UNIVERSITY OF SOUTHAMPTON
Faculty of Engineering and Applied Science
School of Electronics and Computer Science
Intelligence, Agents and Multimedia Group

A mini-thesis submitted for transfer from MPhil to PhD

Principal Supervisor: Prof. Nigel Shadbolt
Secondary Supervisor: Dr. Harith Alani
Examiner: Dr. Kieron M O'Hara

Enabling Active Ontology Change Management within Semantic Web-based Applications

by Yaozhong David Liang

October 2, 2006

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND APPLIED SCIENCE
SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

A mini-thesis submitted for transfer from MPhil to PhD

by Yaozhong David Liang

Enabling traceable ontology changes is becoming a critical issue for ontology-based applications. Updating an ontology that is in use may result in inconsistencies between the ontology and the knowledge base, dependent ontologies and applications/services. Current research concentrates on the creation of ontologies and how to manage ontology changes in terms of mapping ontology versions and keeping consistent with the instances. Very little work investigated on-the-fly keeping track of ontology changes while update (active ontology versioning) and using these information to control the impact on dependent applications/services, which is the aim of our research presented in this thesis. The approach we propose is to make use of ontology change logs as a check-point to analyse changed entities related to the requested services via end-user's incoming queries (RDQL/SPARQL) and amend them as necessary to maintain the validation and continuousness of the dependent application. Firstly, We build up Log Ontology I as the concept structure to organize and construct the change information, develop our prototype system to demonstrate how the change information retrieved from Log Ontology I could be used to control the impacts brought by the ontology changes on the dependent applications and services. And then, by analysing the limitations and difficulties of our prototype system in maintaining the services related to the more complex ontology changes, we identify that the problem which fails the system facing the more complex ontology changes is the inabilities of Log Ontology I to represent complex change information in a semantic fashion. Therefore, we retract to put more focuses on Log Ontology I to enable the implementation of the mechanism to on-the-fly keep track of ontology change information, forming Log Ontology II, in order to reserve the semantics of ontology change from the beginning of ontology update process. Finally we discuss the future direction in terms of how the improved Log Ontology II enables the better service validation and continuousness maintenance of changing-ontology-based applications.

Contents

List of Figures

Chapter 1

Introduction

The Semantic Web is a Web of actionable information supported by the semantic theory which provides an account of meaning to establish the interoperability between the systems. The main vision of the Semantic Web is to provide end-users with more intelligent services based on machine understandable knowledge represented using ontologies and knowledge bases [3]. Ontologies are attempts to more carefully define part of the data world which is used to derive the actionable information, and to allow interactions between data held in different formats. Since Tim Berners-Lee proposed this idea at the First World Wide Web Conference in 1994, there have been a large amounts of efforts directed by World Wide Web Consortium (W3C). Most of them are clustered at specifying, developing, and deploying languages for sharing meaning. From Resource Description Framework (RDF)as the fundamental data format underlying the Semantic Web to the powerful Web Ontology Language (OWL) which provides greater expressivity in the object and relation description, from triple stores as the repositories storing RDF contents to SPARQL language providing reliable and standardised data access into the RDF they hold, ontologies as the main operatable artifacts face the real challenges – they must be well managed to furnish the semantics of the Semantic Web [16].

The number of ontologies that are being developed and used by various applications is continuously increasing. One of the major problems with ontologies is change. Ontologies may change for a variety of reasons, such as when the domain itself or our understanding of it changes, when applying modelling corrections, or expanding the domain representation. Lack of the efficient ontology change management in an engineering fashion would result in the service failure of ontology-deployed applications. Ontology changes may cause serious problems to its data instantiations (the knowledge base), the applications and services that might be dependent on the ontology, as well as any ontologies that import that changed ontology [9]. Therefore, this requires the invention of the methods to handle with the ontology changes to minimise their negative manners on the deployed applications from the beginning of their life-cycle.

There has been much work within the last few years on managing ontology change, so that such updates can be logged and used to provide better maintenance and accessability. Most work so far has focused on ways to handle ontology change, such as change characterisation [9], ontology evolution [11], ontology versioning [5], and consistency maintenance [13, 14, 18]. They feature in analysing the characterisations of the ontology changes to maintain the consistent state of the ontology by discovering changes via comparison of version histories of the same ontology. The common characterisation of these works is that their efforts on the management of ontology changes is put on the identification and collection of the ontology changes from the post-update of the ontology. Not much has been done with the respect to the efforts of keeping track of the ontology change information since the beginning the ontology update process. Moreover, most of the existing works consider the issue of ontology change management only from the perspective of the ontology itself, for example the work of keeping the changing ontologies consistent presented in [18], and not from the perspective of the application, for example the issue of how to maintain the services delivered by the changing ontology-deployed applications. The impact that ontology change can have on the dependent applications and services could be very costly. Therefore, it is would be a new orientation to re-consider the issue of ontology change management from the point when the change happens.

The central research question proposal in this thesis is the following:

> ”Which methods are required to manage ontology changes from the starting of ontology life-cycle in the decentralised Semantic Web environment where changing ontologies are efficiently managed to maintain the knowledge exchange within the deployed application?”

To answer this question, we detail it into four sub-questions:

1. What types of information are needed to describe the ontology change while update in order to make the change process traceable?
2. How to better represent the ontology change operations at the appropriate granularity level so as to reserve the semantics of ontology change?
3. How to dynamically keep track of the ontology changes?
4. Which methods are required to maintain the services of the deployed applications while updating the underlying ontologies?

As the supplement, the answers for the following two questions would enhance the importance of the research question:

1. How to make use of the traceable ontology change information to provide a comprehensive overview of the ontology change history to ontology engineers?
2. How to make ontology change information accessible to the community of practice in order to improve the re-development of the ontology.

In this thesis, we try to achieve a better understanding of the complex ontology change under the umbrella of the Semantic Web. We developed our understanding by analysing the context of the ontology changes and categorising them. Based on this, we introduce Log Ontology I which uses the traditional comparison methods to capture the ontology change information. Then Log Ontology I is used to challenge many problems in maintaining the dependent services which are brought by the changing ontologies. In certain situations, we demonstrate that a number of techniques deployed against Log Ontology I could help solving some of the problems. In the meantime, we identify that the missing of some abilities to represent the complex ontology changes in a semantic manner leads to the failure of the services maintenance in much more complicated situations. According to this, Log Ontology I is upgraded to Log Ontology II where more functions have been enabled to semantically represent the complex ontology changes. This makes us believe that it is a necessity for the dependent application to maintain its continuous services by making use of well-reserved semantics of the change information from the beginning of ontology update process.

This thesis is structured as the following:

- Categorisation and analysis of the existing ontology change management methods within ontology versioning process (Chapter 2)
- Development of Log Ontology I and the experimental work to demonstrate the ability to maintain the application by making use of Log Ontology I to store the change information as well as to disclose the inability to handle more complex ontology changes in this situation (Chapter 3)
- Identification of requirements for upgrading to Log Ontology II in terms of better semantic representation of change information compared with previous generation to improve the dependent application's maintenance (Chapter 4)
- Presentation of the future avenues into which this work will be taken (Chapter 5)

Chapter 2

Ontology Change Management

The dynamic nature envisaged for the Semantic Web must have the capability to cope with continuous evolutions of domain models and knowledge repositories. It is therefore important to manage the ontology changes effectively to maintain the relations that specify how the knowledge is related between the different versions of the same ontology to avoid broken communications.

Passive Ontology Versioning VS Active Ontology Versioning

Based on our investigation of the current researches, existing ontology versioning methods can be categorized into two groups–passive ontology versioning and active ontology versioning.

2.1 Passive Ontology Versioning

Passive ontology versioning methods intend to analyse ontology changes based on the comparison of existing versions of the same ontology. The featured characterisation of these methods is using version comparison to detect and characterise the changes that happened across the multiple versions.

Direct comparison is a popular method for identifying changes between different versions of an ontology. OntoView [14] and PromptDiff [13] are two example systems for ontology comparison. The output of PromptDiff is a proprietary Protégé format (*.diff*). Change information could also be saved in text files if the *Journaling* functionality is activated in Protégé. Neither of the formats used in these files provide semantically structured data. Thus, the reuse of the change information would be more difficult. OntoView exports the difference between ontologies as separate mapping ontologies, which is more advanced than PromptDiff. However, the transformation used to capture the specification of the change operations is based on CVS, which uses the textual transformation mechanism. The granularity level of changes detected by the OntoView is based on a set of triples, RDF statements, which is the smallest directly manageable piece of knowledge [17]. The granularity of changes considered by PromptDiff is on the level of classes and properties.

Currently, there is no agreed versioning and evolution methodology for ontologies on the Web [9]. Within this research area, most work focused on tracking and storing information about ontology change during the evolution process [6, 11] using changes identified at editing time or by comparing a pair of ontology versions. Other effort includes introducing evolution strategies to allow the developers to control and customise the ontology evolution process [1, 15].

Ontology changes can bring unexpected consequences to some dependent applications. However, this realm has received little attention so far in terms of how to better make use of ontology change information to maintain the continuous services delivered by the dependent application. It is identified that the impact of a change in the ontology on the function of the system is hard to predict and strongly depends on the application that uses the ontology [18]. Part of the problem is that ontologies are often not only used as a fixed structure but also as the basis for deductive reasoning. The functionality of the system often depends on the result of this deduction process and unwanted behaviour can occur as a result of change in the ontology. Ontology change information detected by version comparison seems not enough to assist ontology engineers to predict the possible consequences which underlying changing ontology brings to the application and maintain the continuous services. We require the efficient methods to reserve the semantics of ontology changes from the beginning when ontology changes are happening. One of the promising solutions would be a way to enable the better representation structure for the ontology change semantics reservation.

2.2 Active Ontology Versioning

Active Ontology Versioning intends to keep every record of the changes happened on the ontologies from the beginning of updating process. Capturing and logging change information on-the-fly would be the favorable methods to enable active ontology versioning. The difference between the active ontology versioning and passive ontology versioning is the methods and orientation to the change. Active ontology versioning cope with ontology change when they are ready to happen. However, Passive ontology versioning cope with the histories of the ontology changes.

Few works have been performed in active ontology versioning compared with the matured focuses on the works in passive ontology versioning (analysed above). The most recent work is the Stanford's ontology evolution framework [12], which includes the components of Change-management plug-in[1] and PROMPT plug-in[13]. As enabling the change management plug-in within Protégé while updating ontologies, each change to the ontology is recorded as an instance of the change ontology with the timestamp and author of the change. The change information would be saved when the updating process is finished. Thus, with the support of this plug-in, the ontology updating process is monitored and a declarative record of the changes is stored as well. In addition, when ontology developers reviews the ontology changes, the plug-in provides not only an overview of changes and corresponding annotations, but also the associated changes for a specific change. However, the change information are only available for the Protégé project. The desired information could only be retrieved by accessing Protégé knowledge-base API, which limits the deployments of the change information to the other scalable applications under the umbrella of the Semantic Web. Moreover, the lack of version information within the annotation of change records would be the potential problem for coordinating the change information, for example for a specific class, across multiple versions of the same ontology.

[1]The "Changes" tab is a Protege plug-in that allows you to track and annotate changes to your ontologies. URL: http://protege.cim3.net/cgi-bin/wiki.pl?ChangeManagement

Chapter 3

Representation and Manipulation of Ontology Changes in Service Maintenance

In this chapter, we start from the development of Log Ontology I which is used to make-up the ontology change information identified by the version comparison based on our investigation and understanding of the study of ontology changes during the ontology evolution process. With the assistance of the ontology change information classified in Log Ontology I, we develope a prototype system to demonstrate the idea of maintaining the continuous services requested via end-user's RDQL queries by making use of Log Ontology I to store the change information across the multiple versions in the circumstance where application-dependent ontology is changing. In the meantime, a series of experiments are organised to perform on our prototype system to represent how end-user's queries are maintained validated to continuously provide services by means of the change information retrieved from Log Ontology I.

3.1 Log Ontology I

One of the important functions related to managing changes is logging. Logging is necessary to track any modification applied to the ontology. As described in Chapter 2, some research used this method to trace ontologies modifications [11]. However, one of the problems with this system is that it did not represent the information in a semantic format to facilitate its retrieval and understanding by other external applications. For example, it was unable to track the change process from start to end. A semantic representation of change logs may enable other developers and tools to process and understand the evolution history of an ontology [19].

Log Ontology I is produced to semantically mark-up information about changes that can happen between multiple ontology versions. The descriptive information within Log Ontology I includes the new and old status of a specific entity within the underlying ontologies, a set of change operations as well as meta information that relate to each change, such as *author, date, comments,* and *version.*

We chose to base our change representation in Log Ontology I on change operations (shown in Figure 3.1). Change operations precisely define additions, removals or modifications to the definition of a concept, a property, or an ontology as a whole. There are two reasons supporting the use of change operations as the representation structure within Log Ontology I.

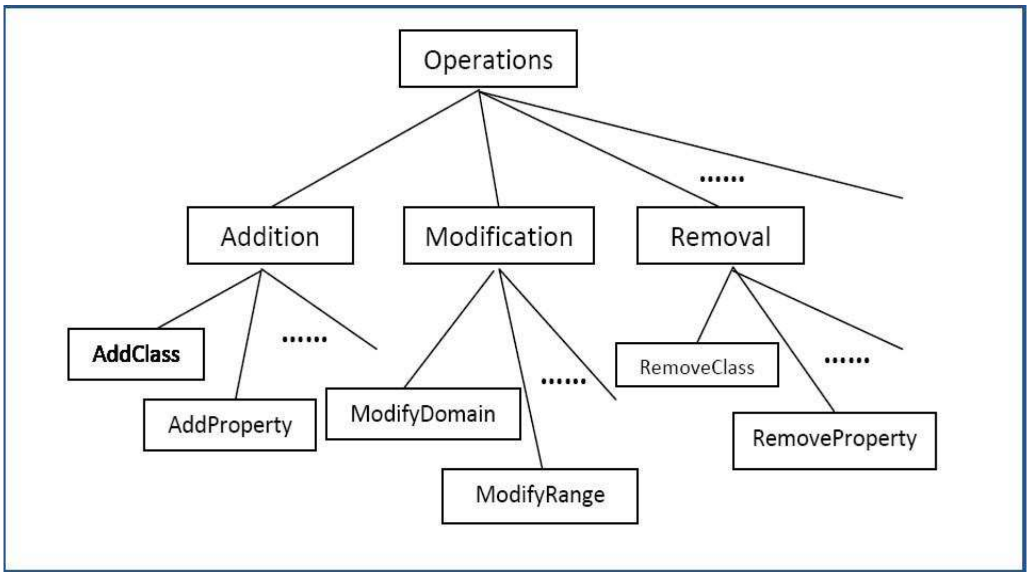


Figure 3.1: The structure of Log Ontology I

- Change operations are useful to specify the operational transformation from one version to another version of the same ontology. However, a set of change operations could not give a complete specification of the change itself. Change operations could also miss some information related to ontology change, such as why the changes happened, the consequences brought by the changes and meta-data. In Log Ontology I, we describe some meta-data such as author, date and comments for the ontology change itself.
- Using change operations could give different level of granularity to the users who are willing to control the representation of the transformations. Most changes made on the ontologies are several modifications in one step. We differentiate these modifications as basic operations and composite operations respectively. By using either basic operations or composite operations, we can specify the granularity level of the change operations.

Log Ontology I is the key element in our prototype system, which is used to keep track of the change information between multiple versions of an ontology. We use Log Ontology I to serve as the basis for the prototype system. In the following sections, the structure and concepts used in Log Ontology I would be explained in detailed via the description of our prototype system.

3.2 System Rational

In Chapter 1, we have stated that it could be costly and perhaps impossible to coordinate every change in an ontology with all dependent applications and services. Our system aims at taking advantage of change-tracks to reduce any impact on the dependent applications that can be brought by changes in the underlying ontologies. An overview of the system is shown in Figure 3.2.

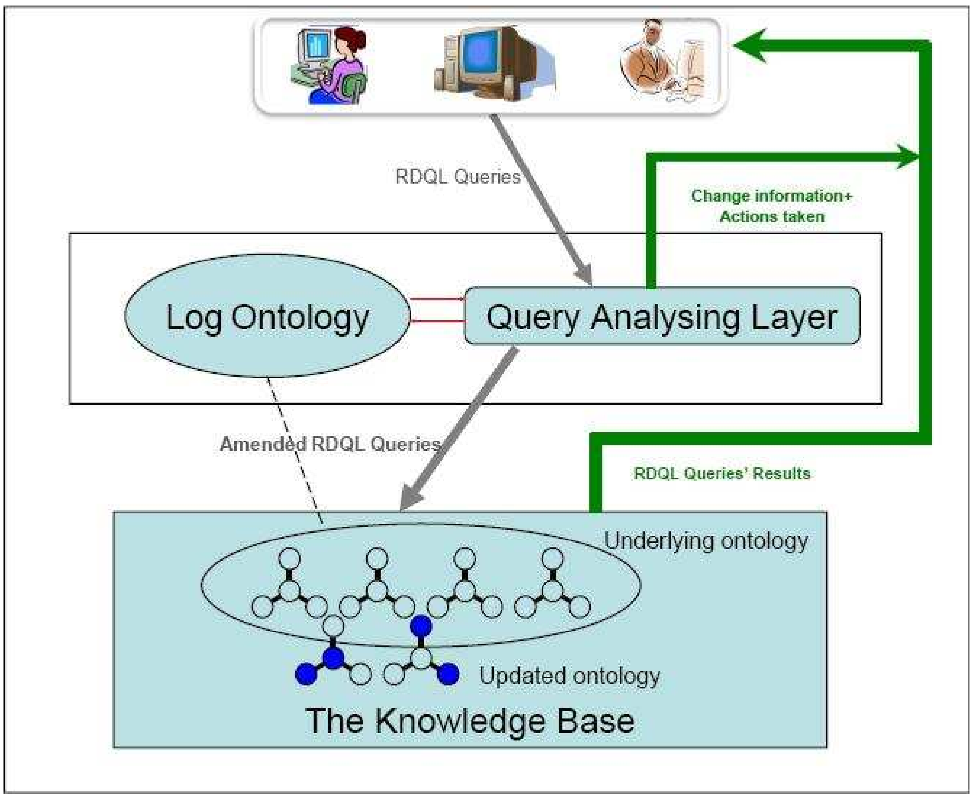


FIGURE 3.2: An overview of the Approach

3.2.1 Problems and Scenarios

As mentioned earlier, ontologies are crucial building blocks for the Semantic Web. However, the data the ontologies encode, and the ontology classification structures can all be subject to change.

From the perspective of end-users of Semantic Web applications, they should expect to have access to robust services that can mediate queries to the changed ontologies without interruption and possible “404 Ontology has changed”errors. The direct and indirect consequences of ontology change could therefore be costly.

If end-user are not aware of any updates to the ontology, then they might not realise how or why their queries started to return different answers, or no answers at all. End-users should anticipate not only obtaining the right knowledge but also to be informed of the updated domain knowledge.

In summary, end-users should expect that applications and services on the Semantic Web can continually deliver the right knowledge at right time in an intelligent fashion. Consistently and efficiently coping with ontology changes will be critical to achieve this requirement.

3.2.2 Suggested Solution

Within our research, we try to achieve a better understanding of a complicated problem. We developed our understanding by analysing the context of our problem and comparing it with the related works in the area. Based on this, our approach was introduced to explore a number of techniques that could be useful to solve some of the problems we identified in the description above.

The solution to tackle the problems identified in our scenarios is described as a series of steps as follows:

1. **Capture**: The changes made between two versions of the same ontology is captured at this stage. Currently, we identify changes by comparing two versions using PromptDiff in Protégé.
2. **Instantiate**: The *Log Ontology I* is populated manually with change information identified in step 1. Our focus is not on automating the population of the Log Ontology I, but on automating the process of analysing the reformatting dated queries to work with the latest changes made to the ontology.
3. **Analyse**: Queries submitted by the applications are analysed to find out whether any of the entities within the queries could be affected by the changes stored in the Log Ontology I.

4. **Update**: If entities within the queries are found to have been changed, they are replaced with their changes to form the new queries with updated entities, and then resubmitted to the queried ontology.

5. **Respond**: After the new-formed queries are submitted to the ontology for processing, the results are returned back to the application. At the same time, a summary of change/update information will also be returned back to the end-users with the query results so as to inform users of the updates.

3.3 The Middle Layer

Normally when the ontologies underlying the applications are updated, not all of the parties utilising the ontologies will be informed of the update. Under such circumstances, the applications may continue to try to query the ontologies unaware of updates or changes to their structure. The Middle Layer aims to serve as a transparent interface between the applications and the ontologies.

The Middle Layer utilises Log Ontology I as the “check-point ”to take the incoming RDQL queries from the end-users, analyses and amends the query as necessary to produce the updated query which is then submitted to the knowledge base to retrieve the required results. Some information will be embedded within the results to be sent back, informing the querier of which of the concepts/properties they have queried have been superceded, what those concepts/properties have been updated to and what kind of actions have been performed on them.

The Middle Layer is implemented as a Java Servlet (see Figure 3.3). The system starts by scanning every URI within the incoming RDQL query. For each URI, the system checks the Log Ontology I which is stored in 3Store[1] by comparing the URIs in the query with their relevant changes in the Log Ontology I. If nothing has happened to a URI, then the program will return it as it was and continue to scan the next URI in the query. However, if a change is logged to the URI, then the program will replace it with the updated URI as recorded in the Log Ontology I. At the same time, the program will extract the trace of change information related to this URI and save it temporarily for later use. When all the URIs in the end-user’s query are processed, the original query will be reformatted with updated URI(s). It is then ready to be submitted to the ontology to retrieve the required results. If there are changes made to the query, then the change information extracted from Log Ontology I previously will be returned to the end-user within the query results.

[1]3Store is an RDF ”triple store”. It provides access to the RDF data via RDQL or SPARQL over HTTP. Available at: http://sourceforge.net/projects/threestore/

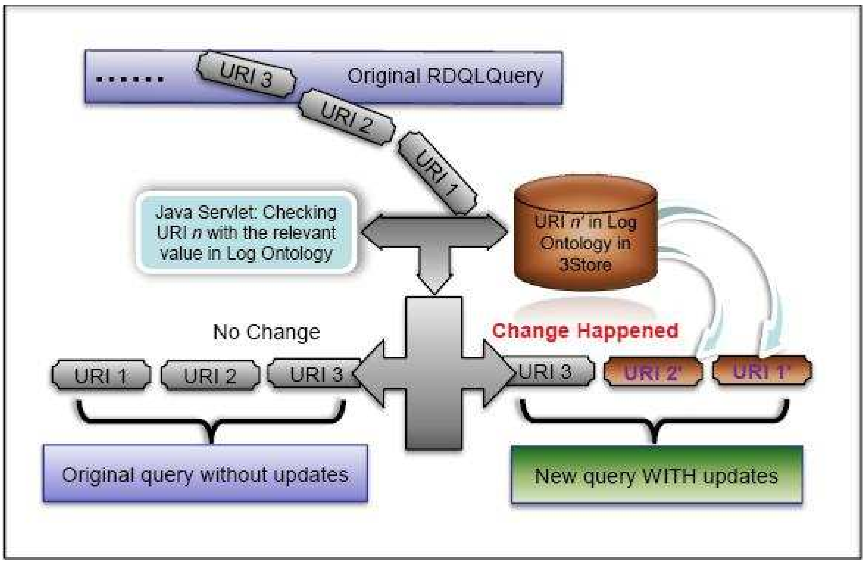

......
URI 3
URI 2
URI 1
Original RDQLQuery
Java Servlet: Checking URI n with the relevant value in Log Ontology
URI n' in Log Ontology in 3Store
No Change
Change Happened
URI 1
URI 2
URI 3
URI 3
URI 2'
URI 1'
Original query without updates
New query WITH updates

Figure 3.3: The working process of the Middle Layer System

3.4 Experiments

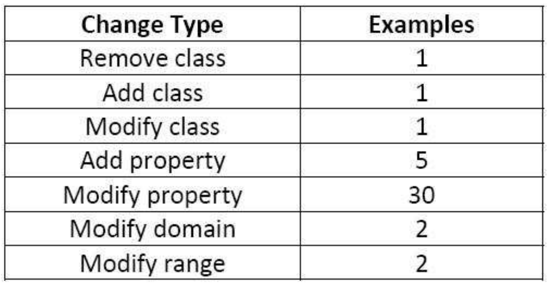
In this section, we will present a number of experiments on a selection of typical RDQL queries to show how our system deals with different type of changes in the CRM ontology.

3.4.1 Example Ontology – CRM

The ontology we used for our experiments is the CIDOC Conceptual Reference Model [2] (CRM). CRM provides a common language and semantic framework for experts and developers in the cultural heritage domain; CRM facilitates the sharing the understanding of cultural heritage information. CRM acts as a semantic glue to mediate between the different sources of cultural heritage information. Our system acts as the gateway for accessing CRM to guarantee that a knowledge base organised by CRM queried by the users' RDQL queries would be accessible when the ontology itself has been updated.

CIDOC provides detailed documentation with each CRM version release. This made this ontology ideal for our experiments due to the ease of change identification. When studying the version history of CRM, one can observe that very few changes were taking place between the versions. We decided to use versions 3.3.2 and 3.4, which have relatively richer changes than the other versions. Figure 3.4 displayed the change types and their numbers as found between these two versions.

[2]The CIDOC Conceptual Reference Model: http://zeus.ics.forth.gr/cidoc/index.html



Change Type	Examples
Remove class	1
Add class	1
Modify class	1
Add property	5
Modify property	30
Modify domain	2
Modify range	2

FIGURE 3.4: The change in CRM ontology

In the following we will demonstrate how the system deal with a sample of queries that relate to different type and combination of changes.

3.4.2 Query 1: Moving a Class

In the first query, we are trying to look for an entity. One of its type is *E23.Information _Carrier*. In addition, we are retrieving who its current keeper is.

RDQL Query

```
SELECT ?entity, ?keeper
WHERE (?entity, <rdf:type>, <CRM:E23.Information_Carrier>),
      (?entity, <CRM:P50F.has_current_keeper>, ?keeper)
USING CRM for <http://cidoc.ics.forth.gr/docs/xml_to_rdfs/
               CIDOC_v3.3.2.rdfs#>
```

Processing Steps

1. Scan user query, and pick up the first URI $< rdf : type >$ to check Log Ontology I for changes. There is no change to this URI
2. Pick up $< CRM : E23.Information\_Carrier >$ to check Log Ontology I whether there are change records related to this entity
3. Change detected. $< CRM : E23.Information\_Carrier >$ was found to have been moved to a new hierarchical place, and renamed as $< CRM : E84.Information\_Carrier >$
4. Replace $< CRM : E23.Information\_Carrier >$ with its new label $< CRM : E84.Information\_Carrier >$ in the user query, and save the relevant change information retrieved from Log Ontology I

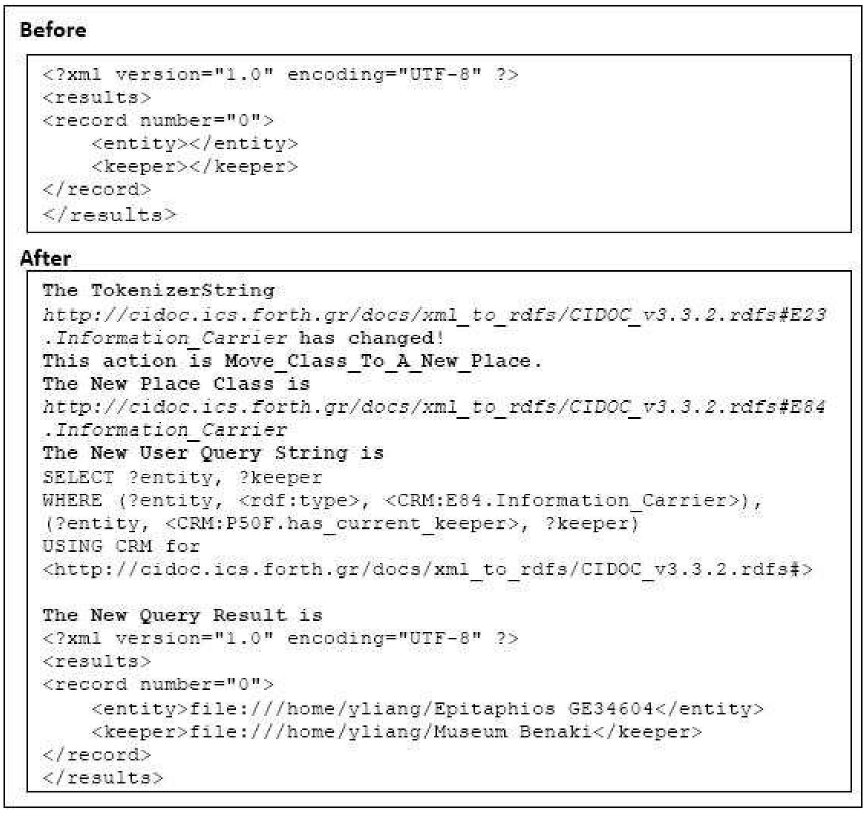
5. Continue to scan the next URI $< CRM : P50F.has\_current\_keeper >$. There is no change to this URI

6. Submit the new formatted query to CRM ontology for execution

7. Embed the change information stored in step 3 with the query result and return them to the user.

Analysis

This change can be treated as a set of two separate change operations: removing a class, then adding a new one. However, in our Log Ontology I we represent a change of the hierarchical position of a class as a "Moving"action. This facilities tracing this change more easily. In version 3.3.2 of CRM ontology, the class *E23.Information_Carrier* is updated to *E84.Information_Carrier* in version 3.4. The actual operation is the change of the position of the class *Information_Carrier* reflected by the digit number given in front of each entity within CRM ontology.



Before

```
<?xml version="1.0" encoding="UTF-8" ?>
<results>
<record number="0">
    <entity></entity>
    <keeper></keeper>
</record>
</results>
```

After

```
The TokenizerString
http://cidoc.ics.forth.gr/docs/xml_to_rdfs/CIDOC_v3.3.2.rdfs#E23
.Information_Carrier has changed!
This action is Move_Class_To_A_New_Place.
The New Place Class is
http://cidoc.ics.forth.gr/docs/xml_to_rdfs/CIDOC_v3.3.2.rdfs#E84
.Information_Carrier
The New User Query String is
SELECT ?entity, ?keeper
WHERE (?entity, <rdf:type>, <CRM:E84.Information_Carrier>),
(?entity, <CRM:P50F.has_current_keeper>, ?keeper)
USING CRM for
<http://cidoc.ics.forth.gr/docs/xml_to_rdfs/CIDOC_v3.3.2.rdfs#>

The New Query Result is
<?xml version="1.0" encoding="UTF-8" ?>
<results>
<record number="0">
    <entity>file:///home/yliang/Epitaphios GE34604</entity>
    <keeper>file:///home/yliang/Museum Benaki</keeper>
</record>
</results>
```

FIGURE 3.5: The comparison of the results between Before and After the updates in query

After scanning the RDQL query, the system detected that the class $E23.Information$ *_Carrier* was subject to changes as stated in the Log Ontology I. The system will fix the query by replacing $E23.Information\_Carrier$ with $E84.Information\_Carrier$ which it has retrieved from the Log Ontology I. Finally, the new query with updated class URI is submitted for execution. All of this information is returned to the querying application along with the query result.

Figure 3.5 shows that before our system fixes the query, the query result would be return empty because the update from *E23.Information_Carrier* to *E84.Information_Carrier* breaks the original query. After query fixing, required results would be retrieved from CRM ontology.

3.4.3 Query 2: Modifying a Property Domain

In the second query, we are trying to find all objects that changed custody (has value for *P30B.custody_changed_by*), and where the change of custody took place (value for *P7F.took_place_at*). This is an example of a nested query.

RDQL Query

```
SELECT ?individual, ?entity, ?place
WHERE (?individual, <CRM: P30B.custody_changed_by>, ?entity),
      (?entity, <CRM:P7F.took_place_at>, ?place)
USING CRM for <http://cidoc.ics.forth.gr/docs/xml_to_rdfs/
               CIDOC_v3.3.2.rdfs#>
```

Processing Steps

1. Scan the user query, and pick up the first URI $< CRM : P30B.custody\_changed\_by >$ to check Log Ontology I for changes

2. Change detected. $< CRM : P30B.custody\_changed\_by >$ was found to be renamed as $< CRM : P30B.custody\_transferred\_through >$.

3. Double-check whether there are more changes logged for property $< CRM : P30B.custody\_transferred\_through >$ considering more constraints will be put on the property

4. Change detected. Domain of $< CRM : P30B.custody\_transferred\_through >$ has been modified from $< CRM : E19.Physical\_Object >$ to $< CRM : E18.Physical\_Stuff >$

5. Replace $< CRM : P30B.custody\_changed\_by >$ with $< CRM : P30B.custody\_transferred\_through >$ in the user query to form a new query, and save the detected change information retrieved from Log Ontology I

6. Continue to scan next URI $< CRM : P7F.took\_place\_at >$. There is no change logged for this URI

7. Submit the new formatted user query to CRM ontology for execution

8. Embed the change information stored in step 5 with the query results and return them to the user.

Analysis
This experiment is designed to test the ability of the system in handling some changes on a property. The change operations involved here are renaming a property as well as modifying domain of the property.

In this case, two operations happen on the property $P30B.custody\_changed\_by$ at the same time. Our system firstly checked in the Log Ontology I that the property name has been changed, and replaced it with the new name label. Secondly, the system identified that the domain of $P30B.custody\_changed\_by$ has also changed, and retrieved and saved the updated domain information to be returned to end-users with query result. Finally, the result, along with the information about relevant changes, are returned back to querying application.

3.4.4 Query 3: The Relationship between the Property and its Sub-Properties

In the query 3, we are trying to retrieve all the information related to the producer. What are his products? What types are they? What kind of note do they have? At the same time, who is responsible for the producer?

RDQL Query

```
SELECT ?entity, ?producer, ?type, ?note, ?org
WHERE (?entity, <CRM:P108B.was_produced_by>, ?producer),
      (?producer, <rdf:type>, ?type),
      (?producer, <CRM:P3F.has_note>, ?note),
      (?producer, <CRM:P14F.carried_out_by>, ?org)
USING CRM for <http://cidoc.ics.forth.gr/docs/xml_to_rdfs/
               CIDOC_v3.3.2.rdfs#>
```

Processing Steps

1. Scan the user query, and pick up the first URI $< CRM : P108B.was\_produced\_by >$ to check Log Ontology I for changes. There is no change to this URI
2. Continue to scan next URI $< rdf : type >$ by checking Log Ontology I for changes. There is no change to this URI
3. Continue to scan next URI $< P3F.has\_note >$
4. Change detected. Properties $< P79F.beginning\_is\_qualified\_by >$ and $< P80F.end\_is\_qualified\_by >$ are now classified as sub-properties of $< P3F.has\_note >$
5. Double-check whether there are further changes to $< P3F.has\_note >$. There is no further change to $< P3F.has\_note >$
6. Save the change information regarding $< P3F.has\_note >$ as retrieved from Log Ontology I
7. Continue to scan next URI $< P14F.carried\_out\_by >$ by looking for relevant changes within Log Ontology I
8. Change detected. $< P14F.carried\_out\_by >$ with the other two properties $< P96F.by\_mother >$ and $< P99F.dissolved >$ were declared as sub-properties of $< P11F.had\_participant >$
9. Double-check whether there are further changes to $< P14F.carried\_out\_by >$. There is no further changes.
10. Save the change information regarding $< P14F.carried\_out\_by >$ as retrieved from Log Ontology I.
11. No changes is required to the user query. Submit it to CRM ontology for execution
12. Embed the change information stored in steps 6 and 10 with the query results and return them to the user.

Analysis

This example query targeted two kinds of situations related to the relationship between the property and its sub-property. P has new sub-properties and property P is a new sub-property to the existing property.

In the case where a property has new sub-properties, two sub-properties have been added to property *P3F.has_note* which appears in the query. The new sub-properties are $P79F.beginning\_is\_qualified\_by$ and $P80F.end\_is\_qualified\_by$. In the case where property P is a new sub-property to an existing property, property $P14F.carried\_out\_by$ was found to be one of the sub-properties of $P11F.had\_participant$. The system also identified that the other properties sibling to $P14F.carried\_out\_by$ were $P96F.by\_mother$ and $P99F.dissolved$.

Our system paid more attention to the changes made on the properties based on the fact that properties are normally used to represent the relationship between two classes which is composed of the complex internal network of the ontology itself. This experiment combined with the previous one addresses a common issue on which we are focusing our research to identify and handle hidden changes appropriately.

Chapter 4

Log Ontology II

4.1 Limitations and Difficulties of the Supports from Log Ontology I

Currently the system scans each URI entity in the user's query, starting from the first URI, to detect changes and fix the query with an updated URI as retrieved from the Log Ontology I. However, where complex changes take place, the current analysis method may cause unpredictable results. This is because the current method takes each entity in isolation and does not prioritise query replacements when an entity has been subject to a number of changes at once. For example, consider the property *hasAge* in Figure 6. The Log Ontology I may log *hasAge* as being subjected to two main changes between *Version 1* and *Version 2*. Firstly, with respect to the class *Person*, *hasAge* has been replaced by *hasBirthYear*. Secondly, property *hasAge* is now being used by the *Vehicle* class. If we were to submit a query:

```
SELECT ?age WHERE (<Person>, <hasAge>, ?age);
```

we would retrieve two changes made to the property *hasAge*. Depending on how we prioritise the change types that will decide whether we change *hasAge* to *hasBirthYear*, or whether *Person* should be changed to *Vehicle* in the query. If we assume the classes are fundamental to the semantics of the query, then the decision of how we handle the entity substitution would be based on the classes in the query. However, if no class URIs are explicitly mentioned in the query, such as:

```
SELECT ?age WHERE (?p, <hasAge>, ?age);
```

then we may still define a priority for query substitution, such that, for example, *hasAge* is replaced by *hasBirthYear* if classes are given higher priority. This would be the case if the query is expressed in terms of the *Version 1* ontology. However, based on that individual query, the system is unable to infer which version of the ontology is being used, because the property *hasAge* exists in both versions. One approach is to reason over the query to determine any particular version of the ontology. References to URIs, or relationships between a property and another entity that are specific to a particular ontology version allow us to infer that the query source is querying that version of the ontology. We may also use previous queries submitted from the querying source for this purpose. If no clues can be inferred, it is pragmatic to assume the query source is querying the latest version.

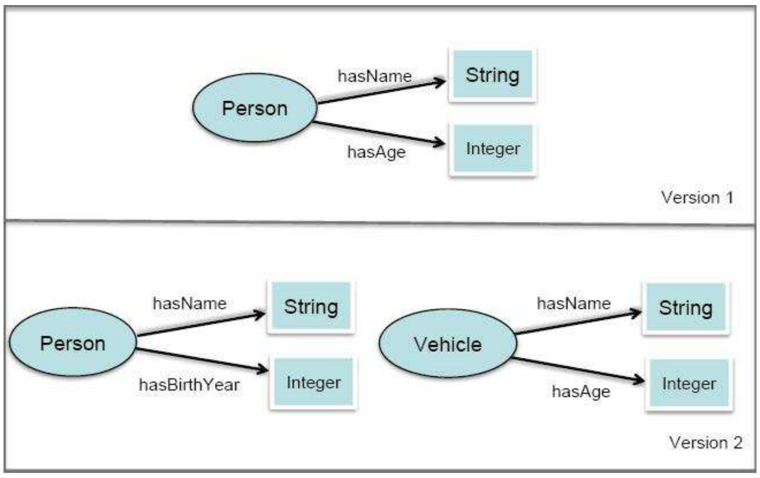


FIGURE 4.1: An example of more complex ontology changes

4.2 Requirements for Log Ontology II

Based on the experiments (*Section 3.4*) on CRM ontology to show how our system deals with different types of changes represented by the concept model of Log Ontology I and the following discussion (*Section 4.1*)to clarify the potential issues on which the improvement could be made to generalise our approach, we identify the limitations of our current approach and learn the new requirements from it which provides us the foundations to upgrade Log Ontology I to Log Ontology II.

- Series of change: currently, the changes we captured are based on two CRM ontology versions only. The representation of change course is therefore relatively simple which is from *Version A* to *Version B*. If we need to represent the same updated entity information in the third version status *Version C* following *Version B*, it would be impossible to build up relationships between these three consecutive change steps. In Log Ontology II, we should enable the ability to represent the consecutive course of change actions. In addition, changes between multiple versions of the same ontology can be iterative. For example, changes can be made from *Version 1* to *Version 2* of *Ontology O*. In *Version 3*, some changes might be changed back to their original form in *Version 2*. We need to represent such series of change actions in Log Ontology II. Our system must handle series of change actions by coordinating the change information retrieved from Log Ontology II. This would allow our system to cope better with more complex ontology changes. It will also assist us to better understand the change evolution process.

- Types of change: we chose the CRM ontology as the underlying ontology for our experiments due to the number of versions available online. However, the number and type of changes that have been applied to this ontology is rather limited. An ontology which has been subject to bigger and more complex changes is needed to widen our experiments. Currently we are investigating the BioSAIL ontology for this purpose, which has been previously used for ontology change studies [8]. By incorporating much more types of ontology changes, Log Ontology II could assist us to better capture and represent the complex changes.

- Correlated changes: The knowledge represented by the ontologies have direct and indirect correlations. Currently Log Ontology I only has the ability to express direct correlations. In Log Ontology II we need to refurbish the ontology change type concept structure by incorporating more descriptive properties for change operation entities to enable the application retrieve the indirect correlations out of Log Ontology II.

4.3 Log Ontology II Construction

The experiments based on Log Ontology I are quite limited in the terms of identified number and type of the ontology change. In order to broaden our understanding in ontology changes, in particular the complex changes, we need much more available ontology version history than CRM ontology to investigate more general ontology changes. With the supports of Stanford Medical Informatics, BioSAIL ontology is chosen for this purpose.

BioSTORM Systems Abstraction and Interface Layer (BioSAIL) is a flexible and extensible ontology. It describes basic conceptual elements of non-clinical data and allows the user to build customised descriptions of specific data sources and formats from these element. BioSAIL is used to facilitate the immediate use of non-clinical data in the absence of a comprehensive standard. We obtained 4 main releases including 16 versions of BioSAIL ontology. Compared with 2 releases including 2 version of CRM ontology used in Log Ontology I, BioSAIL ontology provides us a good source to lay out the investigation about the types of ontology changes (detailed comparison between Log Ontology I and Log Ontology II could be found in the following section).

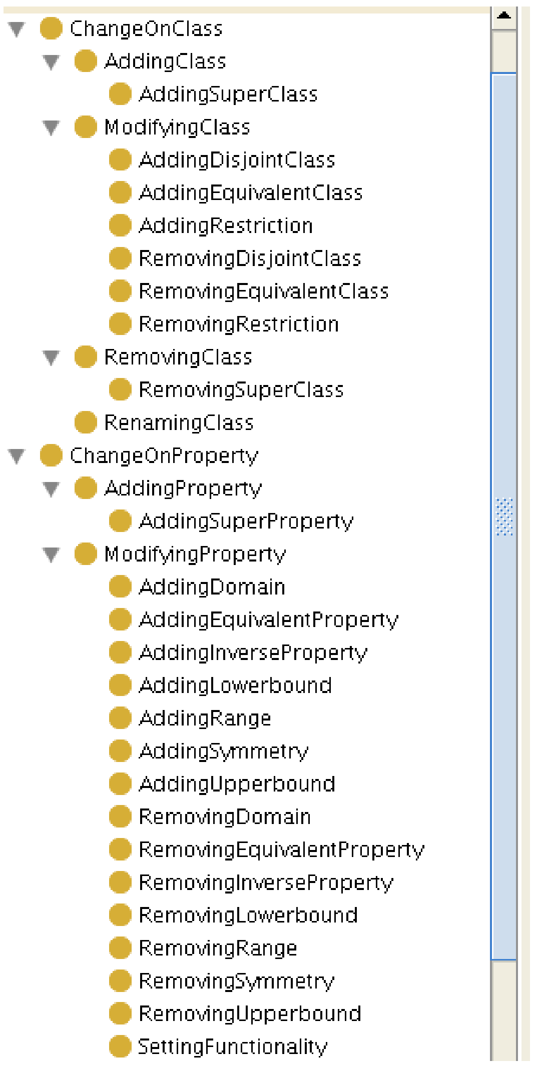
Log Ontology II has been made some significant improvements to enhance the capability to represent the series of operational transformations across multiple versions of BioSAIL ontology. These improvements include:

- The collection of change types has been enlarged with great efforts in identifying the change information from the large numbers of BioSAIL ontology history versions. In Log Ontology I, 9 change types identified from CRM ontology were represented. In Log Ontology II, the number of ontology change operation types has been increased to 33 types. The enlarged collection of change types would not only enable us to represent more complex ontology changes than before, but also help us better understand the complex ontology changes.

- The classification of change operation types is re-organised in terms of change operations performed on the different entities within the ontology. The concept organisation structure of Log Ontology I is displayed in Figure 3.1. The change operations in Log Ontology I are classified into the different change operation type groups (Add, Modification, Removal, ...) based on their shared characterisation of change actions. In the scale of our prototype system using CRM ontology, the structure of Log Ontology I is appropriate and capable of dealing with the change information retrieval for the process of deployed applications. However, as for the applications handling the more complex ontology changes, it is a necessity to have a clear and detailed change operation concept hierarchy to construct the backbone knowledge base for the more powerful change information retrieval. Figure 4.2 shows part of the new concept hierarchy of Log Ontology II.

  In Log Ontology II, the change operations are classified based on their action targets. For example, *AddingDomain* is a change operation performed on Property; *AddingEquivalentClass* is a change operation performed on Class. Following this rule, we divide all the change operation types identified from the ontology version history of BioSAIL ontology into two large groups: *ChangeOnClass* and *ChangeOnProperty.* We still apply the method within Log Ontology I of differentiating change operations as basic operations and composite operations respectively to organise the change operations in Log Ontology II. Therefore, within each group, the change operations are further classified into four sub-groups in terms of action granularity which are *Adding...*, *Modifying...*, *Removing....* and *Renaming....*

  When differentiating the change operations, we keep the operation granularity as basic as possible. This could leave enough flexibility to the users who are willing



- ChangeOnClass
  - AddingClass
    - AddingSuperClass
  - ModifyingClass
    - AddingDisjointClass
    - AddingEquivalentClass
    - AddingRestriction
    - RemovingDisjointClass
    - RemovingEquivalentClass
    - RemovingRestriction
  - RemovingClass
    - RemovingSuperClass
  - RenamingClass
- ChangeOnProperty
  - AddingProperty
    - AddingSuperProperty
  - ModifyingProperty
    - AddingDomain
    - AddingEquivalentProperty
    - AddingInverseProperty
    - AddingLowerbound
    - AddingRange
    - AddingSymmetry
    - AddingUpperbound
    - RemovingDomain
    - RemovingEquivalentProperty
    - RemovingInverseProperty
    - RemovingLowerbound
    - RemovingRange
    - RemovingSymmetry
    - RemovingUpperbound
    - SettingFunctionality

FIGURE 4.2: The concept hierarchy of Log Ontology II

to control the representation of the change transformations. For example, moving a class to a new hierarchy position is de-composed into two separate operations: removing this class firstly and then adding a new class. Although we treat this type of composite change operation in the same manner in Log Ontology I there is no way to present the relationship between two operations. We need the external application, such as Java interpretation program in our previous demonstrated system, to retrieve this relationship out via the shared instance name as the value of the properties which are used to depict the respective change operations. In Log Ontology II, we maintain the operation granularity at the basic level; in the mean-time we introduce the new properties to build up relationship between two basic change operations to present composite change operations without the assistance of the external interpretation program. One hand, this decreases the complexity of the application which deploys the Log Ontology II; on the other hand, this provides us the potential ability to represent the series of changes (details is described in the following bullet point).

- The newly introduced properties used to connect continuous change operations enable the representation of series of ontology changes in Log Ontology II. Log Ontology I and Log Ontology II can both present the continuous changes, however, they used the different methods to implements it. Taking moving a class as an example to illustrate this situation in both Log Ontology I and Log Ontology II. In Log Ontology I populated with CRM ontology, the continuous change operations are connected via the shared instance data. *Information_Carrier* is a case of moving a class. First removing *E23.Information_Carrier. removeClass*'s properties *has_DeletedName* keeping the information of deleted entity (its value is *E23.Information_Carrier*) and *has_NewPlace* keeping the information whether there are new added name afterwards are used to describe the deletion action. If there is a value for property *has_NewPlace*, then there would be a addition action following this deletion; otherwise it is just a deletion. In this case, its value is *E84.Information_Carrier.* Then adding a new class *E84.Information_Carrier. addClass*'s property *has_NewName* (its value is *E84.Information_Carrier*) is to keep the new class name which is same as the value of *has_NewPlace* of *removeClass. E84.Information_Carrier* would be the shared instance data for the external interpretation program to retrieve the information about moving a class. In Log Ontology II populated with BioSAIL ontology, we use property *hasFollowedAdditionOperation* to connect two continuous change operation. *Lab_Results* is an example of moving a class in version 2.0.3 of BioSAIL ontology. *Lab_Results* is moved from being the sub-class of *Specific_Diagnosis* to being the sub-class of *Medical_Data.* First removing *Lab_Results* from the class *Specific_Diagnosis. RemovingSuperClass*'s property *hasDeletedClassName* (its value is *Lab_Results*) combined with property *hasDeletedSuperClassRelationship* (its value is *Specific_Diagnosis*) is used to represent this deletion action. Then adding *Lab_Results* to the class *Medical_Data. AddingSuperClass*'s property *hasAddedClassName* (its value is *Lab_Results*) combined with the property *hasAddedSuperClassRelationship* (its value is *Medical _Data*) is used to represent this addition action. In addition, property *hasFollowedAdditionOperation* of *RemovingSuperClass* connecting this class deletion operation and the later class addition operation is used to represent the process of moving *Lab_Results* from *Specific_Diagnosis* to *Medical_Data* (see Figure 4.3).

Based on the above analysis of the methods used to represent the continuous ontology change operations in Log Ontology I and Log Ontology II respectively, it is obviously observed that applications deploying Log Ontology II would not need specific interpretation program as it does in Log Ontology I to retrieve the information about the continuous ontology changes because of Log Ontology II self-contained ability to represent the continuous change process in a coherent manner (see Figure 4.4).

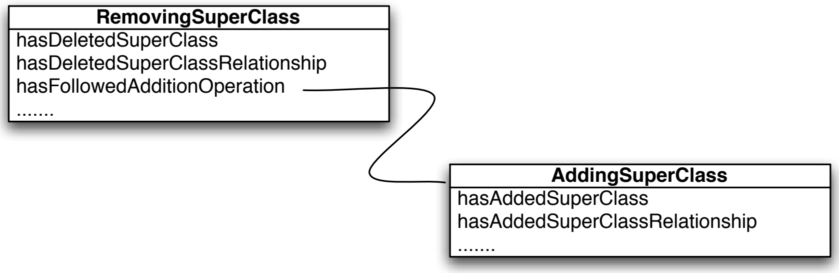
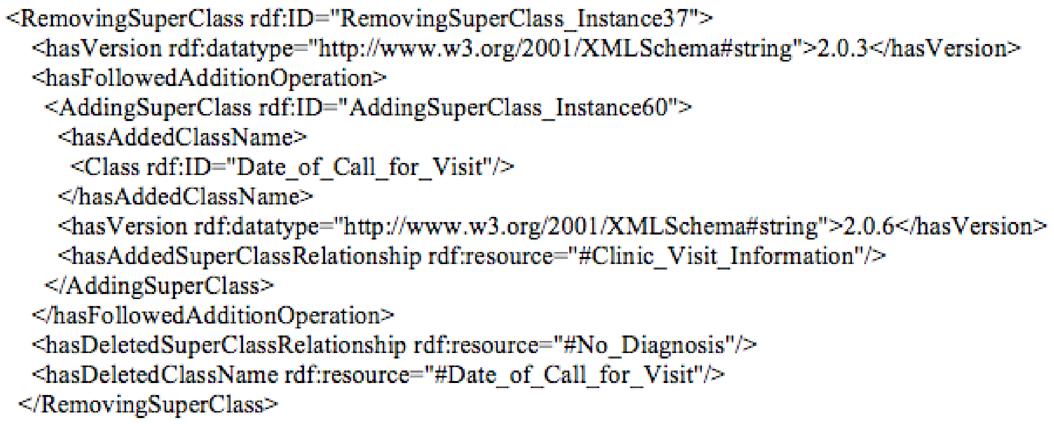

RemovingSuperClass
hasDeletedSuperClass
hasDeletedSuperClassRelationship
hasFollowedAdditionOperation
....
AddingSuperClass
hasAddedSuperClass
hasAddedSuperClassRelationship
....

FIGURE 4.3: The diagram of the representation of continuous change operations in Log Ontology II



```
<RemovingSuperClass rdf:ID="RemovingSuperClass_Instance37">
  <hasVersion rdf:datatype="http://www.w3.org/2001/XMLSchema#string">2.0.3</hasVersion>
  <hasFollowedAdditionOperation>
   <AddingSuperClass rdf:ID="AddingSuperClass_Instance60">
    <hasAddedClassName>
     <Class rdf:ID="Date_of_Call_for_Visit"/>
    </hasAddedClassName>
    <hasVersion rdf:datatype="http://www.w3.org/2001/XMLSchema#string">2.0.6</hasVersion>
    <hasAddedSuperClassRelationship rdf:resource="#Clinic_Visit_Information"/>
   </AddingSuperClass>
  </hasFollowedAdditionOperation>
  <hasDeletedSuperClassRelationship rdf:resource="#No_Diagnosis"/>
  <hasDeletedClassName rdf:resource="#Date_of_Call_for_Visit"/>
 </RemovingSuperClass>
```

FIGURE 4.4: The representation of continuous change operations in OWL

4.4 The Comparison between Log Ontology II and the Related Work

Our Log Ontology was inspired from the works [8]. Both ontologies hierarchy are all built based on the possible change operations during ontology evolution process. The change types identified in Klein's Ontology of Change provide a good foundation for our Log Ontology to investigate the necessary elementary operations to implement traceable ontology changes process. However, it is impractical and impossible to transplant the whole or simply extract part of Ontology of Change for our research work because we need different views on the orientation, concept granularity and application.

1. Our Log Ontology is oriented to be applied to handle with changes information on classes and properties. Our purpose is to keep the concept infrastructure of the application using Log Ontology consistent and coherent. As for the data level, we did not consider the changes happened on the individuals in our work. Klein's

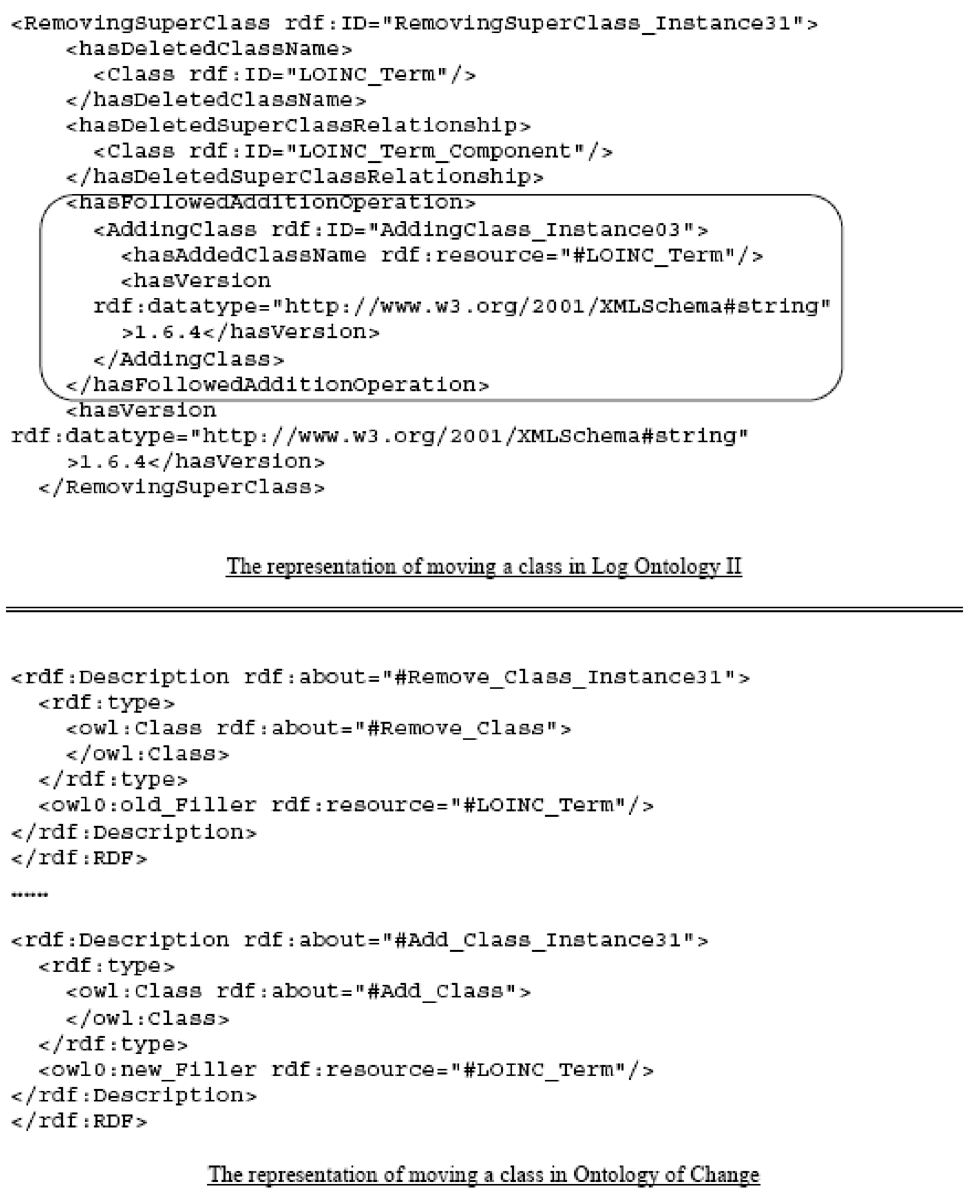
Ontology of Change is a great coverage of changes operations identified so far [2]. It is composed of the change operations for classes, properties and individuals. Its effort is to produce a general taxonomy of changes to specify the consequences the change operations on specific tasks [2].

2. The change operations organised in Log Ontology are all kept elementary because we believe that a composition operation can always be defined as a series of elementary operations at least approximately [10]. In addition, Log Ontology is mainly designed to enable the change process traceable. Keeping the change operations at the atomic level would be beneficial for semantically interpret the composite and complex ontology changes. Klein's Ontology of Change includes not only the elementary change operations but also composite change operations. The composite change operations are used to identify that the effects of composite change operations are more complex than the simple combination of the effects from the related elementary change operations, and also are used as a basis to resolve the inconsistency during ontology evolution process [7].

3. The perspective application also makes our Log Ontology different with Klein's Ontology of Change. Log Ontology is mainly used as a concept structure to organise the change information among the various versions of the same ontology in order to make the change process traceable. This requires that hierarchy of Log Ontology be organised in the atomic granularity to interpret as many complex changes as possible and the change operation types identified in a specific domain be possibly complete to encode as much change information as possible. Klein's Ontology of Change is designed as a complete operation list to describe the possible changes happened in ontologies. It is expected to be a general taxonomy of changes to investigate the effects of the changes brought to the ontologies so as to resolve the inconsistency during ontology evolution. This requires the inclusion of possible ontology change types in Ontology of Change be as complete as possible.

Let's take a look at the detailed comparison illustrated by populating the same instance data into our Log Ontology II and Klein's Ontology of Change respectively.

- Example of moving a class. In Figure 4.5, we present an example of moving a class operation. *LOINC_Term* was a sub-class of *LOINC_Term_Component* in version 1.6.3 of BioSAIL ontology. In version 1.6.4, it became a class in the same level as *LOINC_Term_Component*. In Log Ontology II, we treat it as a composite operation composed of removing a class and then adding the same class to a new hierarchical position as it is shown in the upper part of Figure 4.5. Property *hasFollowedAdditionOperation* is used to connect two atomic operations (removing a class and adding a class) to present this continuous change process. Similarly in Ontology of Change, we decompose moving a class into removing a class first and then adding



```
<RemovingSuperClass rdf:ID="RemovingSuperClass_Instance31">
    <hasDeletedClassName>
      <Class rdf:ID="LOINC_Term"/>
    </hasDeletedClassName>
    <hasDeletedSuperClassRelationship>
      <Class rdf:ID="LOINC_Term_Component"/>
    </hasDeletedSuperClassRelationship>
    <hasFollowedAdditionOperation>
      <AddingClass rdf:ID="AddingClass_Instance03">
        <hasAddedClassName rdf:resource="#LOINC_Term"/>
        <hasVersion
      rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
        >1.6.4</hasVersion>
      </AddingClass>
    </hasFollowedAdditionOperation>
    <hasVersion
rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >1.6.4</hasVersion>
  </RemovingSuperClass>
```

The representation of moving a class in Log Ontology II

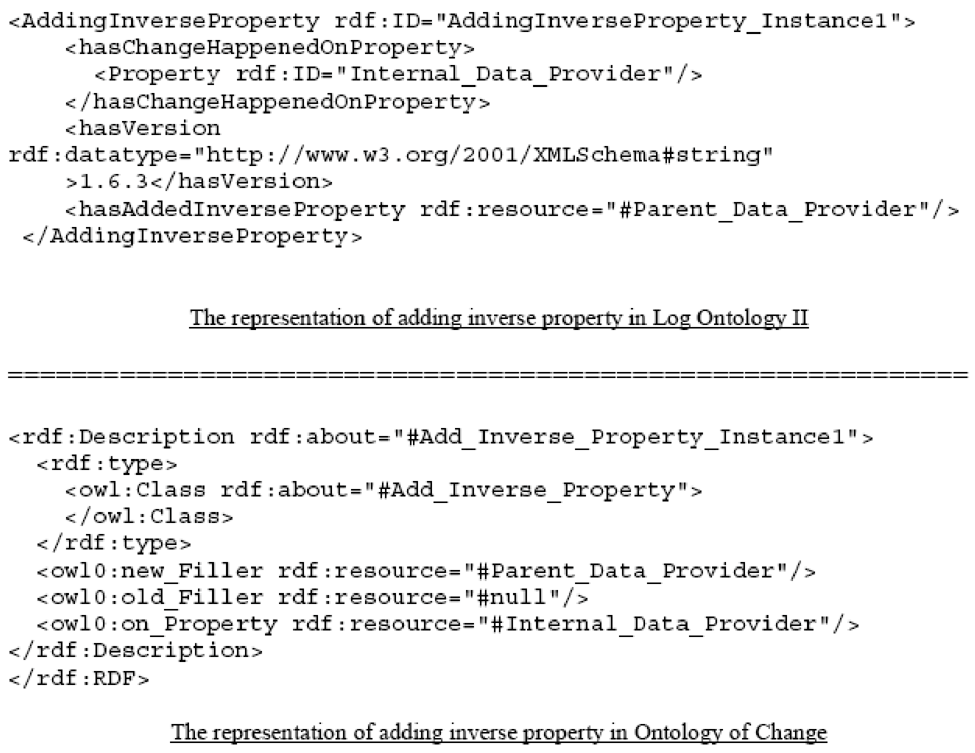
```
<rdf:Description rdf:about="#Remove_Class_Instance31">
  <rdf:type>
    <owl:Class rdf:about="#Remove_Class">
    </owl:Class>
  </rdf:type>
  <owl0:old_Filler rdf:resource="#LOINC_Term"/>
</rdf:Description>
</rdf:RDF>
......

<rdf:Description rdf:about="#Add_Class_Instance31">
  <rdf:type>
    <owl:Class rdf:about="#Add_Class">
    </owl:Class>
  </rdf:type>
  <owl0:new_Filler rdf:resource="#LOINC_Term"/>
</rdf:Description>
</rdf:RDF>
```

The representation of moving a class in Ontology of Change

FIGURE 4.5: The comparison of the OWL representation of moving a class operation between Log Ontology II and Ontology of Change

it again. However, something missing makes two operation steps unlinked (shown in the under part of Figure 4.5). We could not identify the function of composing two atomic change operations into a composite operation to present a continuous change process in Ontology of Change. To keep ontology changes traceable, the ability to present continuous change operation is a crucial functional part during the implementation because it could provided recorded information to keep track of ontology changes. This is the decisive reason that we could not simply take Klein's Ontology of Change for our research purpose.



```
<AddingInverseProperty rdf:ID="AddingInverseProperty_Instance1">
    <hasChangeHappenedOnProperty>
      <Property rdf:ID="Internal_Data_Provider"/>
    </hasChangeHappenedOnProperty>
    <hasVersion
rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >1.6.3</hasVersion>
    <hasAddedInverseProperty rdf:resource="#Parent_Data_Provider"/>
 </AddingInverseProperty>
```

The representation of adding inverse property in Log Ontology II

```
====================================================================

<rdf:Description rdf:about="#Add_Inverse_Property_Instance1">
  <rdf:type>
    <owl:Class rdf:about="#Add_Inverse_Property">
    </owl:Class>
  </rdf:type>
  <owl0:new_Filler rdf:resource="#Parent_Data_Provider"/>
  <owl0:old_Filler rdf:resource="#null"/>
  <owl0:on_Property rdf:resource="#Internal_Data_Provider"/>
</rdf:Description>
</rdf:RDF>
```

The representation of adding inverse property in Ontology of Change

FIGURE 4.6: The comparison of the OWL representation of adding inverse property operation between Log Ontology II and Ontology of Change

- Example of adding inverse property. Both ontologies represent clearly the added inverse property *Parent_Data_Provider* of the existing property *Internal_Data_Provider* (shown in Figure 4.6). According to [8], meta-information such as author and date are included in the change representation to specify who and when made the changes to the ontologies. In our Log Ontology II, we did not include such information. Instead, we incorporate version information which is used to record in which version the change has been made. This would benefit us from assisting the retrieval of the change information within one version or across multiple version history of one ontology. It is also noticed that Ontology of Change includes the property *old_Filler* which is required to undo the changes performed on the ontology, or when the operations need to be inverted. This ability is missing in the representation of Log Ontology II.

As a summary, Log Ontology II and Ontology of Change take different views on the ontology changes. Ontology of Change focuses on the whole tribe of ontology changes during ontology evolution process. Therefore, their efforts in the subject of ontology change are to investigate and identify as many types as possible and classify them based on ontology change types' inherent relationships. As noted in the previous section, Ontology of Change is nearly a complete collection of ontology change types. The stance of our Log Ontology II is standing on the applicable point. We simplified Klein's ontology change types list to put more efforts in making the concept organisation of the ontology change types more suitable to collect the change information via multiple ontology versions of the same ontology which enable the change process traceable.

Chapter 5

Future Work

Our efforts in using change-tracks to eliminate or reduce the impacts that ontology change can have on any dependent applications and services confronted some difficulties in the further development stage. Our current method is using a semantic log of ontology change to amend RDQL queries sent to the ontology to update queries automatically and maintain the flow of knowledge to the applications and services. As it is already noted in the first section of Chapter 4, when amending the RDQL queries, the system deals with the entities within the queries in isolation and does not prioritise the entity replacement. This would cause unpredictable results when the complex changes happen. To solve this problem, one of the solutions would be on-the-fly keeping track of more detailed change information while updating ontologies to enable better semantic representation of the complex ontology change and further to assist the maintenance of the service in such a complicated situation, for example identifying the prioritsed entity from the correlated complex changes.

Based on this requirement, we have re-built Log Ontology I to Log Ontology II to enable an improved concept organisation and structure for better representation of the complex ontology change information. In the next one and half year the main research work would be putting great efforts on how to better and efficiently keeping track of more detailed change information while updating the ontologies to enable monitoring the series of ontology changes which would be beneficial to the much more complicated service maintenance situations such as semantically prioritising the updated entity from the correlated complex ontology changes within the query.

To implement this idea, following work would be achieved in the one and half year:

1. Extending the Change Management plug-in of Protégé to keep track of more detailed change information by incorporating the concept structure of Log Ontology

II into the framework of this plug-in. As described in section 2.2, when Change Management plug-in is activated, the changes made to the ontology would be kept as the records bearing with timestamp and author information. Therefore, the change process is monitored and each record of the ontology change history is declarative. The plug-in could display all the changes to the current project (ontology) with the annotations from the author(s) of the ontology update process. For a single ontology change, it could represent its associated changes as well. The above functions would greatly enable us to capture the ontology change information on-the-fly, in particular the ability to stamp the time record on every change operation and the ability to group the associated changes for a specific one. Both functions are important components to implement the ideas of prioritising the updated entity within the query among the complex ontology changes by means of keeping the ontology changes traceable. However, both functions are necessary but not sufficient. We identified that there are still three functions missing in this plug-in in order to achieve our research task. We will put the great efforts on these in the future work.

- Providing the support to keep track of the change information across multiple versions of the same ontology. The existing implementation of Change Management plug-in could merely work on one version of the same ontology. It keep track of every change operations happened during one version history. After upgrading this version to a newer one, the change information would be impossible to differentiate. Only way to achieve this is through telling the different timestamp of the change operations. Imagining another ontology developer might go back to review the previous version of this ontology while updating the new version. He/she might find some errors at this stage and make the relevant corrections to the previous version. The timestamps of previous version are falling into the same time slot as those of the new version. Therefore, it is unrealistic to differentiate the ontology versions by telling the timestamps. We need separate ontology version information to group the ontology change operations. With the support of the version information, this plug-in could represent change information across multiple ontology versions.
- Providing the support to relate atomic change operations captured by the plug-in to form the composite and complex ontology changes. Currently, Change Management plug-in could capture every change operation you have made to the ontology while updating. This necessarily enables us to present change process step by step. However, the presentation level of this plug-in is lower than what we need to present to end-user. It is not convenient for the end-user to have a clear mind about what is the exact change made to the ontology. For example, moving a class action would be kept by this plug-in as a record of removing a class and a record of adding the same class. When looking through the change information listed in the plug-in,

the end-user would not identify that these two operations are intended to be a composite operation. In this case, it is necessary to have the ability to group the fine-grained atomic change operations into the higher-ordered changes. As a result of this implementation, the change process would be a dual-way process, i.e, composition and decomposition of change operations. It would be beneficial not only for the applications to retrieve the change information at will from any stages of the change process but also for the ontology engineers to curate the changes performed on the ontology to decide the final ready-released version.

- Providing the support to convince ontology engineers to annotate the information about the source of changes. Taking a look back at the example in section 4.1 (refer to Figure 4.1), the application could not decide how to handle with the property *hasAge* with the query because *hasAge* exists in both versions. In this circumstance, if the ontology engineers annotate that *hasAge* is one of the properties of class *Person* while updating then the application would know that *hasAge* should be check to be fixed at this stage. By enabling this ability, presenting ontology change history and keeping ontology change traceable would be easily implemented. As well as, it provides a strong reference for the applications to prioritise the updated entity within the query in some situations.

With the implementation of the above functions combined with its original powers to keep track of ontology changes while update, it is believed that ontology changes with the correlated complex relationships as a whole will be kept completely, i.e. the semantics of ontology changes during the ontology evolution process will be reserved with intactness. It would be helpful and instructive for us to understand deeply what the process of ontology change is and direct us to manage the ontology changes within the deployed application in an efficient and semantic manner. Following the extension work, we would design a series of experiments on a large of ontology versions of the same ontology, for example, large numbers of BioSAIL ontology versions. The purpose is to demonstrate that the requested services upon the underlying changing ontologies are maintained by making use of the ontology change information which is well reserved on-the-fly from the beginning of the update process (active ontology versioning method), which is much better than by making use of the change information identified and characterised by comparing ontology versions (passive ontology versioning method). The better we understand from the experiments how ontology changes happen, the more intelligently we maintain the services upon the changing ontology.

2. Semantically prioritising updated entity from the correlated complex ontology changes within the user's query according to the change information stored in Log Ontology II by monitoring the change information collected from the large

amount to user's queries submitted to the applications. With the assistance from improved Change Management plug-in, applications would be able to represent much more change information on-the-fly, including the version information, timestamp, associated changes for a specific entity and so on. These information would give ontology engineers the clues to decide the order of the ontology changes, to streamline the relationships between the specific changed entity and its associated changes, and to get an overview of the change process within one specific version or across the multiple versions of the same ontology. The more change information we keep track from the ontology updating process, the more decisive factors we could use in our prioritise process. In addition, we would also monitor how the ontology has been queried via the large amount of information within the query logs of the application to make the recommendations on which entity would probably be the prioritised object.

3. Building up the visualisation services on top of Log Ontology II to provide ontology engineers with a comprehensive overview of the history of ontology changes during the course of ontology development. Because the ontology change operations captured during the update process contain the information on the concepts to which the change operations were applied and who and when the changes were made, it is readily to process this information to provide the ontology concept history. Likely, for each concept of the ontology, it is readily to present the history of its change operations, who performed them and when. Because the annotation information might also includes the engineers' rationale describing the changes performed on the ontology and why they made the changes, the visualisation services would not only present what change and where made to the concept but also list out why the change made to the ontology. Our goal are providing the ontology engineer with a clear and comprehensive overview of ontology change history; assisting them to better understand the entire change process in the ontology reviewing process; and helping them make accept/reject decision to release the ontology version.

Chapter 6

Conclusion

In this thesis, we discussed the ontology changes as an open problem with respect to the exploration of what is the efficient and appropriate fashion to represent and manipulate ontology change to minimise its potential serious effects on dependent applications and services. For this purpose, we built the Log Ontology I to store and manage change information between ontology versions. As an experimental example at this stage, we populated the Log Ontology I with information about changes between two versions of the CRM ontology. We developed a prototype system that analyses the incoming queries, amends the entities within the queries according to the change information stored in the Log Ontology I, and informs the end-user of any changes and actions taken. We have successfully tested it on some of the changes we identified on the CRM ontology. We showed that with the extra support from the manipulation of ontology changes stored in Log Ontology I, some of the queries that are targeting parts of the ontology that have changed can be updated and processed properly. However, we confronted the difficulties in semantically prioritising the updated entity with the query, in particular in the correlated complex ontology change relationships. After revisiting the case study and analysing the experiments, we identified that the recognised limited scope of change types and the represented information related to the ontology changes which lacks the semantic connections between each other restrict our ability to enable the semantically prioritise of updated entity in the more complicated service maintenance tasks. We refurbished the concept organisation and structure of Log Ontology I to form a new Log Ontology II which provided a good foundation in the term of semantics allowing us to better keep track of the change information while updating the ontology. In the end, we discussed the future direction in terms of: (1) how to develop the semantics-enabled concept organisation of Log Ontology II to better construct the ontology change information during the update process which would enable semantically prioritise the updated entity within the query; (2) how to better control the impacts caused by ontology changes and maintain services continuously delivered by the ontology-based applications by making use of the well-reserved ontology change semantics stored within Log Ontology II.

Acknowledgements

I would like to thank my supervisors and colleagues for allowing me to contribute to a variety of interesting and stimulating work. Special thanks for the excellent technical supports from my colleague David Dupplaw.

I would like to give my sincere thanks to Zachary Pincus, Michel Klein and Mark Musen from Stanford University for providing me with a large number of BioSAIL ontology versions for my research work.

This work has been supported under the Advanced Knowledge Technologies Interdisciplinary Research Collaboration (AKT IRC), which is sponsored by the UK Engineering and physical Sciences Research Council under grant number GR/N15764/01.

Bibliography

[1] Stojanovic, L., et al. User-driven ontology evolution management. In *Proceeding of the 13th International Conference on Knowledge Engineering and Knowledge Management, Ontologies and the Semantic Web*, pages 285–300, 2002.

[2] Flouris, G., and Plexousakis, D.,. Handling ontology change: Survey and proposal for a future research direction. Technical report, Institute of Computer Science, FO.R.T.H., September 2005.

[3] Berners-Lee, T. Handler, J. and Lassila, O. The semantic web. *Scientific American*, May 2001.

[4] Heflin, J. and Hendler, J. Dynamic ontologies on the web. In *Proceeding of the 17th American Association for Artificial Intelligence Conference (AAAI)*, pages 443–449, Menlo Park, CA, US, 2000. AAAI/MIT Press.

[5] Huang, Z. and Stuckenschmidt, H. Reasoning with multi-version ontologies: A temporal logic approach. In *Proceeding of the 4th International Semantic Web Conference (ISWC)*, Galway, Ireland, 2005.

[6] Kiryakov, A. and Ognyanov, D. Tracking changes in rdf(s) repositories. In *Proceeding of the 13th International Conference on Knowledge Engineering and Management, Ontologies and the Semantic Web*, Spain, 2002.

[7] Klein, M.,. Versioning of distributed ontologies. Public, December 2002.

[8] Klein, M. *Change Management for Distributed Ontologies.* PhD thesis, Vrjie Universiteit, Amsterdam, 2004.

[9] Klein, M. and Fensel, D. Ontology versioning on the semantic web. In *Proceeding of International Semantic Web Working Symposium (SWWS)*, Stanford University, California, U.S.A, 2001.

[10] Klein, M., and Noy, N.F.,. A component-based framework for ontology evolution. Technical report, Department of Computer Science, Vrijie Universiteit Amsterdam, March 2003.

[11] Noy, N.F., Kunnatur, S., Klein, M., and Musen, M.A. Tracking changes during ontology evolution. In *Proceeding of the 3rd International Semantic Web Conference (ISWC2004)*, Hiroshima, Japan, November 2004.

[12] Noy, N.F., Chugh, A., Liu, W., and Musen, M.A. A framework for ontology evolution in collaborative environments. In *Proceeding of The 5th International Semantic Web Conference (ISWC 2006)*, 2006.

[13] Noy, N.F., and Musen, M.A. Promptdiff: A fixed-point algorithm for comparing ontology versions. In *Proceeding of the 18th National Conference of Artificial Intelligence (AAAI)*, pages 744–750, Edmonton, Alberta, Canada, 2002.

[14] Klein, M., Kiryakov, A., Ognyanov, D., and Fensel, D. Ontology versioning and change detection on the web. In *Proceeding of 13th International Conference on Knowledge Engineering and Management*, Siguenza, Spain, 2002.

[15] Plessers, P., and Troyer,O.De.,. Ontology change detection using a versioning log. In *Proceeding of the 4th International Semantic Web Conference (ISWC)*, Galway, Ireland, 2005.

[16] Hall, W. Shadbolt, N. and Berners-Lee, T. The semantic web revisited. *IEEE Intelligent Systems*, 2006.

[17] Kiryakov, A. Simov, K. and Ognyanov, D. Ontology middleware: Analysis and design. Technical report, Onto Text Lab., Sirma AI Ltd., 2002.

[18] Haase, P., Harmelen, F.van, Huang, Z., Stuckenschmidt, H., and Sure, Y. A framework for handling inconsistency in changing ontologies. In *Proceeding of the 4th International Semantic Web Conference (ISWC)*, Galway, Ireland, 2005.

[19] Maedche, A., Motik, B., Stojanovic, L., Studer, R., and Volz, R. Managing multiple ontologies and ontology evolution in ontologging. *Intelligent Information Processing*, pages 51–63, 2002.