented in another ontology by using ontology mapping specifications [5]. Here, they search separate ontology structures as one entity and based on the union of sets they receive results. The example used within [12] uses the American and Japanese restaurants ontology structures. If American Restaurant ontology has class {AmericanRestaurant} and contains {BeikokuRyouri, wine} which contains {Drink} then {BeikokuRyouri} contains {AmericanRestaurant} and {wineList} contains {drinkMenu} and {winList} contains menu and {wineList} contains {BeikokuRyouri, adultMenu}. From the one ontology structure O1 they are able to get to another ontology structure O2 and receive results [5].

Many mathematical theorems are provided for reformulation, but minimally-contained and maximally-contained stand out. In a minimally-contained reformulation the query minimally covers the original query. This gives a broader result or the least upper bounds of the query. A search stemming from a {wine} class in the {JapaneseRestaurant} ontology, yields {adultMenu} from the {AmericanRestaurant} ontology instead of a more specific Chardonnay. Vise-versa, the maximally-contained reformulation of the query maximally covers the original query and produces more specific results [5]. This approach though flexible does not take into account user input for what the expected results should be if the desired match is not found. Our proposed technique allows the user to decide whether or not to accept certain classes as desirable. This allows the user to choose how his ontology will perform during a query.

6. Proposed Architecture – UPGRADE Use Case

In this section we show the upgrade operation in the case of two working platforms (e.g. Figure 6: Platform 1 and Platform 2). The ontologist who works with Platform 1 maintains a local ontology (i.e. Ritz Camera store ontology) that includes knowledge from a shared ontology (i.e. Nikon Corporation camera ontology), developed on Platform 2. The local ontology is maintained to fulfill a specific goal set by the ontologist. This can be an ontology developed by a store to represent the available photo cameras. The local ontology is not started from scratch but it's using some already existing knowledge from a shared ontology. In our case, Ritz Camera decided to start with the Nikon Corporation camera ontology and to further extend it with other types and models of cameras.

In the context described above, any modification to the shared ontology on the Platform 2 (the Nikon Corporation ontology) will directly impact the local ontology on Platform 1 (the Ritz Camera ontology). Let's say that Nikon released a new type of sensor and a new set of cameras and they released a new version of their camera

ontology. The Ritz Camera will want to use the latest version of the shared ontology and will have to adapt its local ontology to use the new knowledge.

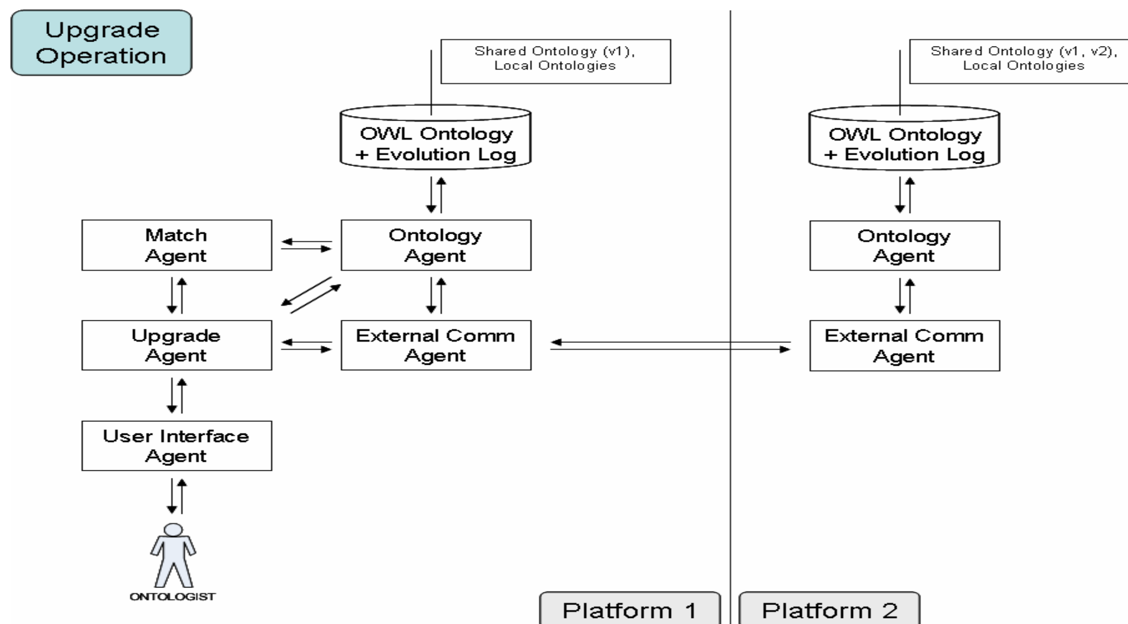


Figure 6. Proposed MAS architecture for the ontology upgrade process

We propose the MAS framework in Figure 6 to take care of the ontology upgrade process (e.g. upgrade a local ontology to work with a new version of a shared ontology).The Ontology Agent (OA) is used by the other agents to get access to the ontology. It uses OWL ontologies behind the scene and stores the ontology modifications in an evolution log. It responds to queries like “what is the root class?” or “what are the subclasses of photo_sensor?” or performs ontology modification like “create the Nikon_18_200_VR_lens instance as a child of zoom_lens”. Please not that the messages are not natural language but rather have a type from a predefined set of message types and parameters.

The External Communications Agent (ECA) is taking care of the inter-platform communication. The other agents inside the platform do not have to know how to reach other platforms nor to be aware of inter-platform communication issues (i.e. Corba/IIOP protocols or firewall limitations) so basically they used ECA as a proxy to the other agents in other platforms. The role of the Match Agent (MA) is to compute the similarity between two given concepts. It is at the core of the semantic mediation process and it’s based on the algorithms described in chapter 8.

6.1 Upgrade Agent

Most of the business logic behind the upgrade process is encapsulated in the Upgrade Agent (UA). It uses the User Interface Agent (UIA) to interact with the user and handles the actual reconfiguration of the existing local ontologies during the upgrade process.The user will initiate the upgrading process through the UIA which will communicate to the UA the user’s intention. The UA will communicate with the Ontology Agent (OA2) from

the Platform 2 (via ECA1 and ECA2) and will receive the differences from the current version of the shared ontology and the latest available version. The differences consist of atomic changes saved by the OA2 during previous ontology modification operations (i.e. “instance Nikon_18_200_VR_lens was created as a child of zoom_lens”). They can be saved in a custom representation in an evolution log or an ontology can be used [8]. The UA will show the user the differences through the UIA and will receive the permission to go on with the upgrade process or to stop it.

During the upgrade process the Upgrade Agent will run an algorithm to process the shared ontology differences. If the differences do not affect elements used in the definitions in the local ontology, they can be performed directly without any modification of the local knowledge. Please note that the differences need to be applied without any tweaking since the old version of the shared ontology need to be brought exactly to the new version. The modifications, if any, are done to the local ontology. We believe that if modifications need to be performed, the semantic mediation can play an important role in finding the best way of reorganizing the local knowledge.

The goal of this paper is not to perform an exhaustive analysis of the types of differences and how they can impact local knowledge but rather to give a concrete example where the semantic mediation can help. Let’s consider a class from the shared ontology that was used in the local ontology as the super-class for some locally defined classes. If the shared class is deleted, something needs to be done with the locally defined children classes. One possibility is to redefine the deleted concept locally and to keep the hierarchy intact. A second possibility is to use semantic mediation and to find a new parent class for the local orphaned classes. In this case algorithms described in chapter 8 will be used to find classes that are the most similar with the deleted class and the user will be asked to choose one of them as the new parent class. The benefit of using semantic matching is in guiding the user by offering only choices that make the most sense and filtering the possible huge numbers of candidates.

7. Development Tools

JADE (Java Agent DEvelopment Framework) is the agent framework that we have used to develop our agent-based application. JADE creates an environment for agents to communicate and is implemented in Java [10]. Since it is compliant with the Foundation for Intelligent Physical Agents (FIPA) specifications, it can facilitate multiplatform agent development. As such JADE was chosen as the development environment for this project. Agents produced in Java using the JADE environment have the distinct advantage of almost always being run-able with minimal supporting download.

Jena [4] is a Java framework for building Semantic Web applications. It provides an API for working with RDF, RDFS, OWL ontologies and SPARQL queries and also includes a rule-based inference engine. Jena was initially developed by HP and released as open source later on.

8. Matching Algorithm

Our matching algorithm runs matches and presents the results to the domain expert for final consideration. The domain expert’s input is only required towards the final stage,

this improves efficiency. The alternative method would be to manually configure matches which could be voluminous for the human expert, especially in complex environments where large set of match candidates are found. The idea here is to reduce the workload of domain experts by eliminating extraneous data and this is how it works. Parameters are entered in each execution of the algorithm and the acceptance threshold is set for (Semantic Relatedness Scores) SRS. Only classes that have SRS scores higher than the specified threshold are presented to the domain expert for scrutiny.

We propose a semantic matching process where classes are matched using highly reliable algorithms based on Lin, Gloss Vector, WordNet Vector and LSA (Latent Semantic Analysis) measures. The similarity function (s) has five components i.e. equivalence (E), inclusiveness (IC), consistency (CN), syntactic similarity (SYN) and semantic similarity (SEM). The five elements within the parenthesis are independent variables that determine the dependant variable (s). Thus producing the following equation:

$$(s)\, f_x = \{\ \mathbf{E, IC, CN, SYN,SEM}\ \}\ \mathbf{(1)}$$

Multiple factors are considered for determining similarity including variables that have nothing in common in order to refine our results. The similarity function negates all disjoint (D) attributes between classes and the modified function is as follows:

$$(s)\, f_x = \{\ \mathbf{E, IC, CN, D, SYN,SEM}\ \}\ \mathbf{(2)}$$

We provide a matching algorithm based on similarity of classes that uses both syntactic and semantic matching in order to determine more reliable and precise similarity scores unlike other methods discussed in the literature survey section earlier. Those with lower scores are recorded into logs and the domain expert reviews them when it calls for his judgment especially in cases where the score is very close to the threshold.

This is because the algorithm is set to discard input classes which do not attain the threshold level. In such cases the class is not recommended to the domain expert. If one or more of the ontology candidates have SRS scores higher than the acceptance threshold, the one with the highest value is chosen as equivalent (a synonym) to the input. Empirical evidence is provided to support this model which will be discussed in the following sections. The following are the steps involved for the matching algorithm:

- Step 1 – Read loaded SO and TO taxonomies: Semantic engine reads taxonomies of the SO and TO. Prepare for detailed matching tests of data labels, go to step 2.

- Step 2 – Equivalence Test: Test for the **equivalence** of source and target classes: Test 1) do they have semantically equivalent data labels, Test 2) are they synonyms or Test 3) do they have the same slots or attribute names. If equivalent, proceed to step 3, 4 and 5. Else go to step 1.

- Step 3 – Inclusive Test: Source and target classes or concepts (C) are **inclusive** if, the attribute (c) of one is inclusive in the other. In other words *selling price* (c_i) is inclusive in *price* (c_j), this is applicable to *hyponyms*. If inclusive, proceed to step 6.

- Step 4 – Disjoint Test: Source and target classes or concepts (C) are **disjoint** if, the intersection of their two attribute sets (c), c_i and c_j results in an empty set { } or ø. If match test is not disjoint, proceed to step 6.

- Step 5 – Consistency Test: Source and target classes or concepts (C) are **consistent** if, all the attributes or slots (i.e. c_1 and c_2) in the class, have nothing in common s.t. $c_1 \cap c_2 = \{\}$. All slots must belong to class that is being tested. This can be configured with RacerPro. If consistent, proceed to step 6.

- Step 6 – Syntactic Match: Syntactic match similarity scores based on class prefix, suffix, substring matches are calculated. This calculation is performed for every class in the source and target ontology. Go to step 7.

- Step 7 – Semantic Match: Semantic match similarity scores based on cognitive measures such as LSA, Lin, Gloss Vector and WordNet Vector are used. This calculation is done for every class in the source and target ontology. Go to step 8.

- Step 8 – Aggregate both similarity scores: Similarity inputs from step 6 and 7 are aggregated, to produce SRS. Go to step 9.

- Step 9 –Populate similarity matrix: The aggregated values (SRS) from step 8 of candidate labels are populated into the similarity matrix. Multiple matches are carried out. Values are to be verified against the threshold. Go to step 10.

- Step 10 – Set threshold: Threshold value (t) is set based on scale used. For a scale between, 0 and 1 the threshold value is usually 0.5 ($t > 0.5$). Those below threshold are logged in file in step 12. If greater than the threshold value, go to step 11.

- Step 11 – Domain Expert Selection: At this stage, candidates from step 10 are presented to domain expert by the system. Input from step 12 is accepted at the discretion of the domain expert.

- Step 12 – Manual Log: Selection is made manually only for those values below threshold. The domain expert uses his own cognitive judgment. Go to step 13.

- Step 13 – Mapping/Alignment/ Merge: All the candidates for mapping, alignment or merge (i.e. integration) chosen from step 11 and 12 are processed. End.

The matching algorithm (see Appendix 1) shows detailed steps before the semantic matching engine produces mappings. The process begins when two ontologies are first loaded (i.e. O_1 and O_2) and they are identified as source ontology (SO) and target ontology (TO).The taxonomies are read and translated for beginning matching. An equivalent test (E) is carried out for data labels to test their similarity in terms of three parameters, 1) test for semantically equivalent data labels, 2) test for synonyms and 3)

test for similar slots or attribute names. *C* is used to refer to classes and *c* refers to attributes or slots.

8.1 Syntactic Matching

Syntactic matching is by and large performed based on approximate prefix, suffix, string and substring matching today. Current ontology mediation methods use syntactic mediation for mapping data labels. They are heavily reliant on grammatical rules and do not support semantic similarity. Syntactic integration defines rules in terms of class and attributes names and does not take into account the structure of the ontology. Syntactic mapping also does not entail coordination of meanings or agreed definitions.

As such, this kind of mapping could be conceptually blind although comparatively easier to implement. However we still believe that it is useful because syntactically similar classes have a higher chance of being semantically similar. As such syntactic matching is recommended to be the first layer for class matching which would eliminate erroneous classes that are not likely to be similar. It can be used to filter through thousands of classes in a given ontology and then in the second layer of matching we use semantics to select classes that are also semantically similar.

8.2 Semantic Matching

Semantics represents of meaning and relations between concept synsets. A class usually has multiple meanings or word senses. Semantics relate to the study of meaning and changes of meanings, which was first defined by Martineau (1887) as a branch of philology. Changes of meanings and analysis are referred to as semantic analysis. Semantic mapping is an approach to map classes of disparate ontologies on the basis of semantic similarity or relatedness that exists between those classes. Before classes are mediated a check for how much similarity exists between those classes must be first determined.

With the aid of cognitive agents and the English language lexical database (i.e.WordNet), concept similarity can be measured. Ignoring word senses in ontology mediation efforts will not give exact mappings. Semantics rely heavily on dictionaries to determine synonyms, evaluate common substrings and consider classes whose documentation share many uncommon words [6]. By combining semantic and syntactic mapping ontologies can be more accurately mediated.

Semantic matching considers all shades of meanings for a concept such as synonymy, meronymy, antonymy, functions, associations and polysemy, which is common in natural language (NL) communication. For example when the same word has more than one meaning (different *senses*) it causes polysemy and when it has opposite meanings it causes antonymy. Synonymy corresponds to the situation when two different words have the same meaning. Since classes are represented in natural language by words, their shades of meanings must be analyzed for similarity before mediation is done. Table 2 shows the four phases involved in our semantic mediation effort. Our mapping framework uses cognitive agents such as WordNet to determine similar classes.

Table 2. Matching Phases

Phase	Task
Semantic agreement	▪ Resolve data heterogeneity –syntax, structure and semantics. Check consistency and integrity. ▪ Identify source and target ontology ▪ WordNet agent eliminates ambiguity
Semantic affinity measurement	▪ Lexical similarity performed using WordNET agent ▪ Synonyms and hyponyms analyzed ▪ Algorithm used to measure similarity
Semantic bridge	▪ Source and target ontology is bridged based on similarity ▪ Data converted to OWL/RDF ▪ Semi-automatic and dynamic mappings performed ▪ 1:1,1:n and n:1 and m:n mappings performed ▪ Complete bridging
Semantic consistency and integrity checking	▪ Ensure results are consistent ▪ Check data integrity ▪ Run again if there are errors present otherwise end

8.3 SRS –Semantic Relatedness Scores

We introduce a hybrid approach that combines both syntactic as well as semantic correspondence to produce SRS. The advantage of our methodology is that we actually adopt cognitive measures for our similarity match which the other researchers have not attempted to do. We derive cognitive measures from combined scores of the WordNet lexical database and LSA. Although several methods exist for measuring similarity of concepts, our tests have shown a high degree of relevance, precision and reliability with the combination of Lin, Gloss Vector, WordNet and LSA measures. Empirical tests based on a study done in Princeton by Miller and Charles shows a high degree of correlation i.e. 92% between SRS scores and human evaluation of 60 concept words. Another advantage of this method is that it also extendable to mapping of multiple ontologies (1:n, n:1 and m:n).

We obtained the Lin and Gloss Vector measures by subscribing to the service supported by Ted Pedersen and Jason Michelizzi from University of Minnesota [1]. WordNet and LSA [2] measures were obtained through the service provided by the Rensselear MSR server. We then aggregate the scores for the four measures to derive the SRS. These services utilize a training phase and a similarity determination phase. In the training phase, the facility first identifies, for a number of pairs of synonyms, the most salient semantic relation paths between each pair of synonyms. The facility then extracts from these semantic relation paths and their path patterns which comprises a series of directional relation types. The number of times that each path pattern occurs between pairs of synonyms is called the frequency.

[1] Ted Pedersen and Jason Michelizzi - http://marimba.d.umn.edu/cgi-bin/similarity.cgi
[2] Rensselaer MSR Server - http://cwl-projects.cogsci.rpi.edu/msr/

9. Conclusion

This paper highlights how changes in an evolving shared hierarchical knowledge repository can be managed in the context of ontology upgrade in a multi-agent system (MAS). A framework has been proposed for multiple agents to deploy changes in a shared ontology environment such as update, deletion and renaming of classes in a dynamic environment. Knowledge sharing and evolution in our multi agent environment adopts semantic mediation techniques for matching classes to overcome inconsistencies. The advantage of our technique compared to others mentioned in literature is that it is based on a hybrid model that combines LSA, WordNet, Gloss Vector and the Lin measure. In essence we combine both semantic and syntactic methods to overcome limitations of previous work. Empirically tests carry out in an earlier study shows a 92% correlation coefficient match which justifies our approach. Our proposed mediation process is a mixed-initiative effort that involves intelligent agents and ontologists. Another important aspect of our work is that it is based on a multi-agent environment and has been implemented in JADE.

References

[1] Barbulescu M., Balan G., Boicu M., Tecuci G., "*Rapid Development of Large Knowledge Bases"* in Proceedings of the 2003 IEEE International Conference on Systems, Man & Cybernetics, Volume: 3, pp. 2169 - 2174, Washington, DC, October 5-8, 2003.

[2] Chaudhri, V. K., Stickel, M. E., Thomere, J. F. and Waldinger, R. J., "*Using Prior Knowledge: Problems and Solutions*", Proc. of the 17th National Conference on Artificial Intelligence and the 12th Conference on Innovative Applications of Artificial Intelligence, 436-442. Austin, Texas: AAAI Press/The MIT Press, 2000.

[3] http://jade.tilab.com

[4] http://jena.sourceforge.net

[5] Jun-ichi Akahani, Kaoru Hiramatsu, and Tetsuji Satoh "Approximate Query Reformation based on Hierarchical Ontology Mapping", pp 1-4.

[6] Lenat, D. B., *"CYC: A Large-scale Investment in Knowledge Infrastructure"*, Communications of the ACM, 38(11): 33-38, 1995.

[7] Li Li, L.W., B.; Yang, Y. *Semantic Mapping via Multi-Agent Systems* in *International Conference on e-Technology, e-Commerce and e-Service*, IEEE, 29th March-1, April 2005: p. 54-57.

[8] Natalya Fridman Noy, A.C., William Liu, Mark A. Musen. *A Framework for Ontology Evolution in Collaborative Environments*. in *International Semantic Web Conference,* 2006: p. 544 -558.

[9] Noy N. and Musen M., "*Anchor-PROMPT: Using Non-Local Context for Semantic Matching*", in Proceedings of IJCAI-2001 Workshop on Ontologies and Information Sharing, Seattle, Washington, August, 2001.

[10] Peter Haase, F.H., Zhisheng Huang, Heiner Stuckernschmidt and York Sure, *A Framework for Handling Inconsistency in Changing Ontologies* Springer-Verlang Berlin Heidelberg, 2005: p. 353-367.

[11] Tecuci G., Disciple: A Theory, Methodology and System for Learning Expert Knowledge, Thèse de Docteur en Science, University of Paris-South, France, 1988
[12] Vincent Schickel-Zuber and Boi Falting "OSS: A Semantic Function based on Hierarchical Ontologies", pp 551-556.

Appendix 1 : Matching Algorithm