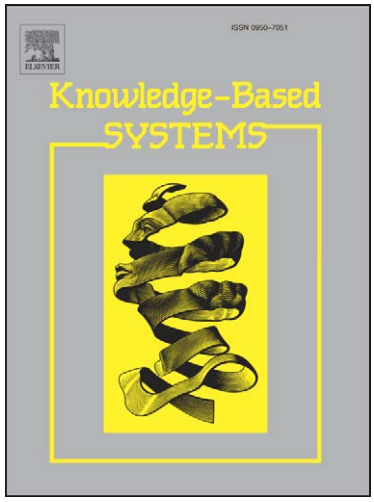

Change Management in Evolving Web Ontologies

Asad Masood Khattak, Khalid Latif, Sungyoung Lee

PII: S0950-7051(12)00132-3
DOI: http://dx.doi.org/10.1016/j.knosys.2012.05.005
Reference: KNOSYS 2306

To appear in: *Knowledge-Based Systems*

Received Date: 17 November 2011
Revised Date: 11 May 2012
Accepted Date: 14 May 2012

Please cite this article as: A.M. Khattak, K. Latif, S. Lee, Change Management in Evolving Web Ontologies, *Knowledge-Based Systems* (2012), doi: http://dx.doi.org/10.1016/j.knosys.2012.05.005

Change Management in Evolving Web Ontologies

Asad Masood Khattak [a], Khalid Latif [b], Sungyoung Lee [a,*]

[a] *Ubiquitous Computing Lab, Department of Computer Engineering, Kyung Hee University (Global Campus), Korea*
[b] *School of Electrical Engineering and Computer Science, National University of Sciences and Technology, H-12 Campus Islamabad, Pakistan*

Abstract

Knowledge constantly grows in scientific discourse and is revised over time by different stakeholders, either collaboratively or through institutionalized efforts. The body of knowledge gets structured and refined as the Communities of Practice concerned with a field of knowledge develop a deeper understanding of the issues. As a result, the knowledge model moves from a loosely clustered terminology to a semi-formal or even formal ontology. Change history management in such evolving knowledge models is a an important and challenging task. Different techniques have been introduced in the research literature to solve the issue. A comprehensive solution must address various multi-faceted issues, such as ontology recovery, visualization of change effects, and keeping the evolving ontology in a consistent state. More so because the semantics of changes and evolution behavior of the ontology are hard to comprehend.

This paper introduces a change history management framework for evolving ontologies; developed over the last couple of years. It is a comprehensive and methodological framework for managing issues related to change management in evolving ontologies, such as versioning, provenance, consistency, recovery, change representation and visualization. The Change history log is central to our framework and is supported by a semantically rich and formally sound change representation scheme known as Change History Ontology. Changes are captured and then stored in the log in conformance with the change history ontology. The log entries are later used to revert ontology to a previous consistent state, and to visualize the effects of change on ontology during its evolution. The framework is implemented to work as a plug-in for ontology repositories, such as Joseki and ontology editors, such as Protege. The change detection accuracy of the proposed system *Change Tracer* has been compared with that of *Changes Tab*, *Version Log Generator* in Protege; *Change Detection*, and *Change Capturing* of NeOn Toolkit. The proposed system has shown better accuracy against the existing systems. A comprehensive evaluation of the methodology was designed to validate the recovery operations. The accuracy of roll-back and roll-forward algorithms was conducted using different versions of SWETO Ontology, CIDOC CRM Ontology, OMV Ontology, and SWRC Ontology. Experimental results and comparison with other approaches shows that the change management process of the proposed system is accurate, consistent, and comprehensive in its coverage.

Key words: Ontology Recovery, Change History Ontology, Ontology Evolution

* Corresponding author. Tel: +82-31-201-2514
Email addresses: asad.masood@oslab.khu.ac.kr (Asad Masood Khattak), khalid.latif@seecs.nust.edu.pk (Khalid Latif), sylee@oslab.khu.ac.kr (Sungyoung Lee).

1. Introduction

Ontologies are formal descriptions of a shared conceptualization of a domain of discourse [14]. Their usage is wide spread in information systems, especially when building a *lingua franca* for resolving the terminological and conceptual incompatibilities between information networks of varying archetype and different provenance [41].

One of the crucial tasks faced by practitioners and researchers in knowledge representation area is to efficiently encode human knowledge in ontologies. Maintenance of usually large and dynamic ontologies, and in particular adaptation of these ontologies to new knowledge, is one of the most challenging problems in Semantic Web research [10,26]. Due to the uncontrolled, decentralized, and complex nature of the Semantic Web, ontology change management is a complicated and multifaceted task. Ontology change management deals with the problem of deciding which modifications to perform in an ontology; implementation of these modifications; and the management of their effects in dependent data structures, ontologies, services, applications, and agents [11]. This has led to the emergence of several different, but closely related, research areas, such as ontology evolution, versioning, integration, and merging [11,15,34]. Ontology evolution research also deals with other associated problems, such as ontology matching, which otherwise are fundamentally different [22]. Ontology evolution covers modifications in ontology when there is a certain need for a change as *Communities of Practice* concerned with the field of knowledge develop a deeper understanding of the domain of discourse. Ontology evolution takes place when the perspective under which the domain is viewed has changed [35].

The ontology evolution process deals with the growth of the ontology as well as capturing and accommodating new information [28,43]. As an ontology evolves to a new state, the dependant ontologies and services may become invalid [3,11,22]. Consequently, ontology change management solutions have to answer a number of questions [12]. One such question is, "how to maintain all the changes in a consistent and coherent manner?" During ontology enrichment, modifications are made to the ontology and it evolves from one consistent state to another without preserving information about the previous state. This makes it harder to refer to the previous state unless the changes are preserved in some form. Moreover, the role of change history information becomes critical when an unauthorized user makes changes, or when an ontology engineer wants to revert the changes. A storage structure for such information is also crucial for effective retrieval. Other aspects, such as change traceability and effective visualizations of ontology changes, must also be taken care of by a comprehensive change management framework.

The general aim of this research work, is to provide a mechanism for temporally tracking down changes to an ontology throughout it's life span. In particular the objective is to work on ontology change management, recovery, and visualization of changes and their effects on an ontology to understand the ontology's evolution behavior. To achieve this, all these changes are maintained and managed in a coherent manner.

A Semantically enriched Change History Ontology (CHO) is developed and used to record the ontology changes in a Change History Log (CHL). For proof of the concept, we have developed the system as a plug-in for the ontology editor Protege, that listens and logs all of the ontology changes in CHL. Afterwards, these logged changes are used for ontology recovery (Roll Back and Roll Forward) purposes. We have designed and implemented the Roll Back and Roll Forward algorithms. The logged changes are also used for visualization of changes and their effects at different stages of the evolving ontology. A play back feature is provided to navigate the ontology history for a better understanding of its evolution behavior.

To verify and validate the developed system, we have compared the change capturing ability of the developed plug-in i.e., *ChangeTracer*, with the *ChangesTab* and *VersionLogGenerator* of Protege, *ChangeDetection*, and *ChangeCapturing* of NeOn Toolkit. The results showed that our developed plug-in has better accuracy than the *ChangesTab*, *VersionLogGenerator*, and *ChangeDetection*; whereas having almost same accuracy as *ChangeCapturing*. Moreover, the proposed system uses *difference()* method to capture and log all the missing changes, whereas, the other systems do not. For validating recovery algorithms (i.e., *RollBack* and *RollForward*), we have tested them on standard data sets, i.e., Semantic Web Technology Evaluation Ontology (SWETO) [1], CRM Ontology [6], Semantic Web for Research Communities (SWRC) Ontology [44], and Ontology Metadata Vocabulary (OMV) Ontology [16]. A high accuracy for both algorithms was observed in

these tests. The overall working shows that the proposed system is accurate, consistent, and comprehensive in nature.

The rest of this paper is organized as follows: Section 2 sets the stage by providing the background subject knowledge about the change management and recovery techniques available in ontology and its sibling domains. We have also contrasted our work with other related approaches. A comprehensive description of the semantic structure developed for maintaining the ontology changes is presented in Section 3. Section 4 presents the different possible applications of CHL. Section 5 discusses the recovery algorithms with a running example and presents implementation details. Section 6 presents the details on system implementation. In Section 7, a comparative analysis and validation of results of the proposed methodology using Semantic Web Technology Evaluation Ontology (SWETO), CIDOC Conceptual Reference Model, SWRC Ontology, and OMV Ontology is presented. Finally, we have concluded the research work and have presented an outlook to future research aspects in Section 8.

2. Background and Review

Ontology change management deals with the problem of deciding which modifications to perform in ontology in response to a certain need for change [9]. This mechanism ensures that the required changes are reflected in the ontology, and that it is in a consistent state. It deals with four different aspects [10,11,38]. (1) *Ontology evolution* is the process of modifying ontology in response to a certain change in the domain or its conceptualization. (2) *Ontology versioning* is the ability to handle an evolving ontology by creating and managing different versions of it. (3) *Ontology integration* is the process of composing the ontology from information found in two or more ontologies covering related domains. (4) And lastly, *ontology merging* is the process of composing the ontology from information found in two or more ontologies covering highly overlapping or identical domains.

To support the dynamic nature of the Semantic Web, there must be some mechanism to cope with the continuous evolution of domain models and knowledge repositories. Therefore, it is important to manage the ontology changes effectively, and to maintain the relationship between the changes and models [30]. A lot of research on schema evolution has been carried out in relational databases. Schema evolution handles changes in a schema of a populated database without losing data, and provides transparent access to both old and new data through the new schema. It is a complicated task considering the dynamics of ontologies [34], and is also critical to the support of networked ontologies. Ontology evolution and versioning could be amalgamated in a holistic approach to manage ontology changes as well as their effects.

2.1. *Ontology Evolution Process*

The process of ontology evolution has the following two variants: *Ontology Population* and *Ontology Enrichment*. New instances of prior coded concepts can be introduced or existing instances can be updated. As a result, the A-Box is changed and reflects new realities. This is called ontology population. Where as in *Ontology Enrichment* process, new domain concepts; properties; or restrictions are introduced or existing ones are updated. This later variant refers to the changes in the schema or T-Box. Overall, the process of evolution takes ontology from one consistent state to another [4,43]. Fig. 1 depicts an overview of this process and shows an interconnection of needed building blocks. In a holistic manner, these components ensure that the ontology has evolved to a consistent new state, incorporating all the required changes. These components are comprehensively discussed in the subsequent sections.

2.1.1. *Change Detection and Description*

The first step in the process is to detect changes, whether the suggested changes are already present in the target ontology. Additionally, schema and individual level differences can be detected effectively, as reported in [46]. In case the concept in focus is totally new and there is no additional information, then the H-Match algorithm [5] is used. It takes the new concepts for addition and the target ontology as inputs, and returns the best matching concept in the ontology in order to identify a taxonomic position for the concept [4].

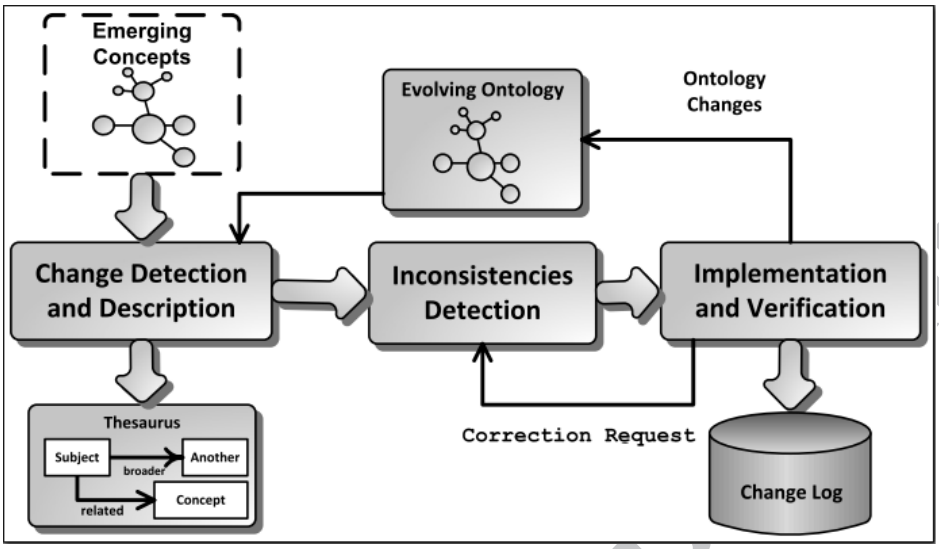

Fig. 1. The Ontology Evolution Process when a change in ontology is requested.

The identified changes: elementary (atomic/simple) change e.g., renaming a class or a property; or composite (complex) change e.g., merging two hierarchies with all their constraints applied, are represented in a consistent format. These changes are first assembled in a sequence, followed by the change implementation. Our focus is on atomic changes and we also consider all the composite changes to be an ordered sequence of atomic changes. Change History Ontology [20] representation is used to represent changes. This representation is also used to log ontology changes in the Change History Log (CHL) discussed later.

2.1.2. *Inconsistencies Detection*

In this module ontology changes are analyzed in a systematic manner to ensure that the consistency of the ontology is not lost. Two types of inconsistencies can occur: (1) syntactic inconsistencies occur when an undefined or inconsistent construct from meta ontology level is used; (2) semantic inconsistencies occur when the meaning of the ontology entity is changed due to the changes. To keep the ontology in a consistent state, further changes are inferred by taking into account the newly introduced ontology changes, known as induced and deduced changes respectively.

2.1.3. *Change Implementation and Verification*

This process covers the following three aspects: 1) change should be applied in complete isolation, and must be atomic, durable, and consistent; 2) each implemented change is verified against the change request; and 3) All the implemented changes must be logged in the CHL to keep track of the changes performed in an ordered manner.

2.2. *Ontology Versioning and Change Management*

Various strategies could be adopted to preserve the changes in ontology, including the use of a database or semi-structured log files. Different researchers have provided various techniques to maintain these changes. For example, *Changes Tab* [33] in Protege and *Change Capturing* [38], listens to the ontology changes and preserve them in a log file, which can be used to add, delete, and modify annotations about changes made to the model. *Changes Tab* [33] and *Change Capturing* [38] can be configured for a client-server model in Collaborative Protege and NeOn Toolkit respectively. It also comes with a conventional tabular view for searching and navigating within the changes.

Klein [26] has done a significant work on change management for distributed ontologies. The author developed change ontology by modeling both the atomic and complex changes. A comprehensive categorization of

different ontology changes is also provided. This categorization and the change ontology are the foundation of our research work, and are used to model a representational structure for ontology changes.

A similar approach to ours is used by [31] and [38]. Ontology changes are stored in a file as a script following a temporal ordering. The script follows the specifications provided in the Log Ontology [30] and OWL 2 Change Ontology [38]. Upon the user's request, this script file is used to carry out undo or redo operations. Such log files are maintained for particular editing sessions. This way, the command history of one editing session is maintained within Protege and is a candidate for transparent query answering. The *Change Capturing* is responsible for capturing the ontology changes and propagating them to the other instances of the same ontology on different nodes, both locally and remotely [38]. We believe that there should be a mechanism to maintain the changes for a longer time-span to support, for example, mining of change patterns in a networked ontology.

2.3. *Ontology and Database Recovery*

To the best of our knowledge, no ontology editing tool addresses the ontology recovery problem. So it is worth mentioning the recovery techniques available in related fields, such as databases. Relational databases also change and these changes are managed by the database management system for recovery and traceability. The basic purpose of database recovery is twofold. Firstly, it is used to recover the data after system/disk crashes. Secondly, it preserves ACID properties in transactions, and brings the database into a consistent state after transaction errors.

From the ontology recovery standpoint we are not concerned with system crashes. This aspect could easily be delegated to the underlying storage system which can maintain a complete backup/shadow of the whole ontology. On the other hand, recovery from inconsistencies doesn't require a complete archival copy of the database. Several techniques have been proposed in the literature to recover databases, among which logging, check-pointing, shadowing (or shadow paging), and differential tables are the most prominent [7]. Instead of directly updating the actual database tables for a change during a transaction, the intermediate updates may be recorded in a sequential file known as transaction log. The log file serves as historical archive of the recent database operations. Database systems can also maintain a checkpoint record in the log. This log can later be used for data recovery, and to bring the database back into a consistent state.

Similar to the logging method, updates could be accumulated in the differential tables rather than making the changes in the original table or maintaining a complete before image [18]. Three differential tables are maintained for a single database relation: 1) a read-only copy of the base table; 2) a differential table for insertion, wherein all the new tuples are first inserted in this differential table; 3) a deletion differential table, wherein all the deleted tuples are maintained in this differential table. Consider, for example, a $\mathit{Student}$ relation. The read-only table for this relation could be referred to as $\mathit{StudentR}$, the insertion differential as $\mathit{StudentD}^{+}$, and the deletion differential table as $\mathit{StudentD}^{-}$. The following equation could then be used to reflect the updates in the actual table, where the union and difference operators are applied to changes in temporal order.

$$\mathit{Student} = (\mathit{StudentR} \cup \mathit{StudentD}^{+}) - \mathit{StudentD}^{-} \tag{1}$$

2.4. *Change Tracking and Visualization*

Most of the previously discussed ontology change management systems focus mainly on detection and archiving the ontology changes [26]. These changes are not used; however, for undo/redo operations during collaborative ontology engineering [38], or even for ontology recovery. Change traceability and visualization remains an under explored research areas.

In [39], the authors provided a structure for persistent storage of ontology changes and a limited support for change visualization. The work reported in [39], introduces a scheme of logging and visualizing multiple ontology changes. The process starts with detecting changes in two versions of ontology using Prompt [36]

and OntoView [27]. The history manager module then visualizes the changes that were detected and logged in the repository, such as transformation of an attribute from a datatype property to an object property.

In a nutshell, most of the existing systems discussed here provide a mechanism for logging the ontology changes. Their main limitations are: (1) For the most part these systems work on two versions of the same ontology and target versioning rather than covering the ontology evolution process, (2) Temporal traceability and recovery lacks in these systems, and (3) lastly, none of the systems support change visualization which can assist in comprehending the effects of changes on the ontology. Change visualization can also help in understanding the evolution pattern and behavior of an evolving ontology by visually navigating through the history of all the changes.

3. Change History Ontology

Recovery and change traceability are essential ingredients of any change management system for evolving ontologies. We propose here a scheme for representing ontology changes, referred to as *Change History Ontology (CHO)*. A comprehensive framework for change traceability is also presented later. The framework helps in automatically detecting and logging all the changes, triggered by the change request from the ontology engineer. Changes can be recorded according to the defined structure in the repository and can later be utilized to bring the evolving ontology to a previous consistent state. Such a framework can also benefit the temporal traceability of changes and the visualization of the change effects. Details of the *Change History Ontology* are presented below.

3.1. *Overview*

A number of changes, ranging from concepts to properties, can affect an evolving ontology. Most of these changes are discussed in greater length in previous literature [2,26,38]. An understanding of different ontology changes is necessary to correctly handle explicit and implicit change requirements [11,15]. For that purpose we have designed and developed an ontology to capture ontology change requirements and keep track of the change history. The proposed *Change History Ontology* [20] reuses constructs from existing ontologies [30] and ontology design patterns [13]. We have introduced new extensions to the existing schemes and some of the notable mentions are discussed in the subsequent sections.

The core elements of *CHO* are the *OntologyChange* and *ChangeSet* classes. The *OntologyChange* class has a sub-class called *AtomicChange* that represents all the class, property, individual, and constraint level changes at atomic level. On the other hand, the *ChangeSet* bundles all the changes from specific time interval in a coherent manner. The *ChangeSet* is responsible for managing all the ontology changes and arranges them in time indexed fashion. This time indexing also classifies the *ChangeSet* as *Instant* type and *Interval* type. *Instant* type *ChangeSet* holds only one change occurred at some time instant, whereas the *Interval* type *ChangeSet* holds the changes occurred in a stretched time interval.

Different resources are referred using different prefixes in this work. It is important to explain the prefixes used. The concepts under the "ch" prefix are used from change history ontology and the prefixes "log" and "bib" represent the repository log and an example of evolving *Bibliography ontology* respectively. "xml" is used for xml, "rdfs" for rdfs, and "owl" for owl namespaces.

3.2. *Change Set*

We initially introduced the notion of *ChangeSet* a couple of years ago [20]. The same has also been suggested in Change Set Vocabulary [45]. The rationale is that individual changes are not performed in isolation and are usually part of a particular session. On the contrary, *ChangeSet* can be used to group the individual changes from a particular session in order to incorporate a holistic view over an ontology evolution. Logging changes in sets also helps in maintaining and managing the ontology changes corresponding to specific sessions which is also required for ontology recovery, as discussed later. The use of *ChangeSet*(s) is common in versioning systems, such as CVS and SVN. A *ChangeSet* holds information about the changes made during an ontology

engineering session. A *ChangeSet* can span over a stretched interval of time. Its members, atomic changes, are singleton changes on an ontology element at some instance of time. Different changes can be part of the *ChangeSet*, such as modifying details of an ontology class or adding a new object property. *ChangeSet*s also help in maintaining the sequence and grouping of changes. The following example (Fig. 2) of *ChangeSet* instance covers an ontology editing session spanning over half an hour:

```
log:ChangeSet01
    rdfs:type            ch:ChangeSet ;
    ch:hasChangeSetType  log:Interval ;
    ch:hasChangeAuthor   log:ChangeAgent01 ;
    ch:startTime         "2010-01-01 15:12:58+1" ;
    ch:endTime           "2010-01-01 15:43:11+1" ;
    ch:hasChangeReason   "Concept X is split into two levels" ;
    ch:targetOntology    http://seecs.nust.edu.pk/vocab/bib .
```

Fig. 2. A *ChangeSet* example with corresponding meta data including the change agent information, the reason for change, the changed ontology, and the start and end times of the change event. The Fig shows the time spam of *ChangeSet*.

3.3. *Conceptual Design Patterns*

Recently, different ontology development methodologies have emerged [17,47], some of which advocate the reuse of concepts and patterns from foundational ontologies [13]. More specifically, patterns are useful in order to acquire, develop, and refine the ontologies. We have reused two of the fundamental ontology design patterns. The *Participation Pattern* consists of a *participant − in* relation between the ontology resource and the change event, and assumes a time indexing for it [13]. Time indexing is provided by the temporal location of the change in a time interval, while the respective location within the ontology space is provided by the participating objects (see Fig. 3 and 5). As an example, consider Fig. 4 as the description of a *ChangeSet*.

In Fig. 4, a *ChangeSet* instance is described using *CHO*. The start and end times of the changes are reflected by *startTime* and *endTime*, respectively. It also logs information about the change agent and the reason for the changes.

Change history ontology is the backbone of the proposed framework. It binds different components of the framework togethers in order to effectively recover ontology from its previous state. In most of the previous approaches, ontology changes are stored sequentially without preserving their dependence or interlinking with other changes [8,11,26,30,38]. Change history ontology, on the contrary, uses *ChangeSets* to group and time indexing of changes in a session to preserve coherence of all the ontology changes. A *ChangeSet* is a

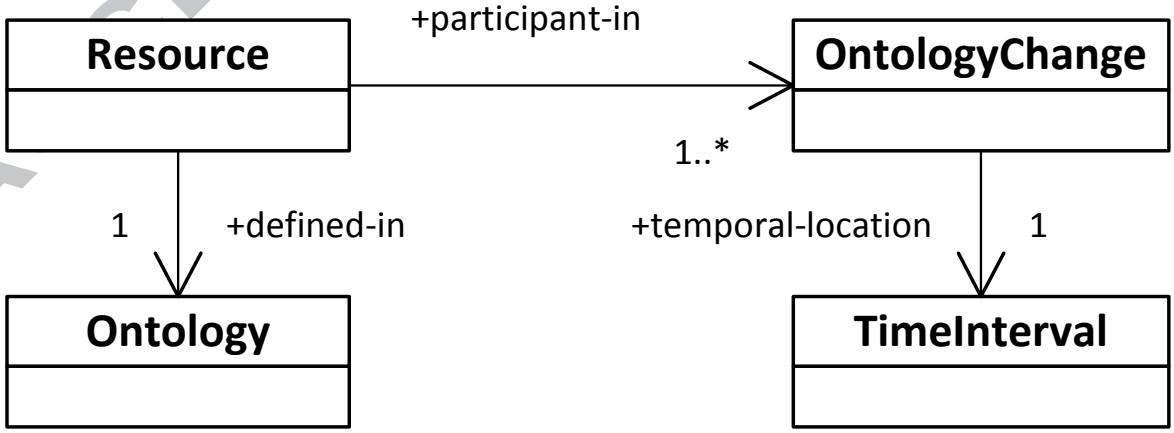

Fig. 3. Realizing Participation Pattern in Change History Ontology.

```
log:ChangeSet192
    rdfs:type           ch:ChangeSet ;
    ch:hasChangeSetType log:Interval
    ch:hasChangeAuthor  log:ChangeAgent2 ;
    ch:startTime        "2010-01-02 16:32:58+1" ;
    ch:endTime          "2010-01-02 16:53:11+1" ;
    ch:hasChangeReason  "Changes after applying rigidity meta property." ;
    ch:targetOntology   http://seecs.nust.edu.pk/vocab/bib .
```

Fig. 4. Example of a *ChangeSet* instance spanning over a time interval.

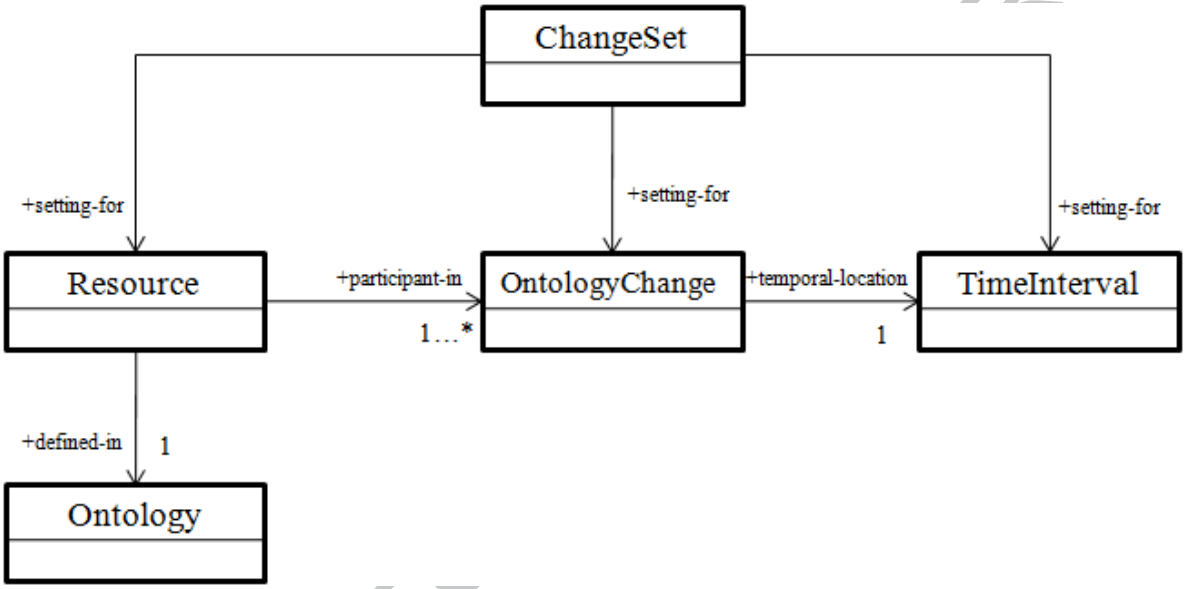

Fig. 5. Reification of time-indexed participation: ChangeSet is a setting for a change event, ontology resources participating in that change event, and the time interval in which the change occurs [24].

setting for atomic changes. One ontology resource participates in a change event at one time interval. Fig. 5 shows diagrammatic depiction of this pattern. The complete change history ontology is available online [1]. Core classes and concepts in the ontology are also shown in Fig. 6.

3.4. *Provenance*

The proposed change history ontology captures provenance information, such as the change author, reason, and timestamp. The author can be an ontology engineer making changes using an ontology editor, or a software agent requesting for some changes, such as an agent during an automatic ontology mapping task. Fig. 7 depicts an instance of the *ChangeAgent* class from *CHO*.

3.5. *Change Types*

The change history ontology supports three types of change operations corresponding to the CRUD interfaces in databases (except the read operation). *Create* allows the addition of new facts and vocabulary in ontology, such as *ClassAddition*, *PropertyAddition*, and *IndividualAddition*. *Update* operation is used for modifying existing triples, such as renaming a class, property, and individual through *ClassRename*,

[1] http://uclab.khu.ac.kr/ext/asad/CHOntology.owl

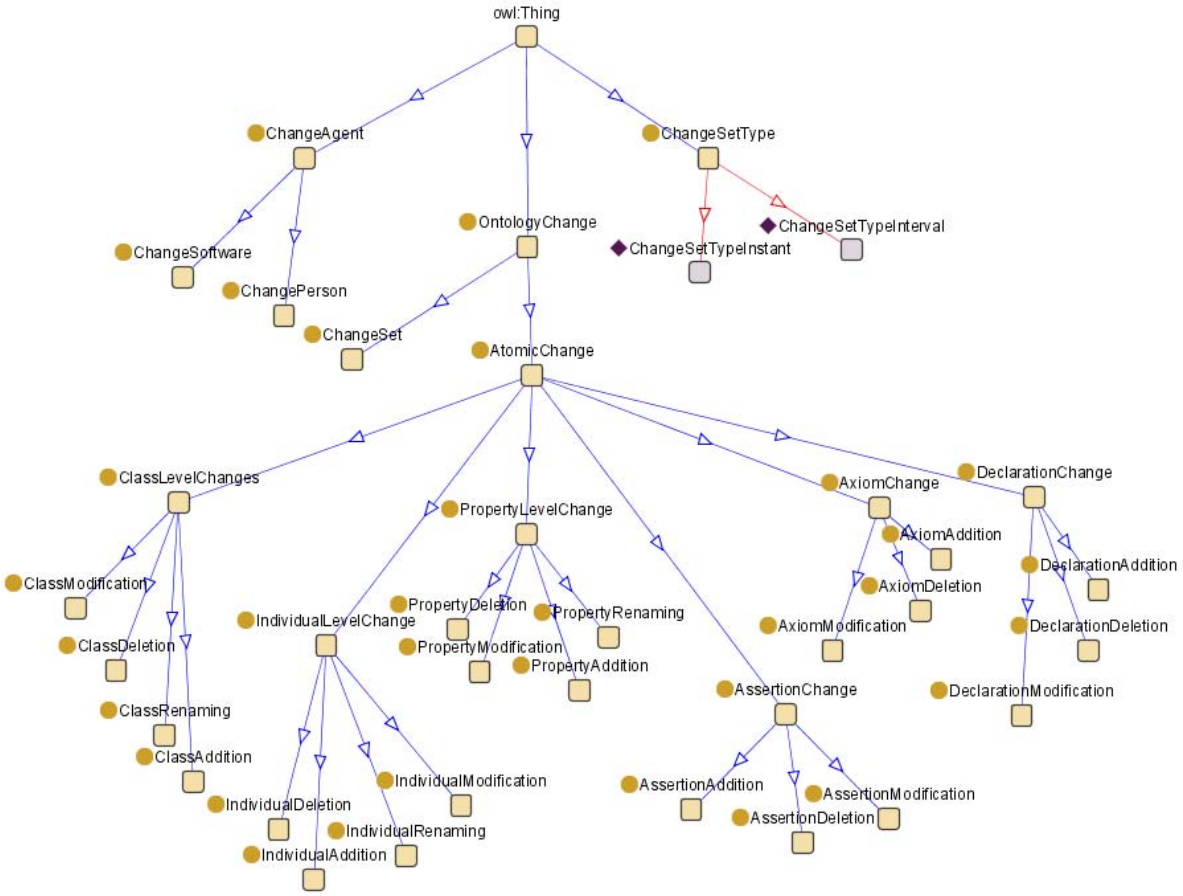

Fig. 6. Snapshot representing core classes of Change History Ontology.

```
log:ChangeSet01
    rdfs:type           ch:ChangeSet ;
    ch:hasChangeSetType log:Interval ;
    ch:hasChangeAuthor  log:ChangeAgent01 ;
    ...
log:ChangeAgent01
   rdfs:type            foaf:Person, ch:ChangeAgent;
   ch:fullName          "Asad Masood".
```

Fig. 7. Representation of the Author of a *ChangeSet* using CHO.

PropertyRename, and *IndividualRename* respectively. And lastly, *Delete* operation serves for removing axioms from the ontology, such as *ClassDeletion*, *PropertyDeletion*, and *IndividualDeletion*. The following axioms depict parts of the conceptual representation of this aspect:

$$\begin{aligned} \mathit{OntologyChange} \equiv\ & \exists \mathit{changeTarget}.(\mathit{Class} \sqcup \mathit{Property} \sqcup \\ & \mathit{Individual} \sqcup \mathit{Ontology}) \sqcap \exists \mathit{changeType}.(\\ & \mathit{Create} \sqcup \mathit{Update} \sqcup \mathit{Delete}) \end{aligned} \tag{2}$$

$$\mathit{ClassChange} \equiv \mathit{OntologyChange} \sqcap \forall \mathit{changeTarget}.\mathit{Class} \tag{3}$$

$$\mathit{ClassAddition} \equiv \mathit{ClassChange} \sqcap \forall \mathit{changeType}.\mathit{Create} \tag{4}$$

$$\mathit{SubClassAddition} \equiv \mathit{ClassAddition} \sqcap \forall \mathit{targetSubClass}.\mathit{Class}$$

$$\sqcap = 1\mathit{targetParent} \tag{5}$$

For instance, the snippet in Fig. 8 represents the addition of a new subclass. *SubClassAddition* is defined as a subclass of *ClassAddition* and is a type of *ClassChange* event. The *hasChangedTarget* represents the newly added class in the *Bibliography ontology*. As the new class is a subclass, the *hasTargetParent* property connects the newly added class with its parent class through a subclass assertion. The *hasTimeStamp* represents the exact time of the change event, whereas, the *isPartOf* connects the change to the corresponding *ChangeSet* instance.

```
log:ClassAddition01
    rdfs:type           ch:SubClassAddition ;
    ch:hasChangedTarget bib:Transaction ;
    ch:hasTargetParent  bib:Journal ;
    ch:hasTimeStamp     "2010-01-01 15:12:59+1" ;
    ch:isPartOf         log:ChangeSet01 .
```

Fig. 8. An example of Transaction class addition as a subclass of parent class Journal and its representation using CHO.

3.6. *Temporal Ordering*

A time stamp is added with each ontology change. Though a single change is performed at an instance of time, it is common that several changes are performed over an extended time interval. A single change is modeled as a change at a time instance, whereas a sequence of changes is considered as one *ChangeSet* spanned over a time interval. So for every change entry that corresponds to a *ChangeSet*, a timestamp value is added. This helps in keeping the ontology change entries in an order. The following snippet in Fig. 9 is an example of timestamp value added to a Property delete event.

```
log:PropertyDeletion01
    rdfs:type           ch:PropertyDeletion ;
    ch:hasChangedTarget bib:title ;
    ch:hasPropertyType  owl:DataType Property
    ch:hasTimeStamp     "2010-01-01 15:24:31+1" ;
    ch:isPartOf         log:ChangeSet01 .
```

Fig. 9. Example showing the timestamp value attached with the Property deletion change instance using CHO.

3.7. *CHO Modeling Language*

To represent the intricacy of changes in; classes, properties, individuals, and constraints, for example, quite a large number of classes with associated object and datatype properties are modeled. This helps in recording all the relevant information about a specific change. The properties in change history ontology, which link the change with its target, are represented as annotation properties. Similarly, object properties are modeled to hold the information of changing classes by setting their range to *owl:class*. Consequently, the model still conforms to OWL-DL; however, it supports very limited DL inference. Advantages of this approach are reducing the likelihood of error, avoiding string manipulation, and removing ambiguity about the change target.

$$\mathit{SubClassAddition} \equiv \mathit{ClassAddition} \sqcap \forall \mathit{targetSubClass}.\mathit{Class}$$
$$\sqcap = 1\mathit{targetParent}$$

3.8. *Complex Changes*

In addition to simple class additions, deletions, and renaming, complex facet modification information is also recorded. Examples include property scope restrictions, equivalence, disjointness, and complex union classes. Similarly, property modification details, such as change in domain/range, setting upper-bound and lower-bound for property values, symmetric, equivalent, inverse, and functional property axioms, are also recorded. Fig. 10 represent information about changes made in the domain of a property and is an example of a complex change event. The *Document* class has two subclasses: *TechnicalDocument* and *ResearchDocument*. The *Document* class is also the domain of an object property *author*. Suppose, due to some reasons the *Document* class gets deleted. This deletion is a complex change event as it will also result in deletion of the subclasses and also unsetting the domain of *author* property. The system records all the changes one by one at atomic level in a sequence. In this deletion event, the change order triggered by Protege is the deletion of the subclasses first, then the deletion of domain of a property, and at the end the deletion of the *Document* class. Proposed system listens to all these changes and logs them in CHL.

```
log:ClassDeletion01
    rdfs:type           ch:SubClassDeletion ;
    ch:hasChangedTarget bib:TechnicalDocument ;
    ch:hasTargetParent  bib:Document ;
    ch:hasTimeStamp     "2010-01-01 15:14:47+1" ;
    ch:isPartOf         log:ChangeSet01 .

log:ClassDeletion02
    rdfs:type           ch:SubClassDeletion ;
    ch:hasChangedTarget bib:ResearchDocument ;
    ch:hasTargetParent  bib:Document ;
    ch:hasTimeStamp     "2010-01-01 15:14:47+1" ;
    ch:isPartOf         log:ChangeSet01 .

log:DomainDeletion01
    rdfs:type           ch:DomainDeletion ;
    ch:hasChangedTarget bib:author ;
    ch:hasDomain        bib:Document ;
    ch:hasPropertyType  owl:ObjectProperty ;
    ch:hasTimeStamp     "2010-01-01 15:14:48+1" ;
    ch:isPartOf         log:ChangeSet01 .

log:ClassDeletion03
    rdfs:type           ch:ClassDeletion ;
    ch:hasChangedTarget bib:author ;
    ch:hasTimeStamp     "2010-01-01 15:14:48+1" ;
    ch:isPartOf         log:ChangeSet01 .
```

Fig. 10. Complex (compound) change resulting from a single change event (Document class deletion).

4. Change History Log (CHL)

In this section we introduce the change history logging scheme. On top of CHL, different applications are possible and are briefly discussed in this section. The subsequent sections introduce and explain two of the applications (log-based traceability and recovery procedures) discussed in this section.

Similar to relational databases, our methodology relies on a logging technique to persistently store ontology changes. Logged changes help in recovering a previous state of the ontology after, for example, un-authorized

changes, version conflicts, or even an inconsistent state of ontology due to accidentally closing the ontology editor. The changes are automatically preserved in a time-indexed manner in a triple store embedded with the framework. Recovery is manually triggered by a knowledge engineer collaboratively building the ontology. The change description in the log conforms to the change history ontology. Each entry in the log is an instance of either the *ChangeSet* or *OntologyChange* class. The log also preserves the provenance information about the changes, such as who made the changes, and when and why these changes were made.

The proposed change history management framework offers numerous benefits ranging from reconciling ontology mappings to increased understanding of ontology evolution process. Some of its applications are briefly discussed below.

Query Reformulation: As an ontology evolves with time, the systems using this ontology also need to be updated to provide proper synchronization and avoid query breakage. To answer a user query, the system first consults CHL to test if the underlying representation scheme has changed. If the ontology has changed then the query from system is reformulated to accommodate the changes [30] and then executed over the evolved ontology. This also helps in avoiding the breakage of query from a system which is still not updated for the evolved ontology. Very recently a SPARQL-based push technique was proposed to broadcast notifications to change listeners [42].

Reconciling Ontology Mappings: Providing reliable mappings among evolving ontologies is a challenging task [40]. *CHL* can contribute towards a solution for this problem. Mappings are basically established between ontologies for resolving the terminological and conceptual incompatibilities. Ontology evolution from one consistent state to another makes existing mappings between ontologies unreliable and stale due to changes in the mapped resources. So there is a need for mapping evolution to eliminate discrepancies from the existing mappings. For example, mappings are established between two ontologies. Due to a change in any of the source ontologies, the existing mappings may become void. As CHL logs all the changes, these changes can be used to reconcile the mappings between evolved ontologies instead of re-initiating the complete mapping generation process, which is time consuming. For detailed procedure on time efficient reconciliation of mappings between evolving ontologies, please refer to [24,25,29].

Change Management and Ontology Recovery: Change history log records all the changes with time-indexing as per the design pattern in the *CHO*. Time-indexing helps in recovering the ontology into a previous consistent state [21]. Managing ontology changes during evolution is also helpful for a new user to get understanding of the changes made. In addition to *change reason*, annotations can also be added with all the logged changes and associated artifacts to help understanding the changes in ontology, data, and application [20]. The log can also be used to understand the semantics of change on the available ontology constructs. As all the changes are logged, one can also try to deduce pattern of change by applying machine learning algorithms.

Temporal Traceability of Ontology Changes: Ontology visualization tools and plug-ins are available in abundance. None demonstrates ontology evolution and changes. A new breed of ontology visualization tools can be implemented using change history log to visualize different ontology states. Such a visualization of change effects on ontology can help in temporally tracing the ontology changes and better understanding the evolution behavior of ontology [21]. When a user requests to visualize an ontology at a particular time instant, all changes after that time interval are reverted back and the older version of the ontology is regenerated and visualized.

We use a running example featuring a *Bibliography* ontology to show the process of logging the changes in CHL. Consider two changes as a part of one *ChangeSet*. First change adds a new *Author* class in the ontology and the second change sets *Author* as *range* of the property *hasAuthor*. Firstly, the process of creating entries in the log is explained. Taking into account the first change, an individual of *ClassAddition* is instantiated. The *isPartOf* property of this change instance is set to the active *ChangeSet*. Secondly, the *hasTimeStamp* value of the atomic change is also recorded for time-indexing of change entries. For logging the range addition entry, an individual of *RangeAddition* class from change history ontology is created and the value for its *hasChangedTarget* predicate is set to the *object property* for which the range has changed. The modified range information is then stored as a value of *hasTargetRange* property. Like all log entries, the *isPartOf* property of the individual is set to the active *ChangeSet* and its *hasTimeStamp* value is also stored with the individual. The code snippet given in Fig.11 represents these changes in RDF/N3 format.

```
log:ChangeSet192
    rdfs:type           ch:ChangeSet ;
    ch:hasChangeSetType ch:Interval
    ch:hasChangeAuthor  log:ChangeAgent2 ;
    ch:startTime        "2010-01-02 16:32:58+1" ;
    ch:endTime          "2010-01-02 16:53:11+1" ;
    ch:hasChangeReason  "Changes after applying rigidity meta property." ;
    ch:targetOntology   http://seecs.nust.edu.pk/vocab/bib .

log:ChangeAgent192
    rdfs:type           ch:ChangeAgent, foaf:Person ;
    foaf:name           "Administrator" .

log:IntervalChangeSet2457
    rdfs:type           ch:ChangeSet ;
    ch:hasChangeAuthor  log:ChangeAgent192 ;
    ch:startTime        00:00:46 ;
    ch:endTime          00:03:21 ;
    ch:hasChangeReason  "User Request" ;
    ch:targetOntology   http://seecs.nust.edu.pk/vocab/bib .

log:ClassAddition245701
    rdfs:type           ch:ClassAddition ;
    ch:hasChangedTarget bib:Author ;
    ch:hasTimeStamp     1224702057078 ;
    ch:isPartOf         log:ChangeSet192 .

log:RangeAddition245701
    rdfs:type           ch:RangeAddition ;
    ch:hasChangedTarget bib:hasAuthor ;
    ch:hasPropertyType  owl:ObjectProperty ;
    ch:hasTargetRange   bib:Author ;
    ch:hasTimeStamp     1224702072640 ;
    ch:isPartOf         log:ChangeSet192 .
```

Fig. 11. Example of changes in *Bibliography ontology* represented using change history ontology constructs. The first change is adding a new class *Author* and the second change is about adding *Author* as the range of the property *hasAuthor*

5. Ontology Recovery and Traceability

In this section, we explain the change recovery process to revert the logged changes. The ontology recovery process is responsible for executing the changes on the ontology model in inverse manner and reverse chronological order. Changes are first converted to their inverse equivalents. For example (see Fig.11), *ClassAddition* instances are translated to *ClassDeletion*, being its inverse equivalent. Similarly, *RangeAddition* instances are converted to *RangeDeletion*. These inverse changes are then applied to the ontology in reverse temporal order of their occurrences. The final outcome is the previous state of the ontology.

For the reason of reverting changes in order to recover ontology, the parser's job is to parse the *CHL* for all the *ChangeSets* (in descending order of their timestamp value) corresponding to the currently loaded model. This descending order is also followed for implementing the inverse changes to revert the ontology to its previous state. The detailed procedure is presented in the recovery *Rollback*/*Undo* Algorithm. This algorithm reads and processes the member entries c_Δ, instances of *OntologyChange*, in the given *ChangeSet*, $\mathcal{S}_\Delta$ i.e., $c_\Delta \in \mathcal{S}_\Delta$.

The *RollBack* algorithm supports roll back operations in order to recover the older state of the ontology.

(Rollback or Undo Algorithm) This algorithm assumes a pre-defined function, **TimeIndexedSort** for sorting member entries of the *ChangeSet* based on their timestamp.
Input: An ontology $\mathcal{O}$.
Input: An instance of *ChangeSet*, $\mathcal{S}_\Delta \in$ *ChangeSet*, which lists the changes made in the ontology $\mathcal{O}$.
Output: The previous version $\mathcal{O}'$ of the ontology $\mathcal{O}$ after reverting the changes mentioned in $\mathcal{S}_\Delta$.
1. /* Sort member entries of the *ChangeSet* in descending order of their timestamp*/
2. TimeIndexedSort($\mathcal{S}_\Delta$, 'DESC')
3. **foreach** $c_\Delta \in \mathcal{S}_\Delta$ **do**
4. /* Process resource addition */
5. **if** c_Δ : *OntologyChange* $\sqcap \exists$*changeType*.*Create* **then**
6. /* Remove the resource(s) which were target of the change */
7. $\mathcal{O} \leftarrow \mathcal{O} - \{x|\langle c_\Delta, x\rangle$*changeTarget*$\}$
8. **else**
9. /* Process resource deletion */
10. **if** c_Δ : *OntologyChange* $\sqcap \exists$*changeType*.*Delete* **then**
11. $\mathcal{O} \leftarrow \mathcal{O} + \{x|\langle c_\Delta, x\rangle$*changeTarget*$\}$
12. **else**
13. /* Process modification*/
14. ...
15. /* Implementation of this algorithm consists of a number of other conditional statements to check the change type and to process it accordingly, such as for annotations.*/
16. **endif**
17. **end**

The algorithm for *RollForward* is given below which transforms the ontology to its next state. This algorithm reads and processes the member entries c_Δ, instances of *OntologyChange*, in the given *ChangeSet*, S_Δ i.e., $c_\Delta \in S_\Delta$.

Consider two logged changes to the *Bibliography* ontology as described in Fig. 11. The first change is adding a new class *Author* and the second change is about adding *Author* as the range of the property *hasAuthor*. In order to revert these changes, the *ChangeSet* entries have to be processed in reverse order of their *hasTimeStamp* values.

During the recovery phase, as described in *RollBack* Algorithm, all changes belonging to a *ChangeSet* are sorted in a reverse chronological order and are implemented in the inverse manner on the currently loaded model. For instance, the *RangeAddition* change is first transformed into a property *RangeDeletion* request and then implemented. The formal representation of *RangeDeletion* (inverse of *RangeAddition*) is given in Fig. 12:

```
log:RangeDeletion245701
rdfs:type             ch:RangeDeletion;
ch:hasChangedTarget   bib:hasAuthor;
ch:hasTargetRange     bib:Author;
ch:hasPropertyType    owl:ObjectProperty;
ch:hasTimeStamp       1224702072649;
ch:isPartOf           log:ChangeSet2481 .
```

Fig. 12. Reverted (inverse of) range addition change.

Similarly, to delete a *ClassAddition* entry, the target class is selected and removed from the loaded model. Any further changes will be reverted in the same manner and the updated ontology model will be preserved along with the revision history. Algorithms for reverting different type of changes are given subsequently.

The *Rollback* and *Rollforward* algorithms are designed to revert the ontology to the previous or next state respectively based on a user request. Both algorithms require the logged ontology changes for the recovery process. For *Rollback* (or undo operation) the changes are retrieved from the history log and are processed

(Range Deletion) This algorithm unsets the range of a property P from class C.
Input: An ontology $\mathcal{O}$, instance of ontology change that belong to *ChangeSet*, i.e., $\mathcal{C}_\Delta \in \mathcal{S}_\Delta$, made in ontology $\mathcal{O}$.
Output: The previous state $\mathcal{O}'$ of the ontology $\mathcal{O}$ after reverting the change of *RangeAddition* as *RangeDeletion* mentioned by $\mathcal{C}_\Delta$.
1. **if** C_Δ : *OntologyChange* $\sqcap \exists$*changeType*.*RangeAddition* **then**
2. /* Unset the range of P from C and update the ontology */
3. $\mathcal{O} \leftarrow \mathcal{O} - \{x | \langle C_\Delta, x \rangle$ *changeTarget*$\}$
4. **endif**
5. **end**

(Class Deletion) This algorithm delete the class C from ontology.
Input: An ontology $\mathcal{O}$.
Input: An instance of ontology change that belong to *ChangeSet*, i.e., $\mathcal{C}_\Delta \in \mathcal{S}_\Delta$, made in ontology $\mathcal{O}$.
Output: The previous state $\mathcal{O}'$ of the ontology $\mathcal{O}$ after reverting the change of *ClassAddition* as *ClassDeletion* mentioned by $\mathcal{C}_\Delta$.
1. **if** C_Δ : *OntologyChange* $\sqcap \exists$*changeType*.*ClassAddition* **then**
2. /* Delete the class C from the ontology and update the ontology */
3. $\mathcal{O} \leftarrow \mathcal{O} - \{x | \langle C_\Delta, x \rangle$ *changeTarget*$\}$
4. **endif**
5. **end**

in reverse chronological order and inverse of these changes are implemented in the ontology. On the flip side, for redo operations, the required changes are retrieved from the history log and are implemented in the ontology in chronological order of their timestamp values (c.f. *RollForward* Algorithm). In the later scenario, reverse changes are not generated. The recovery algorithms work at the finest granularity level of atomic change requests.

6. Implementation Details

We envisioned our proposed framework as an enabling component for the ontology editors. The framework itself doesnt provide ontology editing services, rather it implements listeners (specific to an ontology editor) to monitor and log changes. The framework was implemented for ontologies defined in *rdfs* and all variants of *OWL*. Various components were implemented in the framework to perform tasks related to change history management. For example, the Change Logger component, preserves the changes, and the recovery component, on top of all other components, provides ontology recovery services [21]. The component based framework architecture is given in Fig. 13, whereas a detailed description of these components is given in [19]. To validate the working of the proposed framework, we have also developed a *TabWidget* plug-in, *ChangeTracer* Tab, for Protege ontology editor. Details of various modules are given below:

6.1. *Change Listener*

The Change listener module consists of multiple listeners which actively monitor various types of changes applied to the ontology model in Protégé. Table 1 presents all the listeners that we have implemented using the Protégé OWL API. *ProjectListener* listens for project related changes. One of its main functions is to listen for the save or close commands to save the active *ChangeSet* instance in CHL. *KnolwedgeBaseListener* is the most used listener for capturing the changes which are also triggered by the other listeners. This listener overlaps with *ClsListener, ClassListener, SlotListener, and PropertyListener. FacetListener* also overlaps with the *ClsListener, ClassListener, SlotListener, and PropertyListener*; however, it also provides additional axiom related change information. The *InstanceListener* overlaps with the *KnolwedgeBaseListener*

(RollForward or Redo Algorithm) This algorithm assumes a pre-defined function, **TimeIndexedSort** for sorting member entries of the *ChangeSet* based on their timestamp.
Input: An ontology $\mathcal{O}$.
Input: An instance of *ChangeSet*, $\mathcal{S}_\Delta \in$ *ChangeSet*, which lists the changes made in the ontology $\mathcal{O}$.
Output: The next version $\mathcal{O}'$ of the ontology $\mathcal{O}$ after re-implementing the extracted changes from CHL mentioned in $\mathcal{S}_\Delta$.
1. /* Sort member entries of the change set in ascending order of their timestamp and select the most recent one*/
2. TimeIndexedSort($\mathcal{S}_\Delta$, 'ASC')
3. **foreach** $c_\Delta \in \mathcal{S}_\Delta$ **do**
4. /* Process resource addition */
5. **if** c_Δ : *OntologyChange* $\sqcap \exists$*changeType*.*Create* **then**
6. /* Re-implement (insert) the resource(s) which were target of the change */
7. $\mathcal{O} \leftarrow \mathcal{O} + \{x|\langle c_\Delta, x\rangle$*changeTarget*$\}$
8. **else**
9. /* Process resource deletion */
10. **if** c_Δ : *OntologyChange* $\sqcap \exists$*changeType*.*Delete* **then**
12. /* Re-implement (delete) the resource(s) which were target of the change */
11. $\mathcal{O} \leftarrow \mathcal{O} - \{x|\langle c_\Delta, x\rangle$*changeTarget*$\}$
12. **else**
13. /* Process modification*/
14. ...
15. /* Implementation of this algorithm also consists of a number of other conditional statements to check the change type and to process it accordingly.*/
16. **endif**
17. **end**

for capturing the instance level changes. When a change is committed, its corresponding listener collects the necessary contextual information, such as change agent, target, and updated value.

6.2. *Change Logger*

The changes captured by the listeners are logged with conformance to *CHO*. All the changes are handled at the atomic level. This aspect covers both atomic changes, such as deleting a single concept, as well as complex scenarios e.g., deleting a sub-tree involving multiple concepts. Atomic change are easy to handle. In contrast, compound changes are sometimes harder to implement. For instance, deleting a *ChangeAgent* class will also impact its subclasses. As a result, every change requests has to be handled at atomic level. Logging component ultimately plays a critical role in maintaining atomicity of changes, and undo or redo operations, in case of a failure. Logging component uses *CHO* specifications for persistent storage of changes. Fig. 14 shows the details of change history. The tabular view of different *ChangeSet* instances are given and on the selection of a particular *ChangeSet* instance from the grid, its relevant information is displayed in the below panel. The dropdown-box contains the details of all the changes corresponding to the selected *ChangeSet* and its color represent the nature of that change. When a change element is selected from the dropdown-box, its corresponding change meta information is displayed. For instance: in Fig. 14, *hasReceivedDate* property change is selected which is of type addition and the corresponding details follow.

6.3. *Change History Log Implementation*

CHL is a repository that keeps track of all the changes made to ontology. It is also required for reversibility purpose when an ontology engineer wants to undo or redo some of the changes. The log uses Jena based

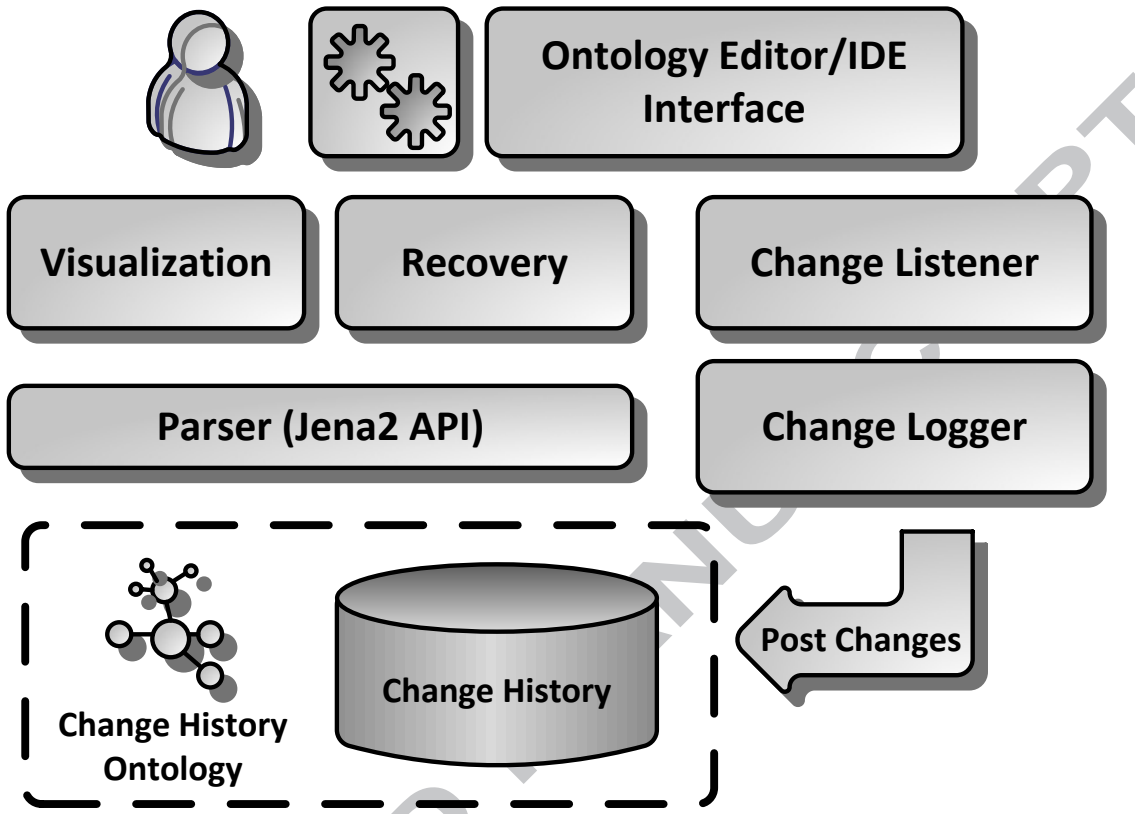

Fig. 13. Component based framework architecture of the proposed system. The architecture is published in [19].

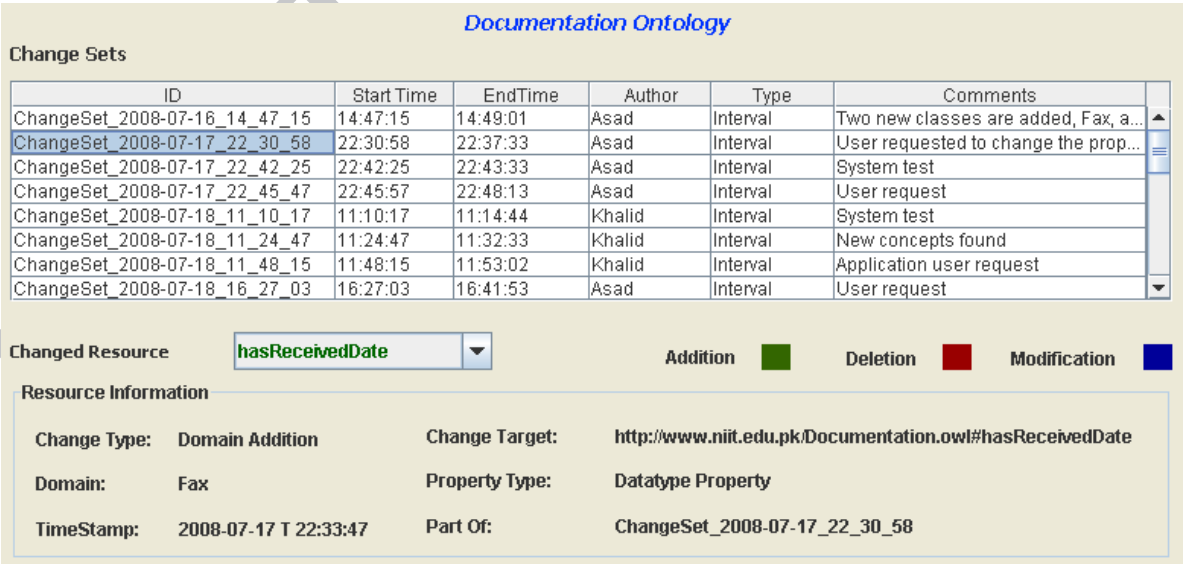

Fig. 14. Tabular view of changeSets and atomic changes in the ontology.

Change Listener	Description
ProjectListener	It listens all the project related events: like saving, closing, form changed, and runtime class widget created.
KnowledgeBaseListener	Helps in listening changes related to the model. It overlaps in its provided methods with all the listeners listed below.
ClsListener	Helps in capturing the class, sub-class, and super-class level changes.
ClassListener	Similar to ClsListener, it helps in capturing the class, sub-class, and super-class level changes.
SlotListener	Helps in capturing the slot, sub-slot, and super-slot level changes.
PropertyListener	Helps in capturing the class property, sub-property, and super-property level changes.
FacetListener	It helps in capturing the changes, such as restrictions, on frames.
InstanceListener	It helps in capturing changes related instances and individuals.

Table 1
List of change listeners implemented in the Change Tracer plug-in to listen and log ontology changes.

triple store and change description is provided by *CHO*, to preserve changes for later use. The details of CHL applications are given in Section 4.

6.4. *Parser*

The job of Parser is to: 1) Parse *CHL* for all the *ChangeSet*(*s*) that correspond(s) to the open model in Protege on user request. 2) produce the inverse changes of the stored ones to recover the pervious state of ontology, then all the *RangeAddition* instances will be converted *RangeDeletion* as shown in Fig. 11 and Fig. 12. The sequence of applying the changes back is also in backward order, i.e., changes in a *ChangeSet* applied at the end will be reverted first, then second last changes and so on. These inverse changes are given to the reverser module which implements these changes in reverse order. Fig. 15 shows the SPARQL queries for parsing the ontology changes form *CHL*. These queries are executed on the CHL by the *Parser* module to extract all the changes corresponding to a specific *ChangeSet* instance. The first query extracts the *ChangeSet* instances in time order, and then based on user needs, appropriate *ChangeSet* instance is selected and its corresponding changes are extracted from the *CHL*. Afterwards the details of these changes are extracted. The queries are important for recovery purpose, including both *rollback* and *rollforward*.

6.5. *Recovery*

The Recovery module is responsible for implementing the applied changes on models opened in Protege, in forward and reverse manner, based on user request. This module gets activated when a user requests to undo/redo any changes or requests for recovering the previous consistent state of ontology. For any of the above requests, this module makes request to parser module to retrieve the required *ChangeSet* entry and all its corresponding changes, which returns the changes of the corresponding logged *ChangeSet* in reverse order, the recovery module then implements these changes on the opened model. The demonstration for Recovery and Visualization of the system is presented in [23].

```
SELECT  ?changes  ?timeStamp
WHERE { ?changes docLog:isPartOf changeSetInstance .
?changes docLog:hasTimeStamp ?timeStamp }
ORDER BY DESC(?timeStamp)

SELECT  ?changedTarget ?isSubClassOf
WHERE { Resource docLog:hasChangedTarget ?changedTarget .
Resource docLog:isSubClassOf  ?isSubClassOf }

SELECT ?change ?changedTarget ?isSubClassOf ?isSubPtyOf ?hasPtyType
?oldName ?changedName ?hasDomain ?hasRange  . . . . .?timeStamp
WHERE { ?change   docLog:isPartOf changeSetInstance .
OPTIONAL {?change docLog:hasChangedTarget ?changedTarget} .
OPTIONAL {?change docLog:isSubClassOf ?isSubClassOf} .
OPTIONAL {?change docLog:isSubPropertyOf ?isSubPtyOf} .
OPTIONAL {?change docLog:hasPropertyType ?hasPtyType} .
OPTIONAL {?change docLog:hasOldName ?oldName} .
OPTIONAL {?change docLog:hasChangedName ?changedName} .
OPTIONAL {?change docLog:hasDomain ?hasDomain} .
OPTIONAL {?change docLog:hasRange ?hasRange} .
                          .
                          .
?change   docLog:hasTimeStamp ?timeStamp }
ORDER BY DESC(?timeStamp)
```

Fig. 15. SPARQL query for extracting changes corresponding to the ChangeSet and then extracting their relevant details.

6.6. *Visualization*

The Visualization module is responsible for visualizing the ontology, ontology changes, and their effect on ontology. The visualization is in graph like structure rather then tree like structure, because the ontology with class and sub-class hierarchy can also have associative relationships with other classes [26]. Fig.16 is an interface for visualizing ontology and visually navigating through its different states. Ontology components (such as concepts and their relationships) as well as the changes made in the ontology are visualized. Effects of these changes, for example, how it evolved to the current state, are emitted by navigating through the life of ontology. In order to visualize changes, the ontology change parser processes the requested *ChangeSets* and their corresponding changes. The changes are reverted or implemented on the ontology with the help of recovery module to take ontology to a previous or next state. We have extended the *TouchGraph* API for graph drawing in order to visualize the graph view of the ontology structure. Resources, such as classes, are depicted as nodes. These nodes are connected through properties, which are depicted as edges. The direction of an edge depicts the direction of the relationship among the nodes. Number of filters are supported in the graph view, such as zooming in and out of the graph and fish-eye view. A modified version of the *Spring* graph drawing algorithm [32] is implemented in the visualization that ensures aesthetically good looking graph structure and well separated nodes. We have provided the playback and play-forward features where not only the ontology but the changes can also be navigated. The visual navigation of changes and change effects on ontology helps in analyzing the trends (Fig. 16). Starting from the very first version of the ontology, the user can play the ontology changes and their effects on ontology and resources. The changing concepts are highlighted and color coded to reflect the changes. For example, the deleted concepts fade out and the new additions gradually appear in the graph. This improves understanding of the evolution history of ontology.

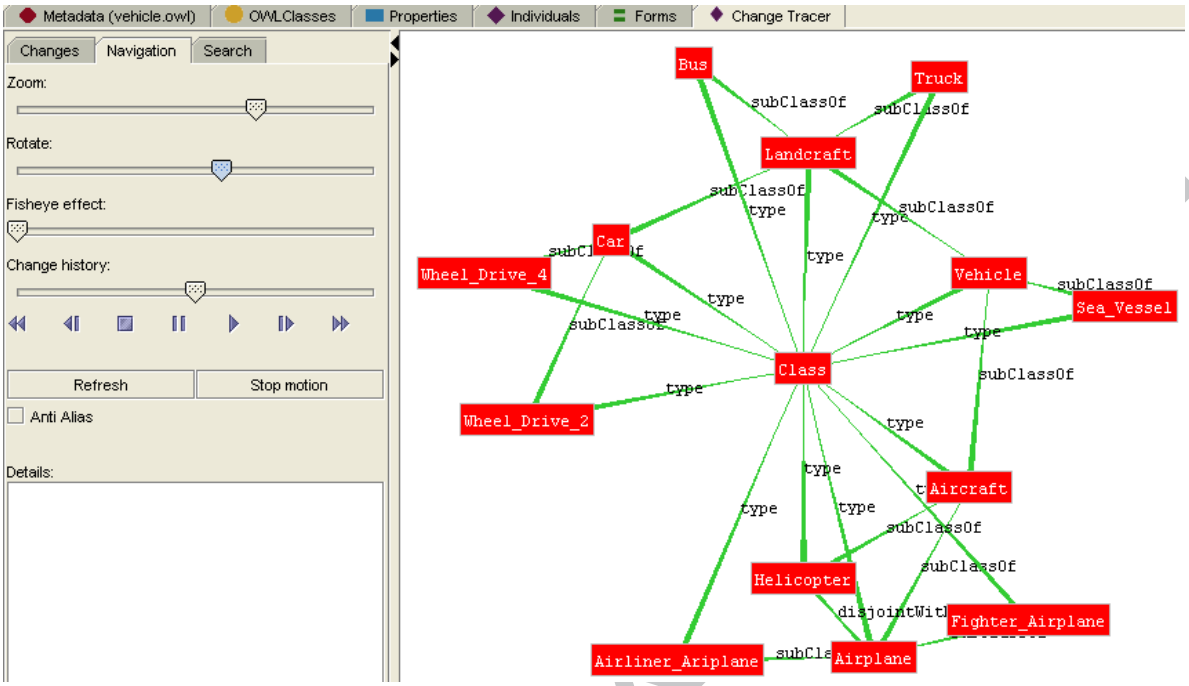

Fig. 16. Graph visualization of ontology with change history playback feature. Users can visually navigate through the ontology changes.

7. Results and Evaluation

The *Bibliography* ontology is used for the development and testing of the plug-in. First of all, the results of the plug-in for change capturing are provided using the *Bibliography* ontology. At the end, an exhaustive performance testing of the plug-in on OMV ontology [16], SWRC ontology [44], CRM ontology [6] and the standard dataset of Semantic Web Technology Evaluation Ontology [1] is provided.

7.1. *Change Capturing*

There exists no such system (that claims all the features our plug-in provides) to compare the developed plug-in with. However, there exist systems, such as: a Protege plug-in *ChangesTab* [33], another Protege plug-in *VersionLogGenerator* [37], *ChangeDetection* [37], and *ChangeCapturing* [38] that do provide the change capturing facility. We have compared the proposed plug-in (*ChangeTracer*) with *ChangesTab*, *VersionLogGenerator*, *ChangeDetection*, and *ChangeCapturing* to analyze its change capturing capability. For this, 35 different changes covering all four different categories (i.e., Change in Hierarchy, Change in Class, Change in Property, and Other Changes) were made to the *Bibliography ontology*. (*ChangeTracer*), *ChangesTab*, and *VersionLogGenerator* were configured with Protege, *ChangeCapturing* with NeOn Toolkit (http://www.neon-toolkit.org/), and *ChangeDetection* was used as a stand alone application. Out of these 35 changes, *ChangesTab* captured 28 changes, *VersionLogGenerator* captured 28 changes, *ChangeDetection* captured 33 changes, *ChangeCapturing* captured 32 changes, whereas our proposed plug-in i.e., *ChangeTracer* captured 32 changes. The graph representing these results is given in Fig. 17, where the y-axis represents the number of changes captured and the x-axis represents the total number of changes made.

Protege internally implements different listeners (see Table 1) that report when a change occurs in the open ontology model. (*ChangeTracer*), *ChangesTab*, and *VersionLogGenerator* were developed to implement these listeners and capture the changes that were triggered by Protege. However, when certain events are triggered, such as element deletion (i.e., Class, Property, Individual), then the element is first deleted and later the event is notified. Due to this reason, the deleted element's information is missed and not captured

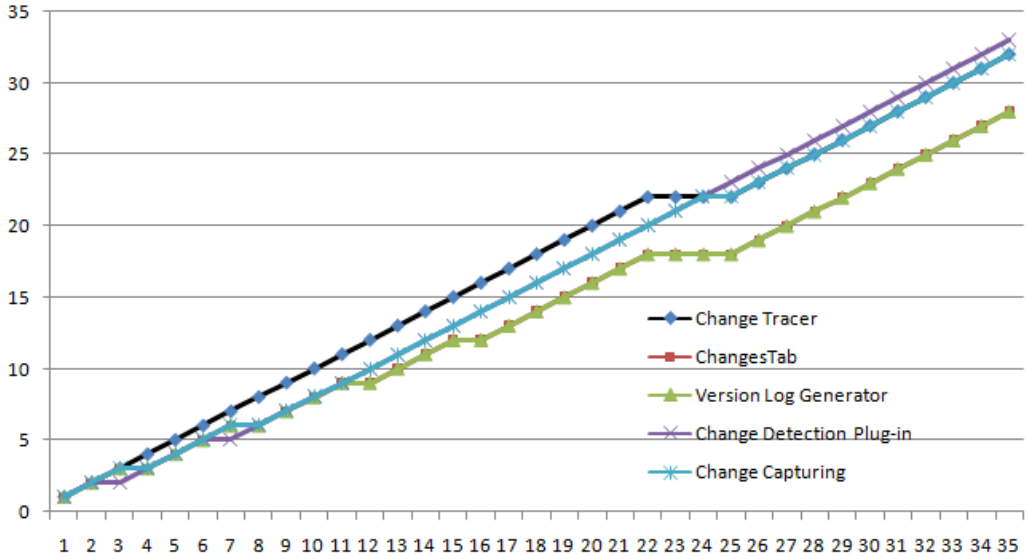

Fig. 17. Comparison of *ChangeTracer* against *ChangesTab*, *VersionLogGenerator*, *ChangeDetection*, and *ChangeCapturing*.

by the plug-ins (*ChangeTab* and *VersionLogGenerator*). This also happens with *ChangeCapturing*. In our plug-in, to handle this issue, difference is computed for the new model and the old model at the time of element deletion event, which provides the information about the deleted element. Another issue with Protege is that for Datatype property range addition, deletion, and modification, there is no event notification. So these changes are misidentified. As visible from the Fig. 17, that *ChangeDetection* captured more changes than the others; however, its performance heavily depends on the types of changes. For example, if the changes are always of element modification in an ontology then the *ChangeDetection* will misidentify them. Moreover, *ChangeDetection* cannot identify a sequence of changes.

To get more concrete results, we repeated the experiment for 20 times with 35 different and random changes. Details and results of these experiments are given in Table 2 and its graph representation is given in Fig. 18. Total of 700 different changes were made. Out of these 700 changes, 663 (i.e., 94.71%) changes were detected by *ChangeTracer*, 632 (i.e., 90.28%) changes by *ChangesTab*, 645 (i.e., 92.14%) changes by *VersionLogGenerator*, 629 (i.e., 89.86%) changes by *ChangesDetection*, and 667 (i.e., 95.28%) changes by *ChangesCapturing*. The results clearly show that our plug-in (i.e., *ChangeTracer*) has outperformed the *ChangesTab*, *VersionLogGenerator*, and *ChangeDetection* in terms of change capturing. In comparison with *ChangeCapturing*, our system has almost the same results. However, our plug-in uses the *difference()* method of *Model* class from *Jena API* to capture the missed changes, whereas the *ChangeCapturing* always misses certain changes [38].

7.2. *Change Recovery*

The aim of this discussion is to validate whether the proposed algorithm for ontology recovery is correct and can scale up to complex ontologies. Validation and verification of the outcome of the recovery process is essential and critical. There has to be a mechanism to prove the hypothesis that the output ontology, after applying the recovery process on top of the *CHO*, is correct. In order to quantitatively measure the performance of the recovery algorithm, an evaluation measure was used which is discussed below.

For the evaluation of the recovery procedure, we took two different versions of ontology i.e., O_{V1} and O_{V2}. The changes between the versions i.e., C_{Δ} were stored in Change History Log (*CHL*) using *CHO*. 35 different changes were manually incorporated in *Bibliography* ontology. All the changes were classified in three categories: (1) Hierarchy level changes, including the changes having effects on classes, properties, and

Specifications	Change Tracer	Changes Tab	Version Log Generator	Change Detection Plug-in	Change Capturing
No of Experiments	20	20	20	20	20
No of Changes per Experiments	35	35	35	35	35
Total Changes	20 * 35 = 700	20 * 35 = 700	20 * 35 = 700	20 * 35 = 700	20 * 35 = 700
Changes Captured	663	632	645	629	667
Average	94.71	90.28	92.14	89.86	95.28

Table 2
Comparative Analysis of Change Detection Approaches i.e., Changes Tab, Version Log Generator, Change Detection Plug-in, and Change Capturing Against Proposed Change Tracer.

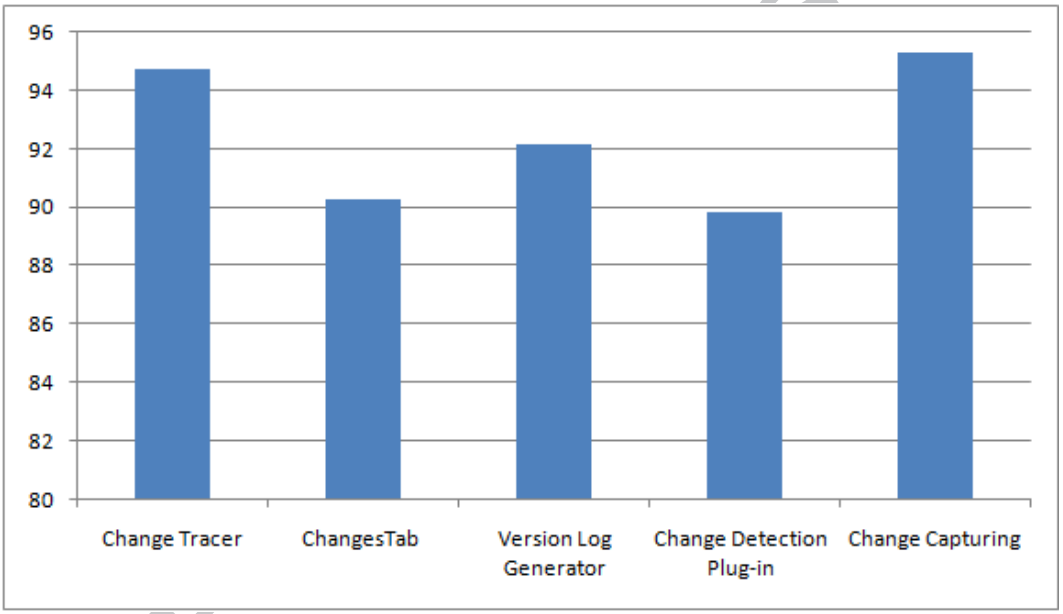

Fig. 18. Shows the average result of 20 experiments with 35 different changes using *ChangeTracer*, *ChangesTab*, *VersionLogGenerator*, *ChangeDetection*, and *ChangeCapturing*.

their constraints, (2) Class level changes, changes resulting from modifications to classes and constraints on classes. These changes also contribute to changes at hierarchy level, (3) Property level changes, changes resulting from modifications to properties and constraints on the properties. They also contribute to changes at hierarchy level. The number of hierarchy, class, and property changes was 10, 10, and 15 respectively. However, changes contributing from classes and properties made the number for hierarchy level changes bigger. After identifying and logging the changes between two versions, we came up with an equation for the verification of recovery procedure. As our plug-in provides both *Rollback* and *Rollforward* facilities, we separated equations for these procedures' verification.

7.2.1. *Roll Back*

To roll back the changes from O_{V2}, we simply need to subtract all the changes i.e., C_Δ from the ontology that caused O_{V2} from O_{V1}. This subtraction of the changes from O_{V2} was all made using proposed recovery (*RollBack*) algorithm. The equation for verification is as under;

$$\mathcal{O}_{Vx} \equiv \mathcal{O}_{V2} - \mathcal{C}_\Delta \tag{6}$$

Roll Back				
Details	**Tests**	**Correct Results**	**Problems**	**Accuracy**
Initial Attempts:	12	5	Domain Addition, Datatype Property Range Deletion	41.67
First Revision:	12	7	Inverse Property	58.34
Second Revision:	12	12	Nil	100
Roll Forward				
Details	**Tests**	**Correct Results**	**Problems**	**Accuracy**
Total Attempts:	36	36	Nil	100

Table 3
Roll Back and Roll Forward procedures results.

$$\mathit{difference}\langle \mathcal{O}_{V1}, \mathcal{O}_{Vx} \rangle \equiv \emptyset \tag{7}$$

The recovery (*RollBack* Algorithm) process was applied on O_{V2}. The recovered version was stored in another temporary version O_{Vx}. The temporary recovered version was checked against the available version O_{V1}. Here we differed O_{V1} from the recovered version i.e., O_{Vx} and if the difference was null (empty) then it means that the recovery process for roll back produced correct result.

7.2.2. *Roll Forward*

To roll forward the ontology from O_{V1}, we simply need to add/apply all the changes i.e., C_{Δ} to the ontology that caused O_{V2} from O_{V1}. This addition of the changes to O_{V1} were all made using proposed recovery (*RollForward*) algorithm. The equation for verification of *Rollforward* algorithm is;

$$\mathcal{O}_{Vx} \equiv \mathcal{O}_{V1} + \mathcal{C}_{\Delta} \tag{8}$$

$$\mathit{difference}\langle \mathcal{O}_{V2}, \mathcal{O}_{Vx} \rangle \equiv \emptyset \tag{9}$$

The recovery (*RollForward* Algorithm) process was applied on O_{V1}. The recovered version was stored in another temporary version O_{Vx}. The temporary recovered version was compared against the available version O_{V2}. Furthermore, we differed O_{V2} from the recovered version i.e., O_{Vx} and if the difference was found null (empty) then it means that the recovery process for roll forward produced correct result.

The difference between two ontology models was calculated using the *difference*() method of *Model* class from *JenaAPI*. We have also checked both these versions using Prompt [36]. Using the *Bibliography* ontology, we have tested the *RollBack* and *RollForward* algorithms and got very good results. The details of these results are given in Table 3, whereas their descriptions are given below.

For *RollBack*, we have tested the plug-in 12 times and obtained 5 correct results. The problems were: (1) when a *DomainAddition* entry is rolled backed, it is reverted as *DomainDeletion*. So the algorithm actually has deleted the domain of some property; however, Protege internally assigns *owl* : *Thing* as domain to all those properties which do not have any domain. (2) When datatype property range is deleted, the range of that property is not captured properly. Because of these two problems we got very low accuracy for *Rollback*. We have solved these problems and tested the plug-in 12 more times. This time, we obtained 7 correct results. Only one issue was found, i.e., when a property is made as inverse property, the information about the other property to which this property is made inverse to, is missing. We resolved the domain issue by letting the domain of a property as empty, range problem by using the *difference()* method, and inverse property problem by introducing the *hasInverseTo* property in CHO. After the corrections, 12 more experiments were conducted and this time we got 12 correct results and had no issues with the recovery procedure.

RollForward was implemented after we have completely implemented the *RollBack* and removed all the problems which we faced during *RollBack*. To justify that the system is correctly working for its *RollForward* operation, we tested the *RollForward* operation with all the 36 tests which were used to test the *RollBack* operation. Out of 36 roll forward experiments, we obtained 36 correct results with 100% accuracy, as shown in Table 3.

Ontology Versions	OMV.owl & OMV-0.7.owl	OMV-0.7.owl & OMV-0.91.owl	swrc-v0.3.owl & swrc-updated-v0.7.1.owl
Total Changes	38	189	310
Change in Hierarchy	18	71	131
Change in Classes	6	34	84
Change in Properties	25	123	172

Table 4
Roll Back and Roll Forward procedures' results.

To validate that the system was not only tested on biased and controlled data sets, we provided a detail system evaluation on four standard online available data sets. The details of all these experiments are given in the next section.

7.3. *System Evaluation*

In this section, detailed evaluation of the proposed recovery algorithms is presented. We have tested the algorithms with different versions of four different standard ontologies openly available. The reason for testing the system on four different data sets was to prove that the system is usable with varity of different ontologies and in uncontrolled environments. Another reason for the test was to cover as many aspects of ontology change as possible. As one can see in Table 4 and 5, in different ontologies the concentration of changes are different. For example, OMV and SWRC ontology have more changes from properties and axiom prospective, SWETO has more on the class perspective, whereas, the CRM has mixed changes.

It is important to mention the process of logging the changes between different versions of ontologies. To log the changes in CHL, *ChangeTracer* was configured with protege and then the respective changes were performed to the ontologies to log them in CHL. Details of these changes, their types, their dependence, their most appropriate sequence, and effects were manually analyzed. For the confirmation of the change analysis between two versions of ontology, the completeness of the changes, and the correctness of the changes, *difference()* method of *Model* class from *Jena API* as well as the *Compare* operation of *Prompt* [36] algorithm were used. These logged changes were used for the evaluation of the system which is given below. Because some changes fall into more than one category, the total number of changes is less than the sum of the specific categories of changes.

7.3.1. *Evaluation using OMV*

Ontology Metadata Vocabulary (OMV) (http://ontoware.org/frs/?group_id=39) [16] is used by the community for better understanding of the ontologies for the purpose of properly sharing and exchanging the information among organizations. To achieve this goal, this standard is set and agreed by the community for sharing and reusing of ontologies. OMV actually provides common set of terms and definitions describing ontologies, so called ontology metadata vocabulary. OMV have different versions available online containing different sets of concepts, properties, and restrictions. We have tested our developed plug-in on three different versions of OMV. The OMV versions that we have used for the experimentation are omv-0.6.owl, omv-0.7.owl, and omv-0.91.owl.

Table 4 shows complete details about the types and number of changes among different versions. These changes comprise class related, property related, and hierarchy related changes, which were captured and stored in *CHL* with the help of *ChangeTracer*. Using these logged changes, we applied the *Rollback* and *Rollforward* procedures with the validity checking using *Rollback* and *Rollforward* techniques presented in Section 7.2.1 and Section 7.2.2. We have compared all the recovered versions with the original version and they all were found error free.

7.3.2. *Evaluation using SWRC Ontology*

Semantic Web for Research Communities (SWRC) ontology [44], models key entities relevant for research communities and related concepts in the domain of research and development. Currently there are 70 different

concepts with 48 object type properties and 46 datatype properties. The reuse of ontologies for the real realization of semantic web and its continues improvement by user communities is a crucial aspect. The description and the usage guidelines are provided for SWRC ontology by the authors to make a complete value of the implicit and explicit facilities.

We have used two versions of SWRC ontology available online, which are swrc-v0.3.owl and swrc-updated-v0.7.1.owl. Changes between version swrc-v0.3.owl and swrc-updated-v0.7.1.owl comprise class related, property related, and hierarchy related changes, which were captured and stored in *CHL* with the help of *ChangeTracer*. Details are given in the Table 4. Using these logged changes we applied the *Rollback* and *Rollforward* techniques presented in Section 7.2.1 and Section 7.2.2, and that resulted in recovered versions. We have compared all the recovered versions with the original version and they all were found correct.

7.3.3. *Evaluation using CRM Ontology*

One of the ontologies we used for our experiments is the CIDOC Conceptual Reference Model (CRM) [6]. CRM provides a common language and semantic framework for experts and developers in the cultural heritage domain and facilitates in sharing the understanding of cultural heritage information. Multiple versions of CRM are available online . For the evaluation of our system, we used two versions of CRM ontology (i.e., cidoc-crm-3.2.rdf and cidoc-crm-3.4.rdf).

Changes between cidoc-crm-3.2.rdf and cidoc-crm-3.4.rdf were first captured and stored in *CHL* with the help of *ChangeTracer*. Details of all these changes are given in Table 5. Using these logged changes, we applied the *Rollback* and *Rollforward* techniques presented in Section 7.2.1 and Section 7.2.2. The *Rollback* from cidoc-crm-3.4.rdf with the help of logged changes produced a temporary recovered version. We checked the recovered version against cidoc-crm-3.2.rdf, both versions were found same. The *Rollforward* from cidoc-crm-3.2.rdf with the help of logged changes produced a temporary recovered version. We checked the recovered version against cidoc-crm-3.4.rdf, both versions were found the same with no differences between them. This shows that the *RollBack* and *RollForward* operations on cidoc-crm-3.2.rdf and cidoc-crm-3.4.rdf are correct.

7.3.4. *Evaluation using SWETO*

Semantic Web Technology Evaluation Ontology (SWETO) is basically an ontology developed as a benchmark by Semantic Web Community for evaluating the scalability of the available semantic web tools. More details on SWETO can be found on [1]. Since it is a benchmark ontology for testing the performance and scalability of semantic web tools, that's why we have adopted it to test the scalability and performance of the developed plug-in. Currently, we have used three versions of the SWETO ontology available in [1] with names: (i) testbed-v1-4.owl (ii) testbed-v1-3.owl, and (iii) testbed-v1-2.owl.

Since we had three versions of ontology, we conducted two experiments. Initially, we used the first two versions, i.e., testbed-v1-2.owl and testbed-v1-3.owl, and stored all the changes between these versions in their corresponding log file. The details of these changes are given in Table 4. Using these logged changes, we applied the *Rollback* and *Rollforward* techniques presented in Section 7.2.1 and Section 7.2.2. The *Rollback* and *Rollforward* operations on testbed-v1-3.owl and testbed-v1-2.owl were implemented in the same way as discussed in previous experiments. With this implementation and comparison, we observed correct results. This shows that the *Rollback* and *Rollforward* operations on the testbed-v1-2.owl and testbed-v1-3.owl are correct. Secondly, we used its two versions i.e., testbed-v1-3.owl and testbed-v1-4.owl, and stored all the changes between these two versions in CHL. The number and types of changes are given in Table 5. Using these logged changes we applied the *Rollback* and *RollForward* techniques presented in Section 7.2.1 and Section 7.2.2. The *Rollback* and *Rollforward* operations were applied in the same way as the previous steps. The differences of original versions and recovered versions from *RollBack* and *RollForward* were analyzed for verification. The results of all the differences were empty which shows that the *Rollback* and *Rollforward* operations on the testbed-v1-3.owl and testbed-v1-4.owl are correct.

Ontology Versions	cidoc-crm-3.2.rdf & cidoc-crm-3.4.rdf	testbed-v1-2.owl & testbed-v1-3.owl	testbed-v1-3.owl & testbed-v1-4.owl
Total Changes	170	124	223
Change in Hierarchy	94	60	170
Change in Classes	54	107	193
Change in Properties	103	14	22

Table 5
Roll Back and Roll Forward procedures results.

8. Conclusion and Future Work

Ontologies are usually large, structured, and dynamic in nature. Changes in ontologies are expected to accommodate new knowledge about the domain of discourse. These changes are influenced by the uncontrolled, decentralized, and complex nature of the Semantic Web. This makes ontology change management a complex task. In this research, we explain Change History Ontology in detail as the backbone conceptual model of the change management framework. It acts as a glue to bind different components in the framework and guarantee effective recovery and proper communication between the components. This ontology is used to record changes by creating a semantically structured change history log. The system has initially been developed as a plug-in for the ontology editor Protege to listen and log all the ontology changes. The framework was later revamped to be an integral part of ontology repositories. Logged changes were used for ontology recovery, roll back and roll forward functions, implemented in the framework. The log of ontology changes was also used to visualize changes and their effect during different states of the evolving ontology. A play back feature is provided to navigate through the history of ontology for better understanding of the evolution behavior.

The change capturing ability of the developed framework is compared with the *ChangesTab*, *VersionLogGenerator*, *ChangeDetection*, and *ChangeCapturing*. The results have demonstrated that our framework has higher accuracy and better coverage. The recovery algorithms (including both *Rollback* and *Rollforward*) were also tested exhaustively for accuracy, and were found to be consistent and accurate. The tests were carried-out using different versions of four benchmark datasets; OMV, SWRC, SWETO, and CIDOC CRM.

Building a flexible and customizable change management framework is only the first step of the ladder. It can lead to many interesting research endeavors. For example, such a framework can be extended to change prediction by applying machine learning techniques. Already captured changes can also be analyzed to identify patterns in the ontology evolution process. Query reformulation on evolving knowledgebases (ontologies) is an important issue and is currently in pipeline based on CHL. The ontology changes captured in our framework can help in reformulating queries on evolved ontologies.

9. Acknowledgment

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 2011-0030823).

References

[1] I. B. Arpinar, Web technology evaluation ontology (sweto), a nsf medium itr project, http://lsdis.cs.uga.edu/projects/semdis/sweto/.

[2] W. Behrendt, E. Gahleitner, K. Latif, A. Gruber, E. Weippl, S. Schaffert, H. Kargl, Upper ontologies with specific consideration of dolce, sumo and sowas upper level ontology, Deliverable D121, DynamOnt Project (2005).

[3] S. Castano, A. Ferrara, G. Hess, Discovery-driven ontology evolution, in: 3rd Italian Semantic Web Workshop: The Semantic Web Applications and Perspectives (SWAP), Italy, 2006.

[4] S. Castano, A. Ferrara, S. Montanelli, Evolving open and independent ontologies, Journal of Metadata, Semantics and Ontologies (IJMSO) 1 (4).

[5] S. Castano, A. Ferrara, S. Montanelli, Matching ontologies in open networked systems, Techniques and applications, Journal on Data Semantics (JoDS) 3870/2006.
[6] M. Doerr, Chair, Heraklion, Crm, cidoc documentation standards working group, http://cidoc.ics.forth.gr/index.html.
[7] R. Elmasri, S. B. Navathe, Fundamentals of Database Systems, 4th ed., Addison Wesley, 2003.
[8] G. Flouris, On belief change in ontology evolution: Thesis.
URL http://dl.acm.org/citation.cfm?id=1219755.1219763
[9] G. Flouris, D. Plexousakis, , G. Antoniou, A classification of ontology changes, in: 3rd Italian Semantic Web Workshop: Semantic Web Applications and Perspectives (SWAP) – Poster Session, Italy, 2006.
[10] G. Flouris, D. Plexousakis, Handling ontology change:survey and proposal for a future research direction, Technical Report TR-362 FORTH-ICS, Institute of Computer Science, FORTH., Greece (2005).
[11] H. K.-D. P. G. Flouris, D. Manakanatas, G. Antoniou, Ontology change: Classification and survey, Knowledge Engineering Review (KER) 23 (2).
[12] E. Gahleitner, K. Latif, A. Gruber, R. Westenthaler, Specification of methodology and workbench for dynamic ontology creation, Deliverable D201, DynamOnt Project (2006).
[13] A. Gangemi, Ontology design patterns for semantic web content, in: Y. Gil, E. Motta, R. Benjamins, M. Musen (eds.), 4th Intl Semantic Web Conf (ISWC), vol. 3729, Springer, Ireland, 2005.
[14] T. R. Gurber, A translation approach to portable ontologies, Knowledge Acquisition 5 (2).
[15] P. Haase, Y. Sure, State of the art on ontology evolution, Technical Report D3.1.1.b, SEKT Project: Semantically Enabled Knowledge Technologies (August 2004).
[16] Y. S.-P. H. M. C. S. F. J. Hartmann, R. Palma, Omv - ontology metadata vocabulary, in: in: C. Welty (Ed.), ISWC 2005 - In Ontology Patterns for the Semantic Web, Galway, Ireland, 2005.
[17] D. Jones, T. Bench-Capon, P. Visser, Methodologies for ontology development, in: J. Cuena (ed.), IFIP XV IT & KNOWS, Hungary, 1998.
[18] S. Khan, P. Mott, Differential evaluation of continual queries, Technical Report 2001.11, School of Computing, the University of Leeds (May 2001).
[19] A. M. Khattak, K. Latif, M. Han, S. Lee, Y.-K. Lee, H. I. Kim, Change tracer: Tracking changes in web ontologies, in: 21st IEEE International Conference on Tools with Artificial Intelligence, USA, 2009.
[20] A. M. Khattak, K. Latif, S. Khan, N. Ahmed, Managing change history in web ontologies, in: Fourth International Conference on Semantics, Knowledge and Grid, China, 2008.
[21] A. M. Khattak, K. Latif, S. Khan, N. Ahmed, Ontology recovery and visualization, in: 4th International Conference on Next Generation Web Services Practices, Korea, 2008.
[22] A. M. Khattak, K. Latif, S. Lee, Y.-K. Lee, T. Rasheed, Building an integrated framework for ontology evolution management, in: 12th International Conference on International Business Information Management Association, Malaysia, 2009.
[23] A. M. Khattak, K. Latif, Z. Pervez, I. Fatima, S. Lee, Y.-K. Lee, Change tracer: A protégé plug-in for ontology recovery and visualization, in: In proceedings of 13th APWeb, China, 2011.
[24] A. M. Khattak, Z. Pervez, K. Latif, S. Lee, Time efficient reconciliation of mappings in dynamic web ontologies, Knowledge-Based Systems (In Press) 0 (0).
[25] A. M. Khattak, Z. Pervez, K. Latif, A. J. Sarkar, S. Lee, Y.-K. Lee, Reconciliation of ontology mappings to support robust service interoperability, in: In proceedings of 13th APWeb, USA, 2011.
[26] M. Klein, Change management for distributed ontologies, Phd thesis, Vrije University, Netherlands (2004).
[27] M. Klein, A. Kiryakov, D. Ognyanov, D. Fensel, Finding and characterizing changes in ontologies, in: 21st Intl Conf on Conceptual Modeling, Finland, 2002.
[28] M. Klein, N. Noy, A component-based framework for ontology evolution, in: IJCAI Workshop on Ontologies and Distributed Systems, Germany, 2003.
[29] S. Lee, Y.-K. Lee, A. M. Khattak, H. I. Kim, M. Han, Method for reconciling mappings in dynamic/evolving web-ontologies using change history ontology, international Patent No. 12/576,342, Oct. 9, 2009.
[30] Y. D. Liang, Enabling active ontology change management within semantic web-based applications, Mini phd thesis, University of Southampton (2006).
[31] Y. D. Liang, H. Alani, N. Shadbolt, Ontology change management in protégé, in: AKT DTA Colloquium, Milton Keynes, United Kingdom, 2005.
[32] C. Lin, H. Yen, A new force-directed graph drawing method based on edge-edge repulsion, IEEE Computer Society 1 (1).
[33] W. Liu, T. Tudorache, T. Redmond, Changes tab in protégé, http://protegewiki.stanford.edu/index.php/Changes_Tab.
[34] N. Noy, M. Klein, Ontology evolution: Not the same as schema evolution, Knowledge and Information System 6 (4).
[35] N. Noy, S. Kunnatur, M. Klein, M. Musen, Tracking changes during ontology evolution, in: Intl Semantic Web Conf, USA, 2002.
[36] N. Noy, M. A. Musen, Prompt: Algorithm and tool for automated ontology merging and alignment, in: Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence, 2000.
[37] O. D. P Plessers, S. Casteleyn, Understanding ontology evolution: A change detection approach, Web Semantics Science Services and Agents on the World Wide Web 5 (1).
[38] A. G.-P. R. Palma, O. Corcho, P. Haase, A holistic approach to collaborative ontology development based on change management, Journal of Web Semantics 9 (3).

[39] D. Rogozan, G. Paquette, Managing ontology changes on the semantic web, in: IEEE/WIC/ACM Intl Conf on Web Intelligence, France, 2005.
[40] P. Shvaiko, J. Euzenat, Ten challenges for ontology matching, in: In Proceedings of The 7th International Conference on Ontologies, DataBases, and Applications of Semantics (ODBASE), Mexico, 2008.
[41] B. Smith, Blackwell Guide to the Philosophy of Computing and Information, chap. Ontology, Blackwell Philosophy Guides, Blackwell Publishing, 2003.
[42] Sparqlpush: pubsubhubbub (push) interface for sparql endpoint, Online, http://code.google.com/p/sparqlpush.
[43] L. Stojanovic, A. Madche, B. Motik, N. Stojanovic, User-driven ontology evolution management, in: European Conf on Knowledge Engineering and Management (EKAW), Spain, 2002.
[44] Y. Sure, S. Bloehdorn, P. Haase, J. Hartmann, D. Oberle, The swrc ontology - semantic web for research communities, in: In Proceedings of the 12th Portuguese Conference on Artificial Intelligence - Progress in Artificial Intelligence (EPIA), vol. 3803 of LNCS, Springer, Covilha, Portugal, 2005.
[45] S. Tunnicliffe, I. Davis, Changeset, Online, http://vocab.org/changeset/schema.html (2005).
[46] M. Tury, M. Bielikova, An approach to detection ontology changes, in: First international workshop on adaptation and evolution in web systems engineering (AEWSE), Brussel, 2006.
[47] M. Uschold, Building ontologies: Towards a unified methodology, in: 16th Annual Conference of the British Computer Society Specialist Group on Expert Systems, Cambridge, UK, 1996.