Operational Change Management and Change Pattern Identification for Ontology Evolution

Muhammad Javed

MSc. Electrical Engineering (FHD, Germany)

BSc. Electrical & Electronic Engineering (IIT, Bangladesh)

A dissertation submitted in fulfilment of the requirements for the award of degree

Doctor of Philosophy (Ph.D.)

to the

Dublin City University

Faculty of Engineering and Computing, School of Computing

Advisor: Dr. Claus Pahl

May 2013

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Doctor of Philosophy is entirely my own work, that I have exercised reasonable care to ensure that the work is original, and does not to the best of my knowledge breach any law of copyright, and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: [illegible]

(Candidate) ID No.: 58106383

Date: 02/05/2013

Examiners:

Dr. Mark Roantree

Head of Interoperable Systems Group (ISG),

Faculty Of Engineering & Computing,

School of Computing,

Dublin City University,

Ireland.

Dr. Sören Auer

Leader of Agile Knowledge Engineering and Semantic Web (AKSW),

Head of the Chair Information Systems & Software Technology,

Department of Computer Science,

Chemnitz University of Technology,

Germany.

Acknowledgment

First of all, I would like to thank God Almighty (Allah), for giving me the strength and courage to complete my research work. In last four years, I had a life-time experience of gaining knowledge and sharing few unforgettable moments with some wonderful people. I was lucky to have explored not only a new field of knowledge but also a new country, a new language and different cultures. Foremost, I am extremely thankful to my supervisor, Claus Pahl, whose support, guidance and encouragement from first to the final level enabled me to develop an understanding of the subject and without him this work would not have been possible. I am really thankful to my colleagues at Centre for Next Generation Localisation[1], who supported and encouraged me at each level of research. In my daily work, I had been blessed with a friendly and cheerful group of fellow colleagues. Yalemisew, I will really miss our conversations regarding research philosophies and most especially the Ethiopian coffee. To my colleagues Veronica, Aakash, Kosala, Pooyan and Wong in Software and System Engineering Group, I sincerely offer my regards and blessings to you who supported me in all aspects during my research work. I wish to acknowledge Science Foundation Ireland[2] for funding my research (under grant no. 07/CE/I1142). Whenever I look back at my past, I always felt myself as a blessed man. It would not have been possible for me to focus and work confidently without the continuous support of my wife Arshia and my children Yahya, Mustafa and Fatima. I believe, I couldn't give much time to you in past few years but I promise that I will try my best to recuperate and fill your life with joy. Last but not the least; I would like to thank my parents for their unconditional love, devotion and support towards my studies. Though I lived one third of my life away from you, but there was not a single day when I didn't remember you. Many times I used to get upset, but your prayers and love kept me going. May Allah give you the best of this life and hereafter.

[1]`www.cngl.ie`
[2]`www.sfi.ie`

To

My Family

Arshia, Yahya, Fatima and ***Mustafa***

Abstract:

Ontologies can support a variety of purposes, ranging from capturing the conceptual knowledge to the organization of digital content and information. However, information systems are always subject to change and ontology change management can pose challenges. In this sense, the application and representation of ontology changes in terms of higher-level change operations can describe more meaningful semantics behind the applied change. We propose a four phase process that covers the operationalization, representation and detection of higher-level changes in ontology evolution life cycle. We present different levels of change operators based on the granularity and domain-specificity of changes. The first layer is based on generic atomic level change operators, whereas the next two layers are user-defined (generic/domain-specific) change patterns. We introduce the layered change logs for an explicit and complete operational representation of ontology changes. The layered change log model has been used to achieve two purposes, i.e. recording of ontology changes and mining of implicit knowledge such as intent of change, change patterns etc. We formalize the change log using a graph-based approach. We introduce a technique to identify composite changes that not only assist in formulating ontology change log data in a more concise manner, but also help in realizing the semantics and intent behind any applied change. Furthermore, we discover the reusable ordered/unordered domain-specific change patterns. We describe the pattern mining algorithms and evaluate their performance.

Keywords: semantic ontology evolution, ontology change patterns, pattern-based ontology evolution, change log graph, graph-based composite change detection, change pattern discovery algorithms.

Contents

List of Figures

List of Tables

List of Publications

- [Javed et al., 2012a] Muhammad Javed, Yalemisew Mintesnote Abgaz and Claus Pahl: Composite Ontology Change Operators and their Customizable Evolution Strategies. ISWC Workshops: Joint Workshop on Knowledge Evolution and Ontology Dynamics (EvoDyn), Boston, USA, (2012).

- [Abgaz et al., 2012a] Yalemisew Mintesnote Abgaz, Muhammad Javed and Claus Pahl: Dependency Analysis in Ontology-driven Content-based Systems. 12th International Conference Artificial Intelligence and Soft Computing, Lecture Notes of Computer Science, Volume 7268, L. Rutkowski et al. (Eds.), pages 3–12, Springer-Verlag, (2012).

- [Abgaz et al., 2012b] Yalemisew Mintesnote Abgaz, Muhammad Javed and Claus Pahl: Analyzing Impacts of Change Operations in Evolving Ontologies. ISWC Workshops: Joint Workshop on Knowledge Evolution and Ontology Dynamics (EvoDyn), Boston, USA, (2012).

- [Javed et al., 2011a] Muhammad Javed, Yalemisew Mintesnote Abgaz and Claus Pahl: A Layered Framework for Pattern-Based Ontology Evolution. 3rd International Workshop on Ontology-Driven Information System Engineering (ODISE), London, UK, (2011).

- [Javed et al., 2011b] Muhammad Javed, Yalemisew Mintesnote Abgaz and Claus Pahl: Graph-based Discovery of Ontology Change Patterns. ISWC Workshops: Joint Workshop on Knowledge Evolution and Ontology Dynamics (EvoDyn), Bonn, Germany, (2011).

- [Abgaz et al., 2011] Yalemisew Mintesnote Abgaz, Muhammad Javed and Claus Pahl: A Framework for Change Impact Analysis of Ontology-driven Content-based Systems. On the Move to Meaningful Internet Systems: OTM 2011 Workshops. Volume 7046 of Lecture Notes in Computer Science, pages 402–411, Springer-Berlin/Heidelberg, (2011).

- [Javed et al., 2011c] Muhammad Javed, Yalemisew Mintesnote Abgaz and Claus Pahl: Towards Implicit Knowledge Discovery from Ontology Change Log Data. 5th International Conference on Knowledge Science, Engineering and Management (KSEM), Lecture Notes of Artificial Intelligence, Volume 7091, Xiong, Hui; Lee, W.B. (Eds.), pages 136–147, Springer-Verlag, (2011).

- [Javed et al., 2010] Muhammad Javed, Yalemisew Mintesnote Abgaz and Claus Pahl: Ontology-based Domain Modelling for Consistent Content Change Management. International Conference on Ontological and Semantic Engineering (ICOSE), Venice, Italy, (2010).

- [Abgaz et al., 2010] Yalemisew Mintesnote Abgaz, Muhammad Javed and Claus Pahl: Empirical Analysis of Impacts of Instance-Driven Changes in Ontologies. On the Move to Meaningful Internet Systems: OTM 2010 Workshops. Lecture Notes of Computer Science, Volume 6428, R. Meersman et. al. (Eds.), pages 368–377, Springer-Verlag, Springer-Berlin/Heidelberg, (2010).

- [Pahl et al., 2010] Claus Pahl, Muhammad Javed and Yalemisew Mintesnote Abgaz: Utilising Ontology-based Modelling for Learning Content Management. ED-MEDIA 2010-World Conference on Educational Multimedia, Hypermedia & Telecommunications, Toronto, Canada, (2010).

- [Javed et al., 2009] Muhammad Javed, Yalemisew Mintesnote Abgaz and Claus Pahl: A Pattern-Based Framework of Change Operators for Ontology Evolution. On the Move to Meaningful Internet Systems: OTM Workshops. Lecture Notes of Computer Science, Volume 5872, R. Meersman, P. Herrero, and T. Dillon (Eds.), pages 544-553, Springer-Verlag, (2009).

List of Abbreviations

ACL	:	Atomic Change Log
AG	:	Attributed Graph
ATG	:	Attribute Type Graph
DPO	:	Double PushOut
LCLM	:	Layered Change Log Model
OCBS	:	Ontology-driven Content based Systems
OCP	:	Ordered Complete Change Patterns
OCS	:	Ordered Complete Change Sequence
OP	:	Ordered Change Patterns
OPP	:	Ordered Partial Change Patterns
OPS	:	Ordered Partial Change Sequence
OS	:	Ordered Change Sequence
OWL	:	Web Ontology Language
PCL	:	Pattern Change Log
RDF	:	Resource Description Framework
RDFa	:	Resource Description Framework in attributes
SPARQL	:	Simple Protocol and RDF Query Language
SPO	:	Subject-Predicate-Object
UCP	:	Unordered Complete Change Patterns
UCS	:	Unordered Complete Change Sequence
UP	:	Unordered Change Patterns
UPP	:	Unordered Partial Change Patterns
UPS	:	Unordered Partial Change Sequence
US	:	Unordered Change Sequence

Glossary of Definitions

ABox - ABox statements in a domain ontology represent knowledge about instances of (TBox) classes.

Atomic change - An atomic change in an ontology adds or deletes one single axiom.

Attributed graphs - A basic graph is a set of nodes and edges. In attributed graph, one can attach a number of attributes to the nodes and edges of the graph.

Composite change - A composite change is a generic change containing a sequence of atomic ontology change operations.

Consistent atomic change - An atomic change is consistent if the existential conditions of its input parameters are satisfied. For example, in case of `Add subclassOf` (`Student`, `Person`), where the entities `Student` and `Person` must exist in the domain ontology as domain classes.

Dangling condition - The dangling conditions for an ontology graph transformation ensures the resultant graph has no dangling edges, i.e. without a source or target graph node.

DPO node - A double pushout node represent an entity from ontology graph (i.e. class, object property, data property or individual).

Graph node - The term “graph node” represent a node from the change log graph, representing an atomic ontology change operation.

N-distance - Node distance between two adjacent graph nodes n_1 and n_2 of an ontology change sequence s refers to the number of graph nodes exist between n_1 and n_2 in the change log graph.

Ontology axiom - Axioms are the coherent statements that can be made in a domain ontology representing certain specific knowledge.

RDF triple - An RDF triple contain three parts i.e. subject, predicate and object. The predicate part is also known as “property” as it represents a property of the subject; where, object is the actual value for such property.

Semantically identical change sequence - Two ontology change sequences are semantically identical to each other if the impact of their application to the domain ontology is same.

Structurally identical change sequence - Two ontology change sequences are structurally identical if the order of the existing change operations (in both sequences) is the same.

TBox - TBox statements in a domain ontology represent schema level conceptualization of classes and properties of these classes.

Chapter 1

Introduction

Ontology-driven modelling is beneficial for a wide range of content-based information systems. Ontology-based data models have helped researchers to take a step forward from traditional content management systems (CMS) to conceptual knowledge modelling to meet the requirements of the semantically aware information systems. Ontology-based approaches are used to capture architecture and process patterns [Gacitua-Decar et al., 2009]. Research as presented in [Filipowska et al., 2009] and [Hesse et al., 2008] stress the contribution of domain ontologies for content modelling. Domain ontologies can support tasks ranging from capturing conceptual knowledge to the organization of digital content and other information artefacts. Domain ontologies can convey useful semantic information for ontology engineers to understand and process.

As described by Gruber *"An ontology is a specification of a conceptualization"*, it is a specification of concepts in a particular domain and the relationships between these concepts through defined properties. In Web Ontology Language (OWL), concepts are regarded as *classes*. Domain ontologies represent the concrete classes of a domain and describe (in the form of properties) how these classes are linked to each other. The main aim of a domain ontology is to capture consensual knowledge of a given domain in a generic and formal way to be reused and shared across applications and groups of people [Gomez-Perez et al., 2006]. Classes in a domain ontology are categorized into more specific subclasses for explanation of specialized elements, creating a taxonomy architecture. Similarly, the properties, providing more explanation about a class, can also be subdivided into subproperties. Domain ontologies are developed to provide a semantic network sharing a common understanding of a concept amongst ontology engineers, domain experts, other users and software agents. An ontology engineer is responsible for developing and maintaining the domain ontologies. He provides semantic support, by mapping the domain ontologies with the various terminologies used in the domain. In contrast, a domain expert is a person with valuable functional knowledge and skills in a particular domain, often called *domain knowledge*.

The development of domain ontologies is a continuous process. As time passes, the domains evolve, requiring new classes to be added in the domain ontology and also relationships among newly added classes.

1.1 Motivation

Ontologies become essential for knowledge sharing activities in areas such as the semantic web, bio-informatics and educational technology systems. Such information systems are always subject to change and ontology change management can pose challenges. As there exists a dependency between the content and the domain ontologies, a change in the content may lead to a change in the domain ontologies. The reason for change in a domain ontology can be the change in the domain, the specification, the conceptualization or any combination of them [Noy et al., 2004]. Some of the changes are about the introduction of new classes, removal of outdated classes and change in the structures and the meanings of classes. A change in an ontology may initiate from a domain knowledge expert, a user of the ontology or a change in the application area [Liang et al., 2005]. This requires an effective ontology change management approach.

Ontology evolution is defined in different ways [Stojanovic et al., 2003, Haase et al., 2003, Flouris et al., 2006]. A comprehensive definition is given as *"the timely adaptation of an ontology to changed business requirements, to trends in ontology instances and patterns of usage of the ontology based application, as well as the consistent management/propagation of these changes to dependent elements"* [Stojanovic et al., 2002]. Based on the different perspectives of the researchers, there are different solutions provided to handle ontology evolution [Stojanovic et al., 2002, 2003, Zablith, 2008, Plessers et al., 2007]. Different phases of ontology evolution have been identified [Stojanovic et al., 2002]. These phases include *change capturing*, *change representation*, *semantics of change*, *change propagation*, *change implementation* and *change validation.* Basic changes in the evolving ontology can be captured using operators. However, the identified change operators focus on generic and structural changes lacking domain-specificity and abstraction. Moreover, these solutions lack adequate support for different levels of granularity at different levels of abstraction.

1.2 Problem definition

Content management systems have received a considerable attention due to appearance of different types of multilingual, multi-formatted content. Domain ontologies can be used to add the semantics and express the inter-linkages between the content elements. Domain ontologies capture and organize the different properties and perspectives on a content unit. The ontology-based semantic annotation is a key approach for adding semantics to content and their guided access. The currently developed models deal with semantic annotation. However, none of them deals with the change consistency issues, i.e. if there is a change in the content, it must be reflected in the respective domain ontology and vice versa. Consistency has to be established during content change management. We distinguish two categories of changes - changes to the content artefacts (content management infrastructure artefacts) and changes to the domain ontologies as the knowledge on top of the artefact layer. In this thesis, we focus on ontology change management only.

The changes in the domain ontology have to be operationalized in a suitable manner. There is no single ontology change management system which is widely accepted and provides different levels of change operators based on granularity, domain specificity and abstraction. There is a wide range of ontology change models which can be used for ontology evolution. However, all of them provide generic level change operations. Similarly, ontology changes represented in the form of change logs etc. also provide details only at the elementary level. Such elementary level change operators and their representation are not adequate for domain experts who have less technical knowledge of ontology languages. Change representation procedures are not sufficient to represent how an ontology evolves over time. The clear representation of intuition behind any of the applied changes is missing in such change representation of an evolving domain ontology.

1.2.1 Research hypothesis

To respond to the research challenges, we have defined the research hypotheses as,

> *A Pattern-based ontology change framework, supporting*
> *- change customization and*
> *- layered representation,*
> *can lead to an effective* ontology change operationalisation and representation mechanism.*

*The effectiveness can be measured in terms of *reusability*, *functional suitability*, *completeness* and *correctness* of the proposed solution.

1.2.2 Research questions

To answer research issues and to achieve the main goals, we have defined a set of research questions which are divided into two sections based on the problem context mentioned earlier.

Section 1: Ontology Change Customization

Research Question 1: *How can the change operators be represented (as building blocks for ontology evolution) so that they are suitable for the domain experts and ontology engineers?*

Research Question 2: *How to represent domain-specific ontology change patterns so that the user can adapt, customize and reuse them easily?*

Section 2: Ontology Change Representation

Research Question 3: *How can the ontology changes be recorded in a way that is suitable for the ontology engineers and other users to clearly understand the evolution of domain ontologies?*

Research Question 4: *How can the intent behind the applied ontology changes be captured?*

Research Question 5: *How can the higher level change patterns be identified?*

1.3 Contribution

Layered ontology change operator framework: Change operators are the building blocks in ontology evolution. In this regard, having atomic (elementary) level change operations only in an ontology editing framework are not sufficient. Such low level change operations can provide only one type of information i.e. addition or deletion of any element of the ontology. Semantics of an applied change are missing from such ontology change representation. We identified four different levels of change operators in order to capture the semantics of an ontology change. These change operators are based on different levels of granularity, domain-specificity and abstraction. The first two layers are based on generic change operators; whereas the next two layers are domain-specific change patterns. These layers of change patterns are specific to a domain and capture the real changes in the selected domain.

Layered change representation model: To date, there is no ontology change management system exist that records the ontology changes based on different levels of granularity. Once changes are performed using elementary level change operations, they are recorded in the database at the elementary level accordingly. Such a change representation procedure is not sufficient to represent the intuition behind any applied change and thus, cannot capture the semantic impact of a change. We support the implementation of the layered change operator framework through layered change logs. Layered change logs capture the objective of ontology changes at a higher level of granularity and support a comprehensive understanding of ontology evolution. The layered change logs are formalised using a graph-based approach.

Algorithms to identify generic composite change patterns: The strategy to maintain the consistency and the validity of the ontology elements vary at elementary and composite level. This is due to the change in realization of intuition of an applied change. We provide the composite change detection algorithms that identify the higher level changes from the ontology change log. Such detection of higher level composite changes not only assist in formulating the ontology change log data in a more concise manner, but also helps in realizing the semantics and intuitions behind any applied change. We opt for a graph-based pattern matching approach in order to capture the composite changes. Detecting the composite level changes from an ontology change log also facilitates the validation of the content (data) in more suitable mode.

Algorithms to discover domain-specific change patterns: Good patterns always arise from practical experience. Change patterns, created in a collaborative environment, provide guidelines to ontology change management. Discovery of recurring change sequences from an ontology change log provides an opportunity to define reusable domain-specific change patterns that can be implemented in existing knowledge management systems. The aim here is the discovery of usage-driven change patterns in order to record the ontology changes at a higher level and support the pattern-based ontology evolution. We identify recurring change sequences from an ontology change log that captures changes at an operational level. We formalize the ontology change log using a graph-based approach. We use attributed graphs, which are typed over a generic graph with node and edge attribution. Each graph node represents an atomic ontology change and the attributes of such graph node provide metadata and change data details. We analyze ontology change logs, represented as graphs, and identify ordered/unordered change patterns.

1.4 Organization of the thesis

The organization of the thesis document is summarized in Figure 1.1.

- Chapter 2 presents the background in the area of semantic technologies and the evolution of domain ontologies. We discuss semantic languages, syntax for ontology representation, ontology elements and the existing ontology editors etc.

- Chapter 3 gives the literature review within the scope of the thesis and discusses state of the art in the area of ontology change operators, ontology change representation, graph-based pattern discovery etc.

- Chapter 4 presents the proposed layered change operator framework that can be used as the basis of the ontology evolution process.

- Chapter 5 describes a layered ontology change log model that logs the applied ontology changes at two different levels of abstraction.

- Chapter 6 describes our approach towards change pattern identification from the lower level change log. The chapter works as a bridge between layered ontology change framework (discussed in Chapter 4 and 5) and the algorithms for change pattern identification (given in Chapter 7 and 8).

- Chapter 7 presents the graph-based algorithm for the detection of generic composite change patterns from the lower-level change log.

- Chapter 8 presents the graph-based algorithm for the discovery of domain-specific change patterns from the lower-level change log.

- Chapter 9 presents the experimental results and the evaluation of the main contributions.

- Chapter 10 summarizes the contribution of the thesis and discusses future work and directions.

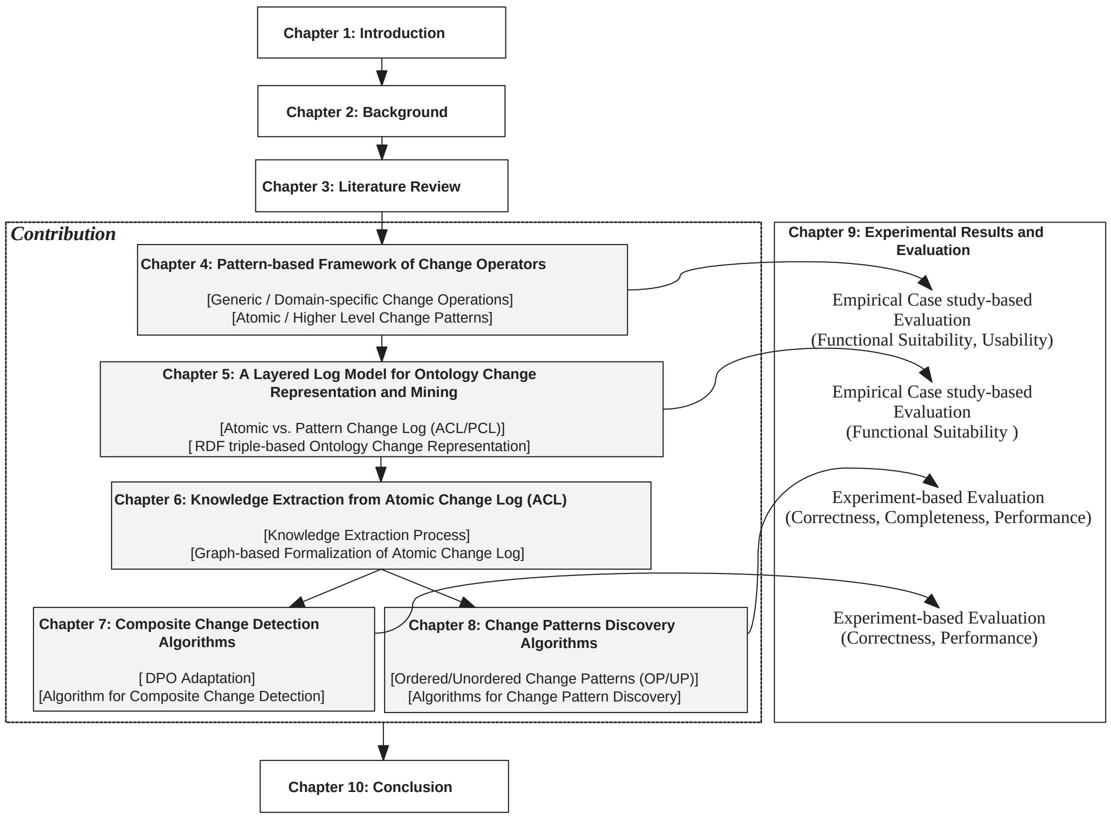

Chapter 1: Introduction
Chapter 2: Background
Chapter 3: Literature Review
Contribution
Chapter 4: Pattern-based Framework of Change Operators
[Generic / Domain-specific Change Operations]
[Atomic / Higher Level Change Patterns]
Chapter 5: A Layered Log Model for Ontology Change Representation and Mining
[Atomic vs. Pattern Change Log (ACL/PCL)]
[RDF triple-based Ontology Change Representation]
Chapter 6: Knowledge Extraction from Atomic Change Log (ACL)
[Knowledge Extraction Process]
[Graph-based Formalization of Atomic Change Log]
Chapter 7: Composite Change Detection Algorithms
[DPO Adaptation]
[Algorithm for Composite Change Detection]
Chapter 8: Change Patterns Discovery Algorithms
[Ordered/Unordered Change Patterns (OP/UP)]
[Algorithms for Change Pattern Discovery]
Chapter 9: Experimental Results and Evaluation
Empirical Case study-based Evaluation (Functional Suitability, Usability)
Empirical Case study-based Evaluation (Functional Suitability)
Experiment-based Evaluation (Correctness, Completeness, Performance)
Experiment-based Evaluation (Correctness, Performance)
Chapter 10: Conclusion

Figure 1.1: Organization of the Thesis

Chapter 2

Background

The aim of this chapter is to present a brief background for non-professionals having no background of semantic technologies, more specifically of Web Ontology Language (OWL), ontology elements and editors. The chapter is structured as follows: In Section 2.1, we introduce the semantic web and its two-part vision. In Section 2.2, we discuss the two web languages, i.e. RDF and OWL. In Section 2.4.2, few OWL syntaxes are discussed. We discuss the ontology elements (including OWL constructs and axioms) and ontology editors in Section 2.4.3 and 2.5, respectively. OWL API is briefly discussed in Section 2.4.4. In Section 2.6, we give an overview of different type of graphs (that can be used to represent the domain ontologies and ontology changes). We end with a summary in Section 2.7.

2.1 Introduction

Tim Berners-Lee, the inventor of the World Wide Web, has a two-part vision of the *Semantic Web* i.e. first, to make the web a more collaborative environment and second, to make the content more understandable, semantically enriched and thus processable by machines. This vision involves more than simply retrieving Hyper Text Markup

Language (HTML) pages from the web servers. There are relationships involved between the difference types of information content. For example, a document *isAbout* a certain entity (Thing) of the world, the entity is *included* in a certain document, the document *isWrittenBy* a certain author etc. Such relationships are missing from the current web. Plus, additional metadata (about the information items) is required in order to make the content machine processable. In order to convert a data into a smarter (vz. semantically enriched) data, it passes through mainly four stages [Michael et al., 2003]. These stages are discussed below. In order to capture the knowledge from the data at these stages, a number of languages have been developed. These languages include HTML, XML, RDF, OWL etc.

1. First stage is the database records and text documents. Such documents are linked to a single application and cannot be used by any other application. The information entities of such documents are represented as unique Uniform Resource Identifiers (URIs). In such a case, the core knowledge of usage of such content lies in the application rather than in the content.

2. The second stage involves the eXtensible Markup Language (XML) documents that use a single vocabulary. XML documents provide the description of the isolated data values. Such isolated data is not linked to one single application; however, can be used by other applications within the same domain only. In such cases, different application within the same domain can use such data.

3. In the third stage, we move from data modeling to knowledge modeling. Here, data can be brought together from multiple domains and can be represented in the form of hierarchical taxonomies. Such hierarchical taxonomies of isolated data do not only help in accurate classification and retrieval of the data, but the relationships among the different categories of data items can also be built. In this case, data becomes smarter, as the documents not only represent the classified data, but

also the relationships among such classified data items. We use the Resource Description Framework (RDF) for such knowledge modeling (discussed in Section 2.2.1). The explicit representation of associations among entities do not exist in XML documents and is therefore a major advantage of RDF.

4. The last stage is to extract more knowledge from the existing data by using means of logical rules and inference. To apply such techniques of knowledge extraction, data can be represented in the form of ontologies using the Web Ontology Language (OWL), discussed in Section 2.2.2. Now, data is not only smart enough to describe the entities of the world and the relationship among them, but semantic techniques can also be used on such data to infer more concrete knowledge from it.

2.2 Knowledge representation languages for the web

In this section, we discuss Resource Description Framework (RDF) and Web Ontology Language (OWL).

2.2.1 RDF

The Resource Description Framework (RDF[1]) is used for modeling the conceptual description of content using an XML-based language. Originally, it was designed as a metadata model that defines the metadata about certain document (externals of a document) such as, author, creation date, last modification date etc. But later, it is used as a concrete data model similar to other conceptual modelling approaches such as entity-relationship or class diagrams. The basic idea of RDF data modelling is to make statements about a certain entity. These statements are in the form of subject-predicate-object expressions, known as triples. For example, triple *(Sky, hasColor, blue)* represents that the sky has the color blue. An RDF specification consists of a well-defined vocabu-

[1]`http://www.w3.org/RDF`

lary of classes and properties which is used to link the knowledge available in a document to its formal semantics.

2.2.2 OWL

The Web Ontology Language is a family of knowledge representation languages that are designed for use by the applications that need to process the content of information rather than just a data model [W3C, 2004]. OWL adds more meaning/semantic richness than that supported by XML, RDF, RDF/S by providing additional vocabulary along with other formal semantics. One example of such vocabulary is the *functional* property. The OWL language also allows inferring more knowledge about the content from the ontology using inference and automatic reasoning techniques. OWL comes in three different flavours, i.e. OWL Lite, OWL DL and OWL Full.

2.3 Query language

2.3.1 SPARQL

SPARQL Protocol and RDF Query Language (SPARQL) is a query language for RDF graphs (where, an RDF graph is a set of triples). It can be used for retrieving and accessing the data stored in the RDF format. SPARQL-based queries allow combinations of triple patterns, conjunctions, disjunctions and optional patterns[2]. The result of a query is an unordered set of the solutions. Each solution represents one possible way in which the variables of the SPARQL query can be bound to the RDF terms. The query may result in zero, one or many solution sequences. SPARQL query has four query forms that use the output from the pattern matching to form the result sets or RDF graphs. These query forms are SELECT, CONSTRUCT, ASK and DESCRIBE. SPARQL is a data-oriented query language that only queries the explicitly available information and

[2]`http://www.w3.org/TR/2006/WD-rdf-sparql-query-20061004`

there is no inference involved in the query language itself. SPARQL is an official W3C Recommendation.

Example 2.1: Below, the formal representation of a SPARQL query is given which retrieves all the `exclusion` type change patterns that use the `cascade` strategy.

```
select  ?p from <http://www.cngl.ie/plog/University_Administration.owl>
where{
?p rdf:type M_O:PatternChange .
?p M_O:changeType  M_O:Exclusion .
?p M_O:usedStrategy  M_O:cascade .
}
```

2.4 Web Ontology Language (OWL)

OWL is one of the knowledge representation languages and is intended to be used when the information contained in documents needs to be processed by applications, as opposed to situations where the content only needs to be presented to humans. To date, two versions of the OWL has been introduced, i.e. OWL 1 and OWL 2. OWL 1 allows users to add semantics to the content. It allows specifying far more about classes, properties and individuals in a domain ontology. A class can be a subclass, superclass, disjoint or equivalent to another defined class. A property can be transitive, inverse or symmetric etc. Individuals can be the same or different from each other. OWL 2 extends OWL 1 and inherits the language features.

As described by W3C, OWL 2 contains following additional features:

1. Syntactic sugar to make common statements easier to say: New constructs are introduced to represent common changes. These constructs includes DisjointUnion, DisjointClasses, NegativeObjectPropertyAssertion, NegativeDataPropertyAssertion.

2. Introduction of new constructs for properties: For example, self reflexivity, qualified cardinality restrictions, reflexive, irreflexive and asymmetric object properties, disjointness among properties etc.

3. Keys: OWL 2 allows defining keys to uniquely identify the named individuals. A HasKey axiom states that each named individual of a class is uniquely identified by a (data or object) property or a set of properties.

4. Extended support for datatypes: OWL 1 support only integers and strings as data types. OWL 2 includes support of different types of numbers (including, positive integer, decimal, float, double etc.) and strings (including PlainLiteral etc.).

5. Extended annotation capabilities.

2.4.1 OWL sublanguages

OWL specification includes three levels of expressiveness, i.e. OWL Lite, OWL DL and OWL Full.

2.4.1.1 OWL Lite

OWL Lite[3] is intended for those users who mainly require an entity classification (in the form of a hierarchical taxonomy) and few simple restrictions. For example, one restriction can be cardinality constraints which can have value either 0 or 1. The cardinality constraints are subdivided into minimum cardinality, maximum cardinality or general cardinality constraints. One can also construct restrictions on how properties can be used for certain instances of a class (known as Property Restrictions). The property restrictions include *allValuesFrom* and *someValuesFrom* restrictions.

[3]`http://www.w3.org/TR/2004/REC-owl-features-20040210/#s2.1`

2.4.1.2 OWL DL

OWL DL[4] is intended for those users who require full expressivity while keeping the computational completeness (all conclusions are guaranteed to be computable) and decidability (all computations will finish in finite time) [W3C, 2004]. OWL DL includes all existing constructs; however, they can only be used within certain restrictions. For example, an entity cannot be used as a class and instance at same time. In contrast to OWL Lite, fillers for cardinality constraints are not restricted to values 0 and 1 only. OWL DL consists of a number of extra synopses which can be used for additional expressiveness and for defining complex expressions. This includes *hasValue* restrictions on a certain property. For example, the property *Gender* can have only two values as a filler, i.e. Male or Female. Furthermore, we have the concept of *Class Expression* that represents a complex class. A complex class can be constructed using boolean combinations of class expression synopsis[5] *unionOf*, *complementOf*, *intersectionOf*.

2.4.1.3 OWL Full

OWL Full supports the same set of synopsis as OWL DL but is designed for maximum RDF compatibility. The main difference between OWL DL and OWL Full lies in restrictions on the ways some of the OWL and RDF features can be used. OWL Full allows mixing of OWL and RDF constructs. For example, similar to RDF Schema and in contrast to OWL DL, any resource can be used as a class, property or individual at any specific time. The benefit of OWL DL over OWL Full is the development of reasoning tools that can infer more knowledge from the ontologies and are supported by a strong combination of constraints. Similar to OWL DL, the classes can be represented as class expressions using available boolean combinations, i.e. *unionOf*, *complementOf*, *intersectionOf*.

[4]`http://www.w3.org/TR/2004/REC-owl-features-20040210/#s2.2`
[5]`http://www.ksl.stanford.edu/people/dlm/webont/OWLFeatureSynopsisJan22003.htm`

2.4.2 OWL ontology syntaxes

The OWL language is not defined with a specific syntax in mind. It is defined as a high level structural specification that can be mapped into a range of concrete syntaxes [Matthew, 2010]. We discuss three of them below.

2.4.2.1 Functional Style Syntax

Functional syntax is one example of a concrete syntax for OWL ontologies. It closely follows the structural specification and is used in the definition of the semantics of OWL ontologies. Figure 2.1 below shows an example of an equivalent class which specifies that the `Supervisor` class is equivalent to a person who supervises another person.

```
EquivalentClasses(:Supervisor
            ObjectIntersectionOf(:Person
                        DataSomeValueFrom(   :supervises
                                                :Person)
                                  )
                  )
```

Figure 2.1: Functional style syntax - An example

2.4.2.2 Manchester OWL Syntax

Manchester OWL syntax is a text-based specification of OWL ontologies. This representation of OWL ontologies are easy to understand and write. The core motivation behind the development of Manchester OWL syntax was the demand from the users, who do not have much knowledge of Description Logic and would like to edit class expressions in tools such as Protégé. The benefit of Manchester OWL syntax is its simplicity and ease of its use. The Figure 2.2 shows an example of an equivalent class `Supervisor` (same as given in Figure 2.1) in Manchester OWL syntax.

```
Class: Supervisor
     EquivalentTo: Person and (supervises some Person)
```

Figure 2.2: Manchester OWL syntax - An example

2.4.2.3 RDF/XML Syntax

There are a number of RDF-based syntaxes available in order to share, edit and construct OWL ontologies. RDF/XML is the standard exchange syntax, which any OWL-based ontology editing tool must comply with. Ontology documents written in RDF/XML syntax are actually XML documents that represent certain domain ontology using RDF and OWL synopses. Most of the ontology editing tools use this syntax as a default syntax for saving ontologies. Figure 2.3 represents the above given example of an equivalent class of `Supervisor` in the RDF/XML format.

```
<owl:Class rdf:about="http://www.cngl.ie/ontology/organisation.owl#Supervisor">
  <owl:equivalentClass>
     <owl:Restriction>
         <owl:onProperty rdf:resource="http://www.cngl.ie/ontology/organisation.owl#supervises"/>
         <owl:someValuesFrom rdf:resource="http://www.cngl.ie/ontology/organisation.owl#Person"/>
     </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>
```

Figure 2.3: RDF/XML Syntax - An Example

2.4.3 OWL ontology elements

In terms of computer science, the widely referred definition of *ontology* is given by Gruber as *a formal explicit specification of a shared conceptualization* [Gruber, 1993]. According to the OWL 2.0 specification, the main component of an ontology is a set of axioms. Each ontology is uniquely identified by its IRI. Different versions of a domain ontology can be recorded. Thus, to differentiate between each version of an ontology, each version can be represented using a versionIRI. An ontology can also import other ontologies in order to get access to their entities, axioms and expressions. Every ontology can also be annotated using annotation properties, such as the creator of the ontology, last modification date etc.

2.4.3.1 OWL constructs

Entities are the building blocks for any given axiom. Before detailing axioms, we first discuss different types of entities.

Class: The term *class* (known as *concept* in description logic (DL)) refers to a set of individuals. They are represented using IRIs in the ontology. Examples of a class can be *univeristy* or *organization* that represent the set of all universities and organizations, respectively. In the ontology, classes can be used to generate axioms such as *subClassOf(University, Organisation)* representing that "every university is an organisation". There exist two pre-defined classes in OWL 2.0, i.e. OWL:Thing and OWL:Nothing. OWL:Thing represents the set of all individuals and OWL:Nothing represents an empty set. Different classes may have relationship among each other such as disjointness, equivalence etc.

In OWL 2.0, classes and properties can be used to construct *class expressions*, in other words, *complex classes*. For example, one can define a complex class (as a superclass) of `Parent` by adding a minimum cardinality restriction on property `hasChild` (i.e. (`hasChild min 1 Person`)).

Individuals: Individuals represent the instances of a defined class. They are actual objects from the domain. OWL 2.0 divides the individuals into two types, i.e. `Named Individuals` and `Anonymous Individuals`. Named individuals are those individuals that are defined explicitly in the domain ontology and given a name to refer to. They are defined using IRIs, thus considered as an entity in OWL 2.0 specification. The example of a named individual in an axiom could be `classAssertion (PhD_Student,` `Javed)` representing that individual `Javed` is a PhD student. Anonymous Individuals are used to represent such instances whose identity is of no use. For example, `ObjectPropertyAssertion(isMarriedTo, Javed, _:a1 )`, `DataPropertyAssertion (hasGivenName, _:a1,''Arshia'')`, `DataPropertyAssertion(hasFamilyName,_:a1, ''Javed'')` represents that `Javed` is married to some unknown person. This unknown person has the given name `Arshia` and the family name `Javed`. Anonymous individuals are similar to the blank nodes in a RDF graph. Anonymous Individuals are represented using `nodeID`. The type of the `nodeID` of any anonymous individual is `String`. As anonymous individuals are not defined using IRIs, they are not considered as an entity in the OWL 2.0 specification.

Object property: Object properties are used in order to represent a relationship between two classes and at the instance level, they are instantiated to represent a relationship between two individuals. For example, `ObjectPropertyAssertion (isSupervisorOf, Claus, Javed)`. Object properties are represented using IRIs. There are two pre-defined object properties in OWL 2.0, i.e. `topObjectProperty` and `bottomObjectProperty`. The `topObjectProperty` connects all possible pair of individuals. `bottomObjectProperty` represents an empty set and does not connect any pair of individuals.

In OWL 2.0, the object properties can be used to construct object property expressions, in other words, complex object properties. So far, OWL 2 supports only two types of object property expressions, i.e. i) object properties itself as the simplest form of an object property expression and ii) inverse object properties that allow a bi-directional navigation.

Data property: Data properties are used to represent the data about a class in the form of a literal value. For example, a data property `hasAge` can be used to represent the age of an individual of type `Person`. Similar to an object property, there are two predefined data properties in OWL 2.0, i.e. `topDataProperty` and `bottomDataProperty`.

Datatype: As the name indicates, datatypes are the entities that represent the *type* of a literal value. The type of the data value can be string, number, Id etc. In this sense, datatypes are actually data ranges which allows them to be used in certain restrictions. Similar to classes, data types are also represented using IRIs in the ontology. An example of a datatype used in an axiom can be `DataPropertyRange(hasAge, xsd:integer)`, representing that the range of data property `hasAge` is xsd:integer.

Annotation property: An annotation property can be used to annotate the ontology itself, an axiom or a single IRI-based entity. Some of the pre-defined annotation properties are rdfs:label, rdfs:comment, owl:versionInfo, owl:priorVersion etc.

2.4.3.2 OWL axioms

An ontology is a collection of axioms[6]. Each axiom represents certain knowledge in a domain. Axioms are divided into a number of categories based on the OWL constructs. These categories include `class axiom`, `data property axiom`, `object property axiom` and `facts`. In OWL 2.0, more categories have been added in the list which includes `declaration axioms`, `HasKey` and `annotation axioms`. The `fact axioms` are renamed into `assertion axioms` in OWL 2.0 specification. The `declaration axioms` are used to declare (add) a new entity (IRI) in the domain ontology.

The `class axioms` are used to create relationships among different classes (such as, `disjoint classes`, `equivalent classes`, `subclasses` etc.). `object property axioms` can be used to define relationships between two object properties. Examples of such axioms are `sub-object property`, `equivalent object properties`, `disjoint object properties` etc. They can also be used to define a relationship between an object property and a class. Examples of such axioms are `object property domain` and `object property range`. `Data property axioms` are basically used to define re-

[6]`http://www.w3.org/TR/owl2-syntax/\#Axioms`

lationships among the data properties. Examples of such axioms are `equivalent data properties`, `disjoint data properties` and `sub-data property`. They can also be used to define a `data property domain` relationship between a data property and a class. Furthermore, we can also define a `data property range` relationship between a data property and a data range. `Assertions` are used to define axioms about individuals. Examples of such axioms are `same individual` (stating that two individual are equivalent), `different individuals` (stating that two individuals are different from each other), `object property assertion` (that allows to join one individual to another using an object property) etc.

2.4.4 OWL ontology editing APIs

Two widely used ontology APIs are OWL API and the Jena API. We discuss the OWL API below.

2.4.4.1 OWL API

OWL API[7] is an open-source Java API for creating and editing OWL ontologies. Currently, it is used worldwide as a reference API for ontology development in the OWL language. The OWL API is in line with published W3C recommendations. To date, two main versions of the OWL API have been released, i.e, OWL 1.0 and OWL 2.0. The latest version of the API focuses on OWL 2.0 specifications which cover OWL-Lite, OWL-DL and a few sections of OWL-Full. The current and previous versions of the OWL API are available for download and use under the LGPL and Apache licenses. The components of OWL API include RDF/XML (parser and writer), OWL/XML (parser and writer), OWL Functional Syntax (parser and writer), Turtle (parser and writer), KRSS (parser only), OBO file format (parser only) and interfaces for FaCT++, HermiT, Pellet and Racer reasoners. The API is maintained by the University of Manchester.

[7]`http://owlapi.sourceforge.net/`

2.5 Ontology editors

A number of ontology editing tools have emerged over time. Based on their purpose and application, they can be categorized in six different groups [Gomez-Perez et al., 2006]. These are ontology development tools, ontology evaluation tools, ontology merge and alignment tools, ontology based annotation tools, ontology querying tools and inference engines and ontology learning tools. Among the existing tools are Protégé, TopBraid Composer, KAarlruhe ONtology and Semantic web Tool (KAON), NeOn, OBO-Edit, Semantic Media Wiki (SMW), AmiGO, Ontolingua, Semantic Web Ontology Editor (SWOOP). In this section, we give a brief summary of two of them.

2.5.1 Protégé

Among all the existing ontology editors, Protégé[8] is the most prominent and widely used tool. The Protégé framework is written in Java, extensible and is based on the OWL API. It provides the basic foundation layer of functionalities (in the form of Plug-ins), such as change visualization, difference determination, consistency management etc. Protégé provides two ways of modeling ontologies i.e. Protégé-Frames and Protégé-OWL.

In Protégé-OWL editor user can create classes, class expressions, properties, property expressions and individuals etc. It is provided with two reasoning applications, i.e. Fact++ and Pellet, which can be used to infer knowledge which is not explicitly given in the ontology. As suggested in the W3C OWL recommendation, the default storage syntax of any ontology is RDF/XML.

The Protégé-Frames editor provides a complete user framework to aid ontology development, customizing the data entry forms (for entering instance level data) and storing the domain ontologies. The data entry forms are very useful to enter recurrent data about any particular type of class. For example, a user can create a data entry form

[8] `http://protege.stanford.edu/`

for a PhD student, in which s/he can add entries such as student id, student name, supervisor, affiliation (research group) etc.

2.5.2 TopBraid Composer

The TopBraid Composer is an industry-based data modelling environment. It can be used for connecting multiple data sources, defining rules and queries for semantic data processing. The TB Composer is fully compliant with W3C standards and can be used for developing semantic application including domain ontologies, knowledge models and their instance-level knowledge bases.

2.5.3 Semantic Media Wiki (SMW)

The Semantic Media Wiki (SMW[9]) is another application that is an extension to MediaWiki - a wiki application that allows browsing, tagging, evaluating and sharing the wiki content. SMW allows user to add semantic annotations to the presented wiki content. By adding semantic annotations, SMW supports a visual display of information, improved data structure, search and external reuse of the data.

2.5.4 KAarlsruhe ONtology (KAON)

KAON[10] (KArlsruhe ONtology and Semantic web Tool) is an ontology management tool specifically developed for business applications. It gives a graphical environment for the support of ontology creation and maintenance, known as "OI-Modeler". The ontology constructs such class, properties etc. can be created by selecting the specific element from the menu item. Users can drag and drop the selected ontology elements into the graphical environment in order to construct the ontology hierarchy. To support ontology evolution, KAON provides an option to setup the evolution parameters. The

[9] `http://semantic-mediawiki.org/`
[10] `http://kaon.semanticweb.org/`

ontology editor can propose how the ontology must react for the change management. For example, when a class is removed from the ontology, whether the orphaned subclasses must be reconnected to the ontology root, to a superclass or must be deleted. These decisions can be taken into consideration using an evolution parameter setup (known as evolution strategies). Such decisions are to be taken before an ontology change take place. At the time of change, one can only either accept or reject the change by looking into the provided impact of change. The tool allows storing ontologies in the KAON file format only (.kaon extension).

2.6 Graphs for ontology change representation

Ontologies and the (recorded) ontology changes can be represented in the form of a graph. In this section, we give a brief background on the graphs and different types of graphs.

In its simplest form, a graph G is a set of nodes and edges $G = (N, E)$. The nodes represent the core entities of a domain (covered by the graph) and the edges represent the links (relationships) among the different defined nodes. One may have one or multiple edges between a pair of graph nodes, depending on the type of the graph. A node without any edge (linked to it) is called an orphaned node. Each edge must have a source and a target node attached to it. Graphs can be of different types including directed, undirected, labelled, mixed, multi, attributed etc. Below we discuss few of them.

2.6.1 Directed/Undirected Graphs

The nodes of the graph are linked to each other using edges. Such edges can be directed or undirected that leads to two types of graph, i.e. directed and undirected graphs. An *undirected graph* is the one in which graph edges have no orientation. In such case, the edge $e(a, b)$ is identical to the edge $e(b, a)$ as there is not fixed source or target node stated for the specific edge. In contrast, edges in the *directed graphs* have specific orientation and thus, edge $e(a, b)$ is not identical to the edge $e(b, a)$. Here, edge $e(a, b)$ is considered to be directed from a (source node) to b (target node). b is called the head of the edge and a is called the tail of the edge. In other words, b is said to be direct successor of a and a is said to be direct predecessor of b. A *mixed graph* G is the one in which some edges may be directed and some other may be undirected.

2.6.2 Multi Graphs

One may find single or multiple edges between two nodes of a graph. It is also possible to have an edge which starts and ends on the same graph node. Such edges are called *loops*. A loop can be directed or undirected. The loops in a graph may or may not be permitted, depending on the requirements of the domain/application. In this sense, a *multi graph* is a graph that allows multiple edges and loops.

2.6.3 Labelled Graphs

The nodes and edges of a graph can also be labelled. This leads to two new types of graphs, i.e. node-labelled graphs and edge-labelled graphs. If used without defined qualifications, the term "labelled graph" refer to a node-labelled graph and means that each node of the graph is distinct and is labelled differently. For many applications, nodes and edges are given labels that are meaningful in those particular domains. Edges may also be assigned weights representing the "cost" of traversing between the two linked graph nodes.

2.6.4 Attribute Type Graphs

Attributes can be associated to nodes and edges of a graph. In this regard, nodes can be categorised into (actual) graph nodes and the attribute nodes. The attribute nodes can represent some property/data about the content represented by a graph node/edge. The attributed graphs with the notion of *typing* lead to a category of attributed graphs typed over an attributed type graph [Ehrig et al., 2004].

2.7 Summary

The vision of semantic web has two parts. First, to make the web a more collaborative environment and second, to make the content of the existing web more intelligent, understandable and semantically rich so that it can be processed by different agents including machines. To do so, a number of web languages have been developed including XML, RDF and OWL. Unlike HTML, the XML language allows to construct user-defined tags for content description. RDF uses XML-based syntax for adding description to web content. OWL is the standard language for creation of domain ontologies. It adds further semantics to the content and makes use of XML and RDF-based content description. The OWL language comes in three categories, i.e. OWL Lite, OWL DL and OWL Full. Domain ontologies written in OWL can be stored in different formats. Most common syntaxes are the Functional style syntax, Manchester OWL syntax, RDF/XML syntax and Turtle syntax.

An ontology is a set of axioms where classes, individuals, object properties and data properties are the main components of such axioms. We differentiate between entity declaration axioms, cardinality restriction axioms and others. Currently, there exist a number of ontology editing tools. Most of them are based on the OWL API, written in the Java programming language. The most common of them are Protégé and the NeOn Toolkit. The consistency of a domain ontology can be checked using ontology reasoners. These generally are available within an ontology editing toolkit. The most common ontology reasoners are FaCT++, Pellet, Hermit and RacerPro.

Graphs are the ordered pairs of nodes and edges. They can be used to represent the domain ontologies and the applied ontology changes. Graphs can be of different types including labeled/unlabeled graphs, directed/undirected graphs, multi graphs and attributed graphs. In this thesis, we used attributed graphs to represent the applied ontology changes, discussed in Chapter 6.

Chapter 3

Literature review

In the previous chapter, we discussed the research area background, in particular the ontology web language (OWL), different syntaxes for ontology representation, ontology constructs and existing ontology editing frameworks. As our main research topic is *ontology evolution*, in this chapter we review the related work in terms of the existing *operational* and *analytical* support for the ontology evolution process.

In the first part (Section 3.1), we discuss the ontology evolution in general. We give a brief summary of the different phases of the ontology evolution process proposed in the literature. We review the proposed evolutionary strategies that can be utilized for keeping the ontology (structurally) consistent. In the subsequent two sections (Sections 3.2 and 3.3), we review the operational side of the ontology evolution process. First, we discuss the ontology change operators proposed in the literature and, later, the recording of ontology changes, in terms of evolution logs, change logs, ontology versions etc.

In Sections 3.4 and 3.5, we review the analytical support for ontology evolution in terms of higher-level ontology change identification. In Section 3.4, we look at the necessity of representation of ontology changes as higher-level change patterns (in the light of published work). We briefly talk about the frequent subgraph mining and string matching algorithms in Section 3.5.

3.1 Ontology evolution

With the increase in usage of ontologies as a conceptual backbone by a large number of content-based applications, ontology change management becomes very vital. In terms of user-driven ontology change management, the requirements of the ontology evolution include

- ensuring the consistency of ontology and depending artifacts [Stojanovic et al., 2002],
- supervised ontology change application to support users [Tallis et al., 1999] and
- continual ontology refinement by advising users through evolution process [Noy et al., 2004].

In this section, we discuss proposed different phases of ontology evolution and briefly talk about the similarities and the differences between ontology and schema evolution. At the end, we examine different evolution strategies that can be utilized to meet the above mentioned requirements.

3.1.1 Different phases of ontology evolution process

Stojanovic [Stojanovic, 2004] proposed a six phase, cyclic ontology evolution process. The phases include change capturing, change representation, semantics of change, change implementation, change propagation and change validation.

- *Change Capturing*: The first stage is to detect the changes. Two types of changes can be distinguished, i.e. top-down changes and bottom-up changes.

  i. Top-down changes are those changes that are explicitly described by the user in the form of a “change request”.

  ii. Bottom-up changes are implicit and must be discovered by examining the underline data. For example, if some ontology classes are not used for quite a period of time then it is reasonable that such classes are not necessary and are obsolete and can be removed from the ontology. Such changes help with the continual improvement of the ontology and symbolize the needs of the evolving domain.

- *Change Representation*: Change representation is an important phase of ontology evolution process. Changes must be identified and represented in a suitable format [Maedche et al., 2002]. The ontology changes are usually described at the atomic level. However, more often the intent of change is described and visible at a higher level of granularity. Such higher levels of change representation help in expressing the objective of the change request explicitly. For example, to merge two classes `c1` and `c2`, one may perform a list of atomic change operations, i.e.

  1. Add class `mc`.
  2. Add subclassOf relationship between `mc` and superclass of `c1` and `c2`.
  3. Transfer all subclasses, properties and individuals (which were earlier related to `c1` and `c2`) to new class `mc`.
  4. Delete subclassOf relation between `c1`, `c2` and their current superclass.
  5. Delete classes `c1` and `c2`.

  It is required to merge two classes, but translating such an operation into five individual change operations (though, steps 3-5 itself are composite changes) leads to the loss of the intent of change and makes the whole process error prone. For example, the combination of change operations 1 and 2 exhibits that a sibling of classes `c1` and `c2` is added to the ontology, grouping of change operations 1, 2 and 3 depict as copying the classes `c1`, `c2` into `mc`. Thus, in order to make the intent of change explicit, it is preferable that changes must be represented at a higher level in the form of composite changes.

- *Semantics of Change:* The aim of the semantics of change phase is to resolve any issues that occurred during the application of ontology changes so that the consistent state of the edited ontology version is preserved. As defined by Stojanovic, “a single ontology is defined to be consistent with respect to its model if and only if it preserves the constraints defined for underlying ontology model” [Stojanovic, 2004]. Researchers have identified two types of inconsistencies, i.e. syntactic inconsistency and semantic inconsistency [Qin et al., 2009].

i. Syntactic inconsistencies - also known as structural inconsistencies, arise due to the existence of orphaned entities in the ontology or invalidation of defined constraints/restrictions. For example, the deletion of a subclassOf relationship from a class `c`, that had a single parent class, will lead to a syntactic inconsistency.

ii. Semantic inconsistency - occurs when the meaning of an entity is changed or gets ambiguous. For example, removal of object property `isCapitalOf` as a domain of class `Washington` leads to an ambiguity, e.g. whether such class refers to the city “Washington DC” or to a person “George Washington”. Such kind of ambiguities leads to semantic inconsistencies and must be removed from the ontology. One of the possible solutions is to represent the object property `isCapitalOf` as a necessary property for the class. Furthermore, introducing the class `City` as a superclass for `Washington` will eliminate any ambiguity. Another proposed potential solution is to store the metadata information along with the entities.

In [Stojanovic et al., 2002], the authors identified a number of branch points such as, what to do with the orphaned classes, what to do with the instances whose parent class is deleted, what to do with the properties without any domain and range etc. For each branch point, a number of evolutionary strategies are proposed, which can be used against the possible inconsistencies. For example, in the case of any orphaned subclasses of a deleted class `c`, in order to preserve the child classes, they can be attached to the parent of deleted class `c` or to the root class of the ontology. Evolutionary strategies are discussed in detail in Section 3.1.3.

- *Change Implementation:* This phase aims to provide the details of an ontology change. That means, what are the requested change operations and what are the consequences of such a change request (i.e. what other change operations will be performed along with the change request operations, in order to keep the ontology consistent). Such a list of change operations, user-requested or induced, is provided. A user can either accept the change operation list or cancel it. For example, a user is willing to delete a class `c` which contains two individuals as instances, and also acts as domain of an object property `p`. Before real implementation of this change request, a change impact analysis is performed and the user is informed that to keep the ontology (structurally) consistent, three more change operations will be induced (i.e. delete class assertions for two individuals and delete domainOf axiom for object property p). Once the user accepts the combined list of change operations, the change is applied.

- *Change Propagation:* The objective of the change propagation stage is to propagate the applied change request to the ontology instances, dependant ontologies and other artifacts.

- *Change Validation:* The last phase in the ontology evolution process is to validate the applied changes and confirm that once the changes have been applied, the ontology is back in a consistent state.

3.1.2 Ontology evolution vs. schema evolution

Ontology evolution is closely related to database schema evolution, specifically, to object oriented databases (OODB). Banerjee and Kim earlier addressed the semantics and implementation of schema evolution in object oriented databases [Banerjee et al., 1987]. They implemented a prototype, called *ORION*. They identified schema change taxonomy and described different types of schema level changes in it. This taxonomy of schema changes is adopted in most of the current schema evolution research for OODBs. One of the key characteristics of their approach is that the schema evolution is under a set of properties known as “invariants”. These invariants are the rules to construct the schema. However, there are a several differences between schema evolution and ontology evolution [Noy et al., 2004]. A few of them are given as follows:

- In object oriented database, there is clear distinction between schema and instance level data. However, in many knowledge representation languages (such as RDF), it is difficult to separate and distinguish between schema and instance level data.
- In the case of object oriented databases, instances and the classes are at different levels. A user performs a query about the database objects (instances). However, in the case of ontologies, instances and classes can be used, manipulated and queried together [Klein, 2004].
- Ontologies can be reused by merging them into other domain ontologies [Stojanovic, 2004]. In case of OODB, schemas cannot be incorporated into other schemas.
- In case of ontologies, reasoning mechanism can be applied. This helps in identifying the implicit knowledge which is not explicitly given in the domain ontologies.
- In terms of change propagation, the change propagation in OODB is only limited to the instances whereas, in case of ontologies, the change propagation is not only propagated to subclasses, direct instances but also on other dependant artifact which may include, linked ontologies, annotations and other applications [Djedidi et al., 2010].
- Different strategies can be used in order to meet the user/domain needs and maintain the consistency of the ontology [Abgaz et al., 2011].

3.1.3 Evolution strategies

To perform an ontology change, different combinations of change operations may lead to different consistent states of an ontology. Thus, it is not reasonable to limit the user to resolve the consistency issue in one way only. For example, a user may be interested in applying ontology changes so that there are a minimum number of effected ontology entities, while other users may be interested in preserving the ontology instances etc. To resolve the ontology changes based on a user's needs, user intervention is necessary.

Stojanovic introduced the notion of evolutionary strategies [Stojanovic et al., 2002], allowing a user to customize the ontology evolution process according to his/her needs. A user can choose one of the provided evolution strategy from the list, which meets his/her requirements. For example, in the case of an orphan property, the property can be connected to the super properties, can be deleted from the ontology or can be left as it is.

Two categories of evolutionary strategies has been proposed, i.e. *elementary evolution strategy* and *advanced evolution strategy.*

- *Elementary Evolution Strategies* - As described by the author, the *elementary evolution strategy EES is a set of possible ways for resolving resolution points.* Resolution points are those points during ontology evolution, from where different resolution ways lead to different versions of ontology. The resolution ways are the set of evolution strategies for resolving a particular resolution point. For example, *how to deal with orphan classes* is one resolution point and possibilities of linking the orphan classes to the parent of the deleted class, linking to the root of ontology or deleting the orphan classes as well, are different resolution ways of such resolution points. The author identified six different resolution points in the ontology which includes handling of orphan classes, orphan properties, properties without a domain etc.

- *Advanced Evolution Strategies* - The advances evolution strategies are based on the real business strategies which automatically combine the elementary evolution strategies. It defines how the evolving ontology must look like at the end of the evolution process. For example, single child class is allowed or not, depth of the class hierarchy should be as small as possible etc. The author classified the identified advance evolution strategies into structure-driven, process-driven and frequency-driven strategies.

Some more work with regard to evolution strategies is done by Elmer P. Wach. In [Wach, 2011], Wach proposed evolution strategies that dictate when and how to evolve the domain ontology by evaluating the impact of the evolution and without human intervention. The aim here is to automatically update and evolve the underlying product domain ontology (PDO), based on the proposed strategies and user feedback cycle.

3.2 Ontology change operationalisation

Change operators are the building blocks of the ontology evolution. The changes in an evolving ontology are performed using change operators. In order to explicitly provide semantics of the ontology changes, researchers have emphasized on classifying the ontology changes into a number of categories. The purpose of such categorisation is to define a layered taxonomy of change operators in order to provide adequate support for ontology users, having different types of background knowledge and reducing the effort (in terms of time and consistency) required in ontology evolution process.

3.2.1 Elementary, composite and complex change operations

The most prominent categorization of ontology changes is given by Stojanovic [Stojanovic, 2004] in the KAON project where changes are separated into three levels of abstraction, i.e. elementary, composite and complex changes.

- *Elementary Changes* - An elementary change performs an atomic change on a single entity of the ontology. The examples of elementary change operations would be `Add class`, `Add individual`, `Delete subclassOfAxiom` etc. Stojanovic argues that such elementary level change representation is not suitable at all times. In most often cases, the *intent of change* is represented at a higher level. If we represent changes as a sequence of elementary level changes, the intent of change can be interpreted in different ways and can mislead. Moreover, there is a mismatch in the objective of change and how the objective is actually achieved. For example in Figure 3.1, the goal is to split a class `Research Student` into two, i.e. `PhD_Student` and `MSByResearchStudent`, however, the goal can be interpreted differently at different time slots during the change operations (Table 3.1). The combination of the first 2 (and the first 4) change operations in the table describes a different intent of change operations, which is to add a new sibling to class `ResearchStudent`. Similarly, different other intents of change operations can be acknowledged through the process of a composite ontology change.

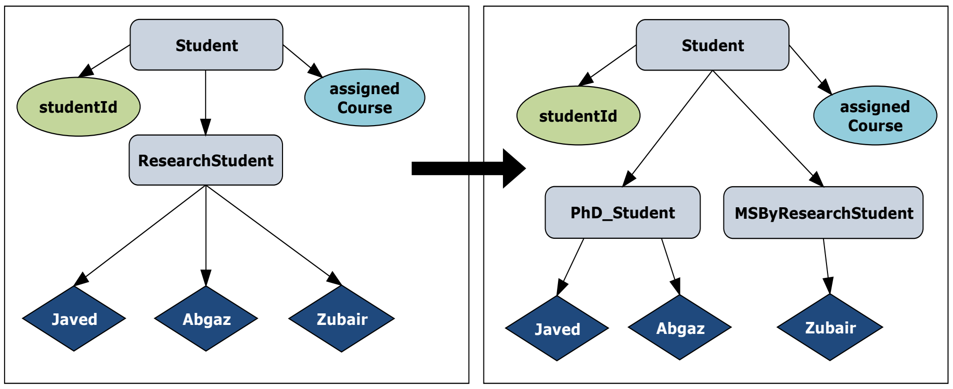


Figure 3.1: Composite change operation ``Split class''

To represent the intent of ontology change more explicitly at a higher level, Stojanovic proposed composite change operations.

- *Composite Changes* - A composite change applies changes to the target entity and

	Change Operations	**Intent of Change**
1 2	Add class (PhD_Student) Add subclassOfAxiom (Phd_Student, Student)	Add sibling to "Research Student"
3 4	Add class (MSByResearchStudent) Add subclassOfAxiom (MSByResearchStudent, Student)	Add sibling to "Research Student"
5 6	Add classAssertionAxiom (Javed, PhD_Student) Delete classAssertionAxiom (Javed, ResearchStudent)	Move instance (Javed, ResearchStudent, PhD_Student)
7 8	Add classAssertionAxiom (Abgaz, PhD_Student) Delete classAssertionAxiom (Abgaz, ResearchStudent)	Move instance (Abgaz, ResearchStudent, PhD_Student)
9 10	Add classAssertionAxiom (Zubair, MSByResearchStudent) Delete classAssertionAxiom (Zubair, ResearchStudent)	Move instance (Zubair, ResearchStudent, MSByResearchStudent)
11	Delete subclassOfAxiom (ResearchStudent, Student)	
12	Delete class (ResearchStudent)	Split class "ResearchStudent"

Table 3.1: Change operations and the intent

its neighborhood. As described by Stojanovic, *The neighborhood of a class consists of its subclasses, superclasses, properties, for which it is specified as a domain or as a range class, and instances defined for that class. The neighborhood of a property contains its domain class, range classes, subproperties, superpropeties, instances it is defined for as well as instances it points to. The neighborhood of an individual includes its (rdf:type) classes, properties that are instantiated for it as well as properties that point to it.*

Examples of composite ontology changes (related to the class-class relationship) include Merge classes, Split class, Move up class, Group classes etc. It is not feasible to present a comprehensive list of useful composite change operations, as in future, different combinations of elementary change operations may lead to new composite change operations. For example, splitting a class into two and making individuals as instances of both split classes may be extended into splitting a class into n (more than two) ontology classes and individuals may be split into all n ontology classes. Stojanovic argues that some extensions can also be domain-specific, for example, grouping of subclasses, which are parent of a concrete individual in an ontology model. Therefore, there exist another higher layer of abstraction of ontology changes, i.e. complex changes.

- *Complex Changes* - apply a change that is an arbitrary combination of at least two elementary and composite ontology changes.

In addition to the classification of ontology changes given above, the author categorized the ontology changes into *Additive* and *Subtractive* changes. The additive changes are those changes which add a new element in the ontology without altering the existing ontology elements. Whereas, the subtractive ontology change involve the deletion of few of the existing ontology elements.

3.2.2 Atomic and composite change operations - basic and complex

A similar categorisation of changes for OWL ontologies is given by Klein [Klein, 2004]. The author classifies the OWL ontology changes into *atomic* and *composite* types.

- *Atomic* change operations are similar to elementary change operations that can modify one single entity of OWL ontology model (e.g. `Delete subclassOfAxiom`, `Add class`). The author states that such atomic change operations can further be classified as *simple* or *rich* in content.

  - An *atomic simple* change operation is a basic change operation that can be determined from the ontology structure. Adding a new class, properties, individuals or creating relationships between classes (i.e. subclass, equivalent classes, disjoint classes) etc. are examples of atomic simple change operations.

- An *atomic rich* change operation is a complex change operation that expresses the implications of the applied changes. For example, an atomic rich change operation may specify that the range of property is *enlarged* (i.e. the range class of a particular property is changed to the superclass of the original range class).

- *Composite* change operations are composed of several atomic change operations and are of complex category. Such composite change operations can also be simple or rich in content, incorporating the implications of the change operations on the ontology model (e.g. add subtree, move siblings, restrict domain, merge multiple siblings, split into multiple siblings etc.).

The major difference between the KAON ontology change classification (given by Stojanovic) and the OWL ontology change classification (given by Klein) is, that Klein considers *modification* as a distinct type of change operation in order to provide complete specification that allows reversing the changes and is often available in the logs of changes provided by the tools. Change operation *modify* takes two arguments, one the old value and the other the new value. The old value is replaced by a new value.

3.2.3 Atomic, entity and complex change operations

Compared to the above given classification of ontology changes, Palma proposed a slightly different taxonomy of ontology changes comprising of *atomic*, *entity* and *composite* changes in his proposed generic change ontology [Palma et al., 2009].

- *Atomic changes* - Palma proposed a lower layer below the elementary change operation layer (as proposed by Stojanovic and Klein) and argues that elementary (atomic) change operations had been introduced as operations that cannot be subdivided into smaller operations; however, such change operations are all at the entity level. In his change ontology, an atomic change includes the applied *axioms* which later at the entity level can be associated to a specific ontology entity.

Addition of the class axioms, assertions, declaration, object property axioms are examples of atomic change operations.

- *Entity changes* - The next level at the top of atomic changes is the *entity* level. Entity level associates ontology changes to the ontology elements. The changes such as Add subclassOf, Add disjointClasses, Add inverseObjectProperty are examples of the entity level change operations as they are linked to a particular ontology element, which can be class, object property, data type property or individual.

- *Complex changes* - Similar to the previous approaches, the final level is comprised of *complex* changes. Complex changes are groups of entity changes that are applied together and constitute a logical entity, e.g. merge a set of siblings, group a set of classes etc. Similar to the past literature, s/he also mentioned that providing an exhaustive list of composite change operations is impracticable as entity and composite changes can be combined together in different ways to create new composite changes.

3.3 Ontology change representation

Once changes are implemented in the ontology model, the next step is to log the changes in a suitable format for explicit ontology change operational representation. In this sense, ontology change log data is a valuable source of information, based on which domain ontologies can evolve in order to reflect the changes in the domain, the user requirements, flaws in the initial design or the need to incorporate additional information [Haase et al., 2003]. In the past, researchers have opted different approaches to record ontology changes. We will discuss a few of them below.

3.3.1 Evolution logs

Stojanovic chooses *evolution logs* in order to record the applied changes [Stojanovic, 2004]. An evolution log keeps track of applied ontology changes in the exact order in which changes have been applied to the domain ontology. In case of any failure, an evolution log makes ontology recovery possible and is also used for traceability such as undo/redo functionalities. Each change entry contains the metadata of the change such as timestamp, author, version etc. The author mentions that as an evolution log is used for undo/redo functionalities, it is essential to differentiate between applied changes and the reverted changes. As one change may induce other additional changes to be applied, the change representation in an evolution log is in a hierarchical tree-like structure rather than in a linear form. A change which is requested by the user is represented using property *requestedChange*, whereas a property *causedChange* represents the cause-consequence relationship between a requested and a caused change.

3.3.2 Version logs

Plessers took a different approach and selects a version log in order to represent the evolutionary aspects of domain ontologies [Plessers et al., 2005]. According to the given definition, *a version log stores different versions of every entity (which includes classes, properties and individuals) ever defined in a domain ontology.* The purpose of a version log is to keep records of the different phases the entities pass through, from their creation, modification to deletion. The evolution of an ontology is recorded by preserving the history of each entity of the domain ontology. Two mechanisms can be used, i.e. timestamp or snapshot. In the case of timestamps, each ontology change is labeled with a timestamp which represents the sequence of ontology changes. Using the timestamp technique also helps in undo/redo functionality. In the case of snapshots, different states of the evolving ontology are captured by taking the snapshots of the ontology over time. The history of the ontology is then described by a sequence of snapshots. The snapshot technique can be used in two ways. First, to take a snapshot of the whole ontology over time and, second, by taking snapshots of each evolving entity over time. Using the first snapshot approach is very inefficient as one needs to store the whole ontology over time. Further, one can move from one state of the ontology to the other only and cannot switch to any state in between the two snapshot versions of the ontology. Thus, the author has adopted the snapshot approach that keeps records of each version of entity, instead of the whole ontology.

A version log also keeps records of the time (known as transaction time) when a new entity is introduced or an existing entity has been modified in the ontology. Each entity version has a status tag that can be *pending*, *confirmed* or *implemented*. The *Pending* state indicates that the change request has neither yet been implemented nor the consistency check has been made. The *Confirmed* state points out that the consistency check has been made, but change has not yet been implemented and the *Implemented* state indicates that the consistency check has been made and change has also been implemented in the public version of the ontology.

3.3.3 PromptDiff - capturing the structural differences

While some researchers have focused on the representation of ontology changes using change logs, others highlight the evolutionary aspects of an ontology by comparing two different versions of it. Noy proposed a fixed point algorithm in order to capture the structural differences between two ontology versions in the absence of ontology change logs [Noy et al., 2004]. This method compares two versions of the ontology and checks for each frame that whether ontology A contains any corresponding frame (its image) in ontology B. The technique represents the results in the form of a table, called the *PromptDiff* Table. The PromptDiff table is a set of tuples $< F1$, $F2$, $rename\_value$, $operation\_value$, $mapping\_level >$ where $F1$ and $F2$ are frames in ontology version 1 and ontology version 2, respectively. $rename\_value$ is a boolean value which is true if the frame has been renamed, otherwise false. *operation_value* represents the applied operation which can be either add, delete, split, merge or map. *mapping_level* indicates whether the two frames are different enough from each other and can have value *unchanged*, *changed* or *isomorphic*. Noy presented a set of heuristics matchers, where each matcher looks for a particular ontology structure, such as is-a hierarchy, properties attached to classes. The efficiency of the overall algorithm was enhanced by identifying the dependency relationships among the heuristics matcher. It was identified that not all the heuristics matcher are dependent on each other. Thus, as a result, a table was generated to realize that which matchers affect the other matchers and whom they affect.

The PromptDiff algorithms were implemented as a plugin in the Protégé-2000 ontology editing framework. An empirical evaluation strategy was selected in order to evaluate the algorithm and its results. Different ontology versions of two large projects (i.e. EON and PharmGKB) were considered as a case study. In total 10 matchers were used in their experiments and on average each matcher executed 2.3 times in each experiment. To evaluate the accuracy, only those frames which were changed within the versions were considered. On average, 96% of the matches between two frames of the ontology versions were identified. Among them, 93% of the matches were identified correctly. An important observation to mention here is that the performance of the algorithm did not decline even if two versions which are further apart from each other were selected.

3.3.4 Transformation set

Klein proposed a *transformation set* [Klein et al., 2003] that provides a list of change operations that if applied to the V_{old} (old version of ontology), the set transforms it to V_{new} (the new version of ontology). Such a transformation set can include elementary change operations, complex change operations or combination of them. Transformation sets are different from the basic change logs due to a number of reasons. First, basic change logs contain the record of all the applied changes, however, the transformation sets contain only the necessary set of change operations for achieving the resulting change. Second, basic change logs contain the *ordered* sequence of change operations. In a transformation set, such an ordering is very limited, which is primarily because all the *additive* operations will take place before any other change operation. Third, a change log is a distinctive representation of the exact applied change operations, whereas there can be several unique transformation sets that can produce the same resulting change.

3.4 Higher level ontology change identification

Recently, some researchers have focused on detection of higher level generic ontology changes [Papavassiliou et al., 2009, Groner et al., 2010]. In [Papavassiliou et al., 2009], the authors proposed a framework for defining changes in a formal language for RDF/S ontologies which considers change operations (in the form of RDF triples) in both schema and data. The RDF graph represents knowledge in the form of triples (subject, predicate, object). A set of all triples in an RDF graph can be given as a cross product.

$$\text{Triple Set } (\tau) = \mathcal{U} \text{ X } \mathcal{U} \text{ X } (\mathcal{U} \cup \mathcal{L})$$

where $\mathcal{U}$ denotes the URIs and $\mathcal{L}$ repesent the Literal values.

As low-level changes alone may not be enough to fully capture the intuition behind a change, the authors proposed an algorithm which detects composite changes based on comparison of two versions of an ontology. For example, we need to capture high level changes, such as *renaming of an entity*, *generalisation of the domain* etc, whose intuitions are not clear at the low level change representation. To capture the intuitions behind any change, it is necessary to represent a change at a higher level. As discussed before, a higher-level representation of a change is preferable than a lower-level one, as it is more intuitive, concise and closer to the intentions of the ontology editor and captures more accurately the semantics of a change. The representation of changes at a higher-level can help in checking the validity of the data. For example, if we only know that the domain of a property is changed we cannot presume anything about the validity of the existing data. But, if we know that the domain is generalised, we can assume that the validity of the data is not violated.

3.4.1 Change detection algorithm

In [Papavassiliou et al., 2009], the authors designed a change detection algorithm to capture the higher-level changes with respect to the proposed formal language. S/he mentions that the low level change triples can be associated with one and only one high level change. The author defines the changes Δ into two categories, the triples that are added in the new version (Δ_1) and the triples that are deleted (Δ_2). Thus, overall changes in a version correspond to the set of (Δ_1) and (Δ_2). A change is a set of addition changes (Δ_1), deletion changes (Δ_2) and the conditions for a certain composite changes to be true (ϕ).

$$Change(C) = (\Delta_1, \Delta_2, \phi) \tag{3.1}$$

The algorithm works as follows: It reads one single low level change and looks for all potential changes that could lead to a high level change. Once potential changes are found, for each captured change, it is to determine whether its conditions are satisfied or not. Based on the satisfied conditions, the set of potential higher level changes are generated.

3.5 Pattern mining

The mining of sequential patterns was first proposed by Agrawal and Srikant in [Agrawal et al., 1995] and later, the authors proposed the GSP algorithm based on an apriori property [Srikant et al., 1996]. Since then, many sequential pattern mining algorithms often based on specific domains [Li et al., 2008, Plantevit et al., 2010, Stefanowski, 2007, Zhu et al., 2007, Altschul et al., 1990, Zhang et al., 2007] have been suggested.

In the domain of DNA or protein sequences, BLAST [Altschul et al., 1990] is one of the most famous algorithms. Given a query sequence (candidate sequence), it searches for a matching sequence from the databases. In contrast, we focus on mining of change sequences (patterns) from an ontology change database. The MPPm algorithm [Zhang et al., 2007] focuses on mining frequent gap-constrained sequential patterns in a single genome sequence. In [Zhu et al., 2007], the authors proposed the MCPaS algorithm to answer the problems of mining complex patterns with gap requirements. Similar to our approach, it allows pattern generation and growing to be conducted step by step using gap-constrained pattern search.

3.5.1 Frequent subgraph mining

Several algorithms focus on graph-based pattern discovery [Inokuchi et al., 2000, Yan et al., 2002, Kuramochi et al., 2001, Huan, 2006]. In [Inokuchi et al., 2000], the author propose an apriori-based algorithm, called AGM, to discover frequent substructures. In [Yan et al., 2002], the authors proposed the gSpan (graph-based Substructure pattern mining) algorithm for mining frequent closed graphs and adopted a depth-first search strategy. In contrast to our work, their focus is on discovering frequent graph substructures without candidate sequence generation. A chemical compound dataset is compared with results of the FSG [Kuramochi et al., 2001] algorithm. The performance study shows that the gSpan outperforms FSG algorithm and is capable of mining large frequent subgraphs. The Fast Frequent Subgraph Mining (FFSM) algorithm [Huan, 2006] is an algorithm for graph-based pattern discovery. FFSM can be applied to protein structures to derive structural patterns. Their approach facilitates families of proteins demonstrating similar function to be analyzed for structural similarity. Compared with gSpan [Yan et al., 2002] and FSG [Kuramochi et al., 2001] algorithms using various support thresholds, FFSM is an order of magnitude faster. gSpan is more suitable for small graphs (with no more than 200 edges).

3.5.2 String matching algorithms

String matching is a fundamental part of text processing applications and databases. String pattern matching is similar to ontology change pattern matching in database, where a single word of the input string may refer to a single atomic change operation of the domain ontology. Given an input text string $s = s_1, s_2 \cdots s_m$ of length m, and a pattern string $p = p_1, p_2 \cdots p_n$ of length n, a string matching algorithm finds the instances of the pattern p in the text string s. A number of string matching algorithms have emerged with time. Frequently used algorithms include Brute Force, Boyer-Moore, Knuth-Morris-Pratt (KMP) and Karp-Rabin.

The simplest string matching algorithm is *Brute Force*. Starting from the first character of the given text string s, algorithm matches the first character of the pattern p. If the match is confirmed, the algorithm matches the remaining characters of the string using an iteration process. If the match fails during an iteration, the algorithm slides over one character ahead in string s and again starts the iteration from the first character. If the complete pattern string match is found, the algorithm returns the starting location of the pattern in string s.

Our algorithms for ontology change pattern identification differ from the brute force algorithm in the sense that we first iterate over the complete change log graph once and identify the locations of the seed (i.e., the locations in the change log graph from where a matched change sequence w.r.t. the referenced change sequence may start). Once such seed locations are identified, similar to the brute force algorithm, we perform the step by step matching task which includes comparing the change operation, change element and parameter types. Further, we introduce the notion of *node-distance* that allow us to define the search space for a match. The metrics and the change pattern mining algorithms are discussed in detail in chapters 6, 7 and 8.

3.6 Discussion

In this chapter, we discussed different phases of ontology evolution including change capturing, change representation, semantics of change etc. The ontology changes are categorized into top-down and bottom-up changes. The top-down changes are explicitly defined by the ontology engineers. Whenever there is a change in the domain, underlying domain ontologies need to be evolved accordingly. Bottom-up changes can only be identified through observations. One can exploit the ontology query log and can learn which classes or the areas of the domain ontology are more frequently visited. Based on such knowledge, the ontology engineer can add further knowledge in such areas of the domain ontology. By looking into ontology change logs, one can learn which domain classes are frequently changed. These changes indicate an evolving conceptualization of specific domain classes. Furthermore, change patterns can be identified from the ontology change logs (discussed in Chapter 7 and 8). In contrast, those classes of the domain ontologies which are not visited for a period of time can be considered as obsolete. Data mining approaches can be explored for such knowledge extraction processes (c.f. Section 6.1). Graphs are useful here as they support and can communicate information visually (c.f. Section 6.2).

Many evolution tasks cannot be applied using a single ontology change operator and requires a combination of them. Though, higher-level change operators have been suggested in the past (discussed in Section 3.2), changes are usually represented at the atomic level. To date, no ontology editing framework exist that allows users to perform and record ontology changes at a higher level of granularity. For each composite change to be performed, ontology engineers need to utilize atomic level change operations. This has a major impact on the ontology change operational cost in terms of number of steps to be performed (c.f. Section 9.2.3.2). Furthermore, performing each step manually makes the whole process error prone and leads to a risk of loss of ontology consistency and validity of instances [Qin et al., 2009]. Evolution strategies have been proposed in the past that help in resolving the consistency issues [Stojanovic et al., 2003]. These evolution strategies can only be applied at the atomic level and can resolve consistency issues at the structural level. For example, they can resolve the consistency issue of an orphan class by attaching it to the parent class or to the root class. However, "what to do with the subclasses, properties and individuals when a class is split into two sibling classes", "what to do when a class is pulled down in the class hierarchy and becomes a subclass of an earlier disjoint sibling class" etc. are still open issues. Such observations lead to our proposal, that evolution strategies are required at each level of granularity of ontology change operators (c.f. Section 7.3).

Different approaches have been used in order to record the ontology changes. One can utilize the evolution logs, where the applied ontology changes are recorded in the form of a sequence. As evolution logs record every single applied change including requested and caused changes, they can also be utilized for undo/redo functionalities or recovery processes. In contrast, version logs record the different versions of the ontology. Here, one can identify the difference between two versions. However, such representation does not illustrate how one reaches a version of the ontology. Different sets of change operations can be performed to reach a final version of the ontology. Another approach is to record structural differences between two versions of the ontology, rather than recording the complete ontology versions. Such an approach is more useful and preferable in case of large size domain ontologies. Again, similar to the version logs, structural differences between two versions of the domain ontologies illustrate the differences between the two versions, but what changes are actually been performed (that lead to differences) are missing from the representation. Another possibility is the transformation sets that record the necessary set of ontology change operations that, if applied to previous version of the domain ontology, will lead to the current version.

Similar to change operationalisation, the techniques to log the ontology changes record them at the atomic level. Though, few of such ontology change representations are complete (i.e. evolution and version logs), the intent of change request is not explicit and requires considering the applied ontology changes in a composite setup. A higher-level change log approach can fill this gap and can present the intent of change request explicitly (discussed in Chapter 5). Such higher-level abstraction of applied ontology changes helps in a comprehensive understanding of ontology evolution.

Similar to the higher-level change operators discussed in this chapter, composite change operators are suggested in our research. These change operators are generic and can be applied to any domain ontology. Higher-level ontology change operators are useful at the structural level. However, we believe that the changes at a higher level of granularity, which are frequent in a domain, can be represented as domain-specific patterns (discussed in Chapter 4). These change patterns are based on the viewpoints and activities of the users. They can either be explicitly defined by the users (user-defined change patters) or can be identified from the ontology change logs (usage-driven change patterns). Pattern mining approaches are required to identify such change patterns from the ontology change logs. In this regards, graphs are very useful to represent the change log data and to identify patterns from them. A number of graph-based pattern mining approaches already exist (a few are discussed in Section 3.5). These techniques need to be adapted and cannot be directly applied. One of the reason is the existence of type-equivalent ordered and unordered change sequences in ontology change logs (discussed in Chapter 6). Differences in two type-equivalent change sequences lead to sequence gaps [Zhang et al., 2007, Li et al., 2008]. Such sequence gaps have to be identified and dealt accordingly.

Chapter 4

Pattern-based framework of change operators

Change operationalization is a vital part of the ontology evolution process. In this chapter, we define a layered change operator framework, consisting of lower-level change operators and higher-level change patterns, that deals with ontology evolution, and in particular change customization and operationalisation. We identify four different levels of change operators based on the granularity, domain-specificity and abstraction of changes. The bottom two layers present the generic atomic and composite change operations. While compositional changes have been considered in the past, we added a domain-specific perspective through domain-specific change patterns that link the structural changes to the aspects represented in the domain ontologies. The top two layers present the domain-specific change patterns and the abstractions of such change patterns. Abstractions of the change patterns provide support for the transferability of the change patterns to a similar domain. We also present a coherent treatment of preservation of constraints throughout the compositional layers.

The chapter is structured as follows: In Section 4.1, we introduce the ontology change operations as the building blocks for ontology change management. In Section 4.2, we discuss the proposed layered change operator framework using some examples from the *university administration* and *database systems* domains. In Section 4.3, we introduce the constraints (data/value restrictions) that can be applied at each level. We conclude by giving a small summary at the end of the chapter.

4.1 Introduction

Ontology change operators are the building blocks of ontology evolution. Different levels of change operators have been suggested in the past [Stojanovic, 2004, Klein, 2004, Palma et al., 2009]. Such levels include atomic, composite and complex change operations. Atomic level change operations perform a single change on a single entity of the ontology. Such change operations can either add or delete an axiom (related to a target entity) from the domain ontology. Composite change operations are performed on the core building blocks (i.e. classes, properties and individuals) of the ontology hierarchy. They perform a set of atomic changes on the elements of the ontology hierarchy. "Modification" change operations (such as, *renaming a class*) are not considered here. At a fine-grained level, they are actually a combination of add and delete change operations and, thus, are considered as a composite change operation. Complex change operations perform changes in the ontology hierarchy at an arbitrary level.

In large domain ontologies, as a single change may violate multiple (structural and semantic) consistency constraints [Qin et al., 2009] due to its diverse and cascaded impact on other ontology elements and artifacts, manual ontology maintenance is error-prone, time consuming and, thus, not practically valid. Furthermore, as the domain experts normally are concerned with the evolution of information systems as a whole, they often have little knowledge of the ontology-based domain modelling. This restricts the usage of ontology change operations to ontology engineers only and reduces their usability.

A semi-automatic approach with proper human intervention (in terms of selecting change parameters and evolution strategies [Javed et al., 2012a, Stojanovic et al., 2002]) is the solution undertaken here that resolves the issues of time consumption and consistency management of the domain ontologies. The functional suitability and the adequacy of the solution are the core challenges for the proposed solution. The usage of ontology change operations may be extended to the domain experts and content managers. Higher level change operations can be applied to represent the semantics (intent) of the applied changes at a higher level. Further, constraints (c.f. Section 4.3) can be deployed at each level of change operations in order to keep the structural and semantic correctness of the underlying domain ontologies.

4.1.1 Introduction of case study domains

In this section, we briefly discuss the domains being used in the empirical case studies for the analysis and evaluation of our research work.

The main objective here is to study, identify and classify the changes that occur in the domain ontologies. As case studies, the domains *university administration*, *database systems* and *software application* were taken into consideration.

- The "university" as a domain represents an organisation involving people, organizational units and processes. The university ontology covers the core constituents of the domain which includes students, faculties, departments, schools etc. We considered Dublin City University (DCU[1]) as our case study and we conceptualized most of the activities and the processes in DCU for the construction of the domain ontology. Currently, the ontology consists of 61 classes, 22 object properties, 15 data properties and more than 450 individuals. In the university ontology, the changes are frequent at instance level due to employees joining or leaving, the introduction of new courses, student enrolment or graduation etc., but do also occur, albeit more irregular at schema level such as the introduction of a new employee

[1]www.dcu.ie

type, opening of new department etc. We distinguish between the schema-level and instance-level data of university ontology. We record the instance-level data (i.e. owl individual declarations and class/property assertions) into a separate RDF triple store. For our case study and pattern mining experiments (discussed in Chapter 8), we gathered the data of the faculty members, research students and administrative staff at the School of Computing[2] at DCU. The university ontology was developed by our research team.

- The "database systems" is a technical domain ontology that can be looked at from different perspectives - for instance being covered in a course or a textbook on the subject. The database textbook ontology was derived from the taxonomy arising from the table of content and the index [Elmasri et al., 2006]. Different database classes, such as relational algebra, relational calculus, database languages, etc. are specified in the database textbook ontology. In the database textbook, we identified a number of relationships between the different sections. For example, the *prerequisite* relationship among the different chapters, i.e. the topics covered in a chapter are necessary to understand the advanced topics available in another chapter (broader/narrower relationships). Such observations helped us in constructing the database course outline ontology. The technical database domain ontology was developed by the domain experts.

- The "software application" domain ontology is based on our work with an industrial partner where we took a content-centric perspective of a software application. The application system is a content management and archiving system. We specifically focused on the help system of the software application which contains help files. These help files contain a number of task components such as archiving, searching, sorting, messaging etc. One can find the key entities, such as GUI ele-

[2]`www.computing.dcu.ie`

ments, commands, procedures, role etc. of different software components in these help files. We identified such entities from the help files and added them to the software application domain ontology, constructing a class hierarchy. Furthermore, the software application domain ontology was extended by looking into the help management of the application as it contains the key components to provide definitions and solutions in the form of procedures, corresponding to the problems encountered in the software system.

More details about the empirical case study and the results are given in Chapter 9. The case study domain ontologies in RDF/XML format are given in appendices D - F . In the next section, while describing the layered change operator framework, different examples from the university and database domain ontologies are discussed.

4.2 Framework of change operators and patterns

Changes have to be identified and represented in a suitable format to resolve ontology change management issues [Oliver et al., 1999]. An explicit representation of ontology changes allows one to clearly analyze and understand a change request. A change in an ontology reflects the flaws in the earlier conceptualization of a domain, addition of new classes in the domain, removal of outdated classes from the domain etc. Based on our observation of common changes in all versions of the domain ontologies (c.f. Section 9.2), we studied the patterns they have in common, resulting in a framework of change operators (Figure 4.1):

- level one: elementary changes which are atomic tasks.
- level two: aggregated changes to represent composite, complex tasks.
- level three: domain-specific change patterns.
- level four: abstraction of the domain specific change patterns.

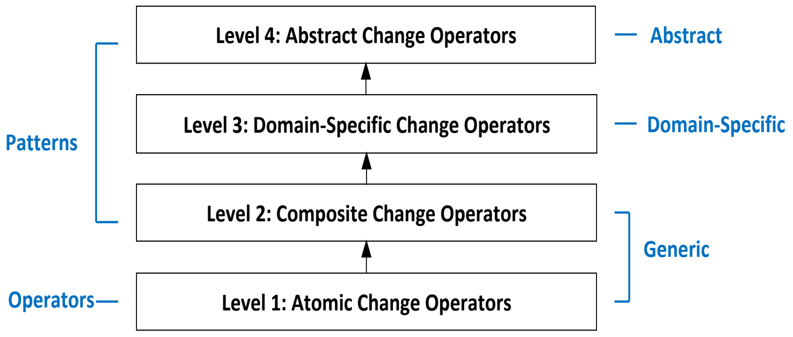

Level 4: Abstract Change Operators
Level 3: Domain-Specific Change Operators
Level 2: Composite Change Operators
Level 1: Atomic Change Operators
Patterns
Operators
Abstract
Domain-Specific
Generic

Figure 4.1: Different levels of change operators

The operators in level one and level two are similar to the types of change operations identified by Stojanovic [Stojanovic, 2004]. These change operators reflect generic structural changes; however, the change operators in level three and level four need to be customized (domain-specific change patterns). We observed that ontology changes are driven by certain types of common, often frequent changes in the application domain. Therefore, capturing these in the form of common and regularly occurring change patterns creates domain-specific abstractions. An example of such an abstraction from the *university ontology* domain is given in Figure 4.2. A number of basic change patterns may be provided so that users may adapt and generate their own change patterns to meet the domain demand. This makes the ontology evolution faster and easier. The dots at each level in Figure 4.2 represent that change operators are extensible.

Below, we discuss the layered change operations in general, using examples from the case study domains, for better clarification. More formalization of the layered change operations will follow in Chapter 5.

4.2.1 Generic structural levels

The first two layers of change operations are generic. These layers include *atomic* and *composite* level change operations. As these change operations are generic, such change operations can be used on any specified domain ontology. Below, we discuss each layer.

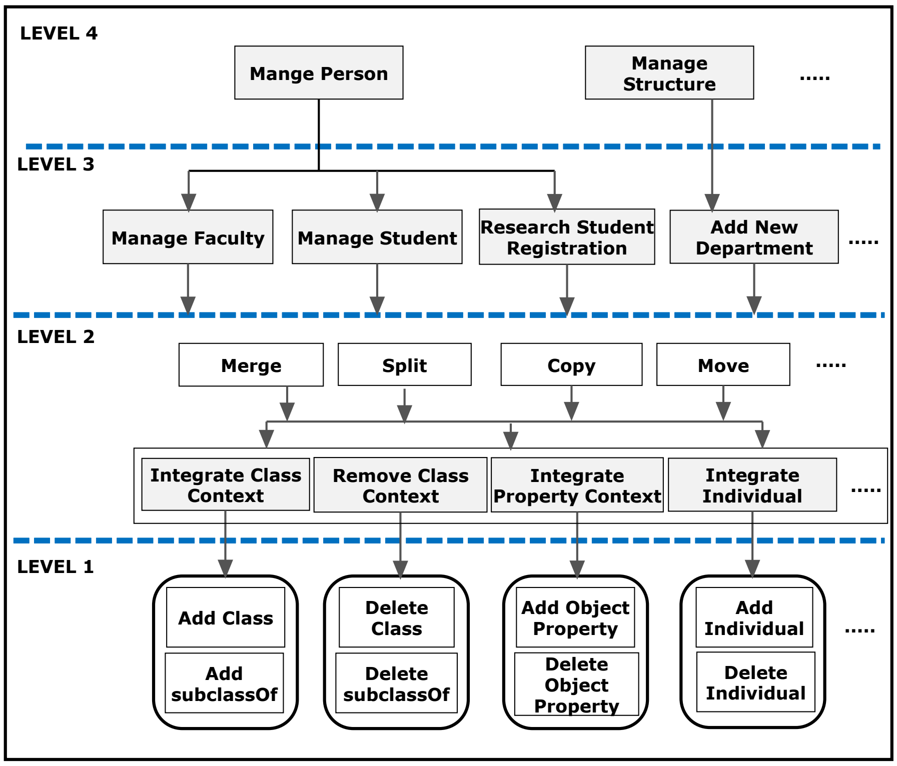


Figure 4.2: Architecture of layered change operators (university ontology)

4.2.1.1 Level one - atomic changes

Definition 4.1: Level one change operators are the atomic operators that can be used to perform a single ontology change. These operators add or remove a single axiom about a target entity in the domain ontology. A single change operator performs a single task that can add a single class, a single property or delete a single cardinality axiom, etc.

One can identify the atomic change operations based on the constituent components of the ontology. In terms of RDF triples, an atomic change can be represented using a single RDF triple. The subject and the objects of such RDF triple can be the named entities or the blank nodes (that point to other resources). An example of atomic change could be addition/deletion of a declaration axiom.

Example 4.1: Every defined class in a domain ontology is (at least) a subclass of owl:Thing. This can be accomplished using `classDeclarationAxiom`. However, in order to create a new class as a subclass of an existing one, implementation of such change operation alone is not sufficient. To do so, the change operations are always applied in a set where a class is declared in the ontology and added as a subclass of an existing class in the class hierarchy. For example, one can create a new class `PhD_Student` as a subclass of `Student` by using two atomic level change operations (Figure 4.3):

- creating a class `PhD_Student` using `Add classDeclarationAxiom`(`PhD_Student`) and
- making `PhD_Student` as a subclass of `Student` using `Add subClassOfAxiom` (`PhD_Student`, `Student`)

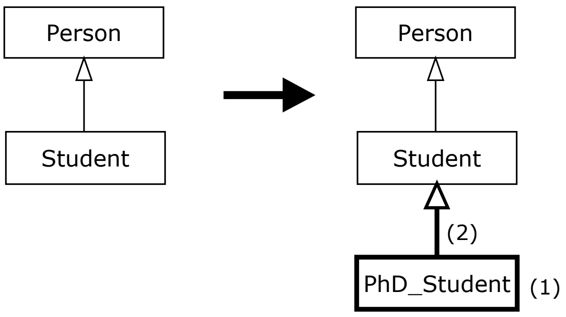


Figure 4.3: Atomic level change operations

4.2.1.2 Level two - composite changes

Many evolution tasks cannot be done by a single atomic change operation. A sequence of atomic change operations, defining a generic change pattern, is required. In terms of defining a composite change operation, there are two main approaches prominent in the literature. In the first approach, a user can combine different atomic change operations in different numbers depending on the specific goal in an ontology evolution situation. Such specific goals may come with their own set of atomic change operations and represent a “mental” operation [Klein, 2004]. Adding disjointness among the sibling classes (`addDisjointness`), stating that all individuals are different (`allDifferent`) or moving siblings (`moveSiblings`) are examples of such an approach. In the second approach, a composite change specifies a *target entity* in the form of “context” of the change and modifies (creates, removes or changes) the neighborhood of that target entity in the domain ontology [Stojanovic, 2004]. Such composite changes can be identified by looking into the neighborhood of any ontology entity. Splitting a class into two or more sibling classes (`Split class`), moving a property higher in the class hierarchy (`Pull up property`) or merging two or more classes into one (`Merge classes`) are examples of such an approach. In our research, we adopted the first approach as the composite change operations identified using second approach are actually a subset of the former.

Definition 4.2: We consider the composite change operations as generic change patterns that are identified by grouping atomic change operations to perform a composite task. For example “Remove Class Context” not only deletes a class from the class hierarchy, but also deletes all its roles[3]. To delete a single class `Faculty` in the university ontology, removing the class from the class hierarchy is not sufficient. Before we remove the class, we have to remove it from the domain and the range of properties like `hasPublication` or `supervises` that are attached to it. In addition, we need to either delete its subclasses (cascade delete) or attach them to the parent class. Depending on this context of an element, we use different operators from level one, resulting in a generic change pattern.

Example 4.2: In the *database* teaching domain, if an instructor wants to add a single chapter `SQL` to his/her course outline, the operator *“Integrate Class Context”* can be used. For example,

- create class `SQL` using `Add classDeclarationAxiom(SQL)`

[3]a role is an ontology axiom by which an entity (`Class`, `Property`, `Individual`) is attached to another entity of the domain ontology.

- make `SQL` subclass of `QueryLanguage` using `Add subClassOfAxiom` (`SQL`, `QueryLanguage`)

Example 4.3: If an ontology engineer wants to merge two or more classes, the operation requires operators higher than the "Integrate Class Context". For example, if the ontology engineer wants to combine two sibling categories of students, i.e. `PhD_Student` and `MSStudent`, into one single class `ResearchStudent`, the `Merge classes` composite change pattern can be used (Figure 4.4).
`Merge classes`((`PhD_Student`, `MSStudent`), `ResearchStudent`):

- *Integrate Class Context* by creating class `ResearchStudent` using `Add classDeclarationAxiom`(`ResearchStudent`) and then adding a subclass relationship using `Add subClassOfAxiom`(`ResearchStudent`, `Student`)

- *Add Object Property Domain* by adding domain `ResearchStudent` to `affiliatedTo` using `Add domainOfAxiom`(`affiliatedTo`, `ResearchStudent`)

- *Add Object Property Range* by adding range `ResearchStudent` to `isSupervisorOf` using `Add rangeOfAxiom`(`isSupervisorOf`, `ResearchStudent`)

- *Remove Class Context* by deleting class `PhD_Student` using `Delete Class` (`PhD_Student`) and then deleting class `MSStudent` using `Delete Class` (`MSStudent`)

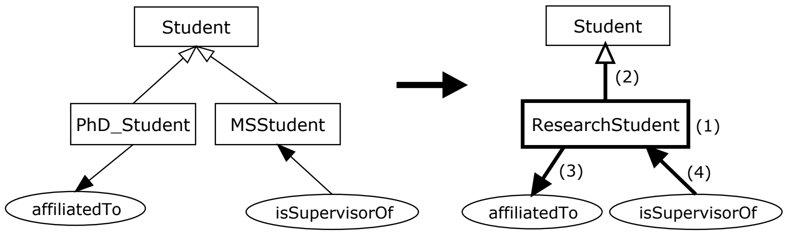


Figure 4.4: Composite level change operation ``Merge classes''

4.2.2 Domain-specific level

4.2.2.1 Level three - domain-specific change patterns

Definition 4.3: Level three change operations link the structural changes to the domain-specific aspects represented in the domain ontologies. In order to execute a single domain-specific change pattern, operations from level one and two are used. In addition, level three is constructed on the perspectives we identified in the construction stage of the domain ontologies. The change patterns are based on the viewpoints and activities of the users. Two users may have different perspectives to view the domain ontology which results in the use of a different combination of operations from level one and two. As the perspectives are different, the number of operations or the sequence of operations might differ. This difference results in patterns of change based on the perspectives of the ontology engineers.

Below, we discuss a few examples from our case study domains.

Example 4.4 - database systems: In the database system domain, the different perspectives we mentioned define their own patterns. From the teaching perspective, *"manage chapter"* has a pattern of calls such as create class *"chapter X"* for a specific topic, create properties such as *"isRequiredBy"*, *"isAlternateTo"* and *"isBasedOn"* to sequence topics in a course. From the perspectives of an author, a pattern to create class *"chapter X"* and `Pull Up Class` *"chapter X"* is often used. A technology domain expert only needs to include the technology as a new class and calls to create a class *"new technology"*.

Level three operators enable us to treat domain-specific operations separately and allow an ontology engineer to define his/her own change patterns once, which can be executed many times. For example, an instructor wants to manage the contents of his/her database course. S/he has different ways of managing the chapters by adding new chapters, altering the prerequisites, merging or splitting the chapters or a combination of one or more of the above.

Example 4.5 - university administration: In the case of the university administration domain, level three may contain change patterns such as *"manage faculty"*, *"open new department"* or *"PhD student registration"*. If a user needs to register a new PhD student in the university, then s/he creates a new individual of class *"PhD_Student"* and assigns a student id, email id, supervisor and department to him/her.

`PhD Student Registration` (`John`,`‘‘5810638’’`,`‘‘john2@dcu.ie’’`,`Gray`,`Computing`)

- *Integrate Individual Context* by creating a new individual `‘‘John’’` using

  `Add Individual(‘‘John’’)` and then

  `Add classAssertionAxiom(John, PhD_Student)`

- *Assign a student id* using

  `Add dataPropertyAssertionAxiom(John, studentId, ‘‘5810638’’)`

- *Assign an email id* using

  `Add dataPropertyAssertionAxiom(John, emailId, ‘‘john2@dcu.ie’’)`

- *Assign a faculty supervisor* using

  `Add objectPropertyAssertionAxiom(John, hasSupervisor, Gray)`

- *Assign a university department* using

  `Add objectPropertyAssertionAxiom(John, isMemberOf, Computing)`

Example 4.6 - university administration: In the case of a new managerial employee joining the university, the user can create a new individual of class *"StaffMember"* and assign the department, email id and supervisory function to him/her.

`Joining of a new Employee` (`Mark`, `Registry`, `‘‘a.mark@dcu.ie’’`, `Finance`)

- *Integrate Individual Context* by creating a new individual `‘‘Mark’’` using

  `Add Individual(‘‘Mark’’)` and then

  `Add classAssertionAxiom(Mark, StaffMember)`

- *Assign a department* using

  `Add objectPropertyAssertionAxiom(Mark, isMemberOf, Registry)`

- *Assign an email id* using

  `Add dataPropertyAssertionAxiom(Mark, emailId, ''a.mark@dcu.ie'')`

- *Assign a supervisory function* using

  `Add objectPropertyAssertionAxiom(Mark, isDirectorOf, Finance)`

The change pattern mentioned above is domain-specific and can be reused at instance level for the addition of new faculty or managerial employees. However, these variations are similar within a domain. A generic employee change pattern with common operations emerges (Level four change pattern).

4.2.3 Abstract level

4.2.3.1 Level four - generic categorisation

Definition 4.4: Level four change patterns are constructed based on the abstraction of the domain classes in level three. The main objective of introducing this level is to provide a facility that allows us to map domain-specific ontologies to existing upper level ontologies (i.e. categorizing domain classes in terms of abstract ones) and that helps to generalize and transfer patterns to other domains.

Example 4.7: The university administration ontology can be mapped and linked to any other organization that has a similar conceptual structure. In the university domain, one can identify classes such as students, faculties and employees; a production company may have employees, customers, owners or shareholders. Level four provides an abstraction to represent all these classes using a general class *"Person"*. In a similar fashion, the university system has research groups, departments, and committees; whereas a company may have research groups, departments and board of directors. We can abstract them as *"Structures"*. Furthermore, we have admission, examination, teaching, auditing in a university system and production, auditing and recruitments in an organization. We can abstract them to *"Processes"*. In the database systems ontology, one can identify relational algebra, relational calculus and SQL classes, whereas an ontology for Java programming may have classes such as control statements, class and thread. Level four provides an abstraction to represent all these classes using a general class such as *"Theory"*.

The benefit is the reuse of domain-specific patterns and their re-purposing for other, related domains, i.e. the transfer between domains. Level four provides a common ground to link the ontology with existing higher-level ontologies such as the Suggested Upper Merged Ontology (SUMO[4]) or MIddle Level Ontology (MILO) that provide the bridge between domains. It provides change patterns that can be applied to any subject domain ontology that is composed of a similar conceptual structure. Level four is constructed on top of level three and level two. Figure 4.2 represents the architecture of how the four levels are integrated and interconnected to each other.

This can actually be seen as part of the evaluation, where genericity and transferability are important criteria. Level four is actually a framework aspect that guides transfer of patterns to other domains (rather than being of specific importance for the user of a concrete application ontology).

4.3 Consistency constraints

The evolution of the domain ontologies with time is inevitable. Importing new classes into a domain, change of the conceptualization of a domain, adaptation to different applications etc. cause the execution of changes [Liang et al., 2005]. As the domain ontologies contain the commonly shared knowledge about a domain, the ontology evolu-

[4]Available at `http://www.ontologyportal.org`

tion process must maintain the consistency of the domain ontology and its entities. The consistency of the domain ontology can be preserved by introducing constraints at each level of change operators. These constraints are the conditions imposed on the structure of the domain ontology and the value restrictions for any specific ontology entity. The constraints not only aid in preserving the structural consistency of the domain classes, but also assist to keep the semantic consistency [Qin et al., 2009] at the domain-specific and abstract level.

Similar to our categorisation of change operators, we subdivided the constraints into generic, domain-specific and abstract (Table 4.1). As the change operators at level one and two are generic, the constraints introduced at these levels are the conditions to keep the structural consistency of the domain ontology (i.e. structural consistency constraints). As structural consistency constraints are not part of any specific domain, they can be introduced in the form of customizable evolution strategies [Stojanovic, 2004] in any ontology editing toolkit. The constraints at level three are domain-specific and can be utilized to keep the semantic consistency of the domain ontology entities (i.e. semantic consistency constraints). As semantic consistency constraints are domain-specific, they are part of the domain ontology and can be introduced in the form of value restrictions, cardinality restrictions, property characteristics etc.

4.3.1 Generic structural constraints

The consistency constraints at the generic level of layered change operator framework (i.e. level 1 and level 2 change operations) are used for keeping the structure of the ontology consistent according to the requirements of the user. For example, having a user-defined constraint that "a parent class cannot be a subclass to any of its child classes" ensures that there is no cyclic structure in the ontology hierarchy. A few of such generic consistency constraints are already defined in the existing ontology editing frameworks. "A class without any defined parent class (i.e. orphaned class) is always a

Type	**Constraints**	**Example**
Generic structural constraints for *Level 1, 2.*	Class fan-out	A class cannot have more than four subclasses
	Subclass Depth	A superclass cannot have a depth of more than five subclasses.
	Loopback Constraint	A broader class cannot be narrower to any of its subclasses.
	Orphaned Classes	Every defined class is at least a subclass of owl:Thing.
Domain-specific restriction constraints for *Level 3. (university administration)*	Value Constraint	Age of any faculty member must be between 25 and 60.
	Participation Constraints	A student cannot take more than 6 courses a semester.
	Participation Constraints	At least 15 elective courses must be offered in the final year of each undergraduate degree.
Abstract restriction constraints for *Level 4.*	Cardinality Constraints	A person cannot have more than one birth date.
	Cardinality Constraints	An organization department cannot have more than one Head.
	Value Constraints	The instantiation for the property *age* may have only positive integer values.

Table 4.1: Examples of consistency constraints

subclass of owl:Thing" and "an entity of the domain ontology cannot be deleted until all its roles are deleted", are the examples of such already defined constraints.

4.3.2 Domain-specific restriction constraints

The consistency constraints at level three of the change operator framework are domain-specific. They are directly applicable to the underlying domain of the ontology. Most of such constraints are defined either on the relationship restrictions among the defined classes (i.e. participation constraints) or on the values of the property instantiations at the instance level (i.e. value constraints). "A student cannot register for more than

6 courses in a semester", "the age of a faculty member must be between 25 and 60" are examples of level three domain-specific consistency constraints for the university administration ontology.

4.3.3 Abstract restriction constraints

The constraints at level four of the change operator framework are mainly the abstraction of the defined constraints at level three. At level four, one may add value constraints such as, that the data property *Age* defined for a class *Person* may only have positive integer values (value constraint), a person cannot have more than one date of birth (cardinality constraint), etc.

4.4 Summary

The ontology change operation can be atomic, composite or complex [Noy et al., 2004, Stojanovic, 2004, Qin et al., 2009]. This indicates that the effectiveness of a change is significantly dependent on the granularity, how the change operators are combined and the extent of their effect in the ontology. The impact of the change operators can affect the consistency of the ontology. Thus, a coherent treatment of the change operators and their effect on the consistency at each level of granularity becomes vital.

In this chapter, we introduced a framework to deal with ontology evolution through a framework of compositional operators and change patterns, based on the empirical evaluation of changes in a number of domain ontologies. We selected the university administration (as an organizational domain) and database systems (as a technical domain) as two key domains of our empirical case studies. The layered change operator framework has been empirically developed by looking into actual changes being applied in these domains. In this regard, domain experts and ontology engineers have contributed to the study. Different perspectives were identified based on the different viewpoints of the users that lead us to a layered change operator framework.

We discussed our approach for ontology evolution as a pattern-based compositional framework. The approach focuses on four levels of change operators and patterns which are based on granularity, domain-specificity and abstraction. We focus on domain-specific perspective-linking structural changes in domain ontologies. The changes at a higher level of granularity, which are frequent in a domain, can be represented as domain-specific change patterns, which are often neglected by the lower-level compositional change operators addressed in the literature. Thus, while ontology engineers typically deal with generic changes at level one and level two, domain experts and content managers can focus on domain-specific change patterns at level three. The abstraction of level three change patterns enable us to map the domain-specific level to abstract level and facilitate the smooth linking of domain ontologies with higher level ontologies, like SUMO and MILO.

We evaluated the proposed framework through empirical case studies, based on the practical validity and adequacy of the solution (discussed in Chapter 9). The empirical study indicates that the solution is applicable and adequate to efficiently handle ontology evolution. Our framework benefits in different ways. First, it enables us to deal with structural and semantic changes at two separate levels without loosing their interdependence. Second, it enables us to define a set of domain-specific change patterns. These domain-specific change patterns can be shared among other domain ontologies that have similar conceptualizations and specifications (our future work).

Chapter 5

A layered log model for ontology change representation and mining

Ontology-based information models helped researchers to take a step forward from traditional content management systems (CMS) to conceptual knowledge modelling to meet the requirements of semantically aware information system. Ontology-based approaches can be used to capture the architecture and process patterns [Gacitua-Decar et al., 2009]. Research as presented in [Filipowska et al., 2009] and [Hesse et al., 2008] stresses the contribution of ontologies to conceptual knowledge modelling. In this chapter, we discuss recording of the applied ontology changes in the form of a change log. Ontology change log data is a valuable source of information which reflects the changes in the domain, the user requirements, flaws in the initial design or the need to incorporate additional information. Ontology change logs can provide operational as well as analytical support in the ontology evolution process. We utilize an ontology change log in two ways, i.e. *recording* of applied ontology changes (operational) and *mining* of higher level change patterns (analytical).

We present a layered change log model approach to deal with customization and abstraction of ontology-based model evolution. The implementation of the change operator framework (discussed in Chapter 4) is supported through a layered change log model. The layered change log model at a higher level records the objective of the applied ontology changes and supports a comprehensive understanding of ontology evolution. We look into different knowledge gathering aspects to capture different facets of ontology change. The knowledge-based change log facilitates the detection of similarities within different time series, mining of change patterns and reuse of knowledge.

The chapter is structured as follows: In Section 5.1, we present the layered change log model. We discuss the RDF framework format, that is used to construct and represent the applied changes in RDF triple-form, in Section 5.2. In Section 5.3, we discuss the layered change log in terms of recording of applied changes. We talk about the fuzziness in the structure of higher level change log in Section 5.4. At the end, we give a brief summary of the layered change log model in Section 5.5.

5.1 Layered change log model (LCLM)

Recording the ontology changes at the atomic level is not sufficient. The atomic level representation of applied ontology changes can only present addition or deletion of an axiom from the domain ontology. The representation of intent behind an applied ontology change is missing from such a change log and mostly specified/deduced at a higher level of granularity. It is hard for an ontology engineer to understand why changes were performed, whether it is an atomic level change or a part of composite change and what is the impact of such change. Based on Figure 4.1(given in Chapter 4), we propose a layered change log model (LCLM), containing two different levels of granularity, i.e. an atomic change log (ACL) and a pattern change log (PCL)- Figure 5.1.

- *Atomic change log (ACL)*: The atomic change log represents an ontology change using atomic level (level 1) change operations. The benefit of storing ontology changes at an atomic level is their fine-grained and complete representation. Fine-

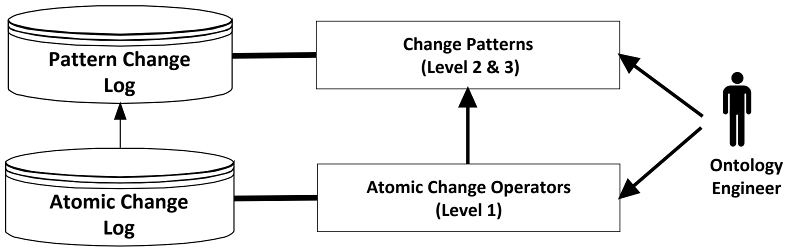

Pattern Change Log
Change Patterns (Level 2 & 3)
Atomic Change Log
Atomic Change Operators (Level 1)
Ontology Engineer

Figure 5.1: Layered Change Log Model (LCLM)

grained representation of ontology changes helps ontology engineers to understand the impact of the ontology changes at the atomic level. However, in most of the cases, the ontology changes are being applied as a group. Thus, the impact of the grouped changes must be identified at a higher level rather than atomic. One can extract such higher level change operations from the atomic change log, using pattern matching and discovery approaches, that leads to a comprehensive ontology change management approach.

- *Pattern change log (PCL)*: Pattern change log represents an ontology change using higher level (level 2 and 3) change operations, i.e. composite and domain-specific change patterns. Using the pattern change log, one can capture the objective of the ontology changes at a higher level of abstraction that help in a comprehensive understanding of ontology evolution. The intent behind any applied change is more visible at pattern level as compared to atomic.

The layered change log works with the layered change operator framework presented in previous chapter. The layered change log model has been used to achieve two purposes, i.e. change recording and pattern mining.

- *recording* ontology changes at different levels based on the utilized change operators - representing operational change log data (discussed in Section 5.3) and
- *mining* of valuable knowledge such as intent behind any applied change, domain-specific change patterns etc. from analytical change log data (discussed in Chapters

7 and 8).

5.2 RDF framework format

We use the RDF triple-based representation, i.e. *subject - predicate - object* (spo), to conceptualize the domain ontology changes in the change logs.

In order to keep a transparent record of applied ontology changes, we record two types of core metadata, i.e. who performed the change (User) and when the change was performed (Timestamp). To record such metadata, *Provenance Vocabulary Core Ontology*[1] terms can be used. In this regard, an ontology change can be considered as an activity (`rdf:type :Activity`) that is performed by a certain agent (`:Activity :performedBy :Agent`) where an agent can be a user (i.e., `:HumanActor`) or an application (`:NonHumanActor`). Further, a timestamp can be attached to such activity using the `completedAt` datatype proprerty (`:Acitivity :completedAt xsd:dateTime`). Similar to the provenance vocabulary core ontology, we constructed a change metadata model[2] using the OWL language. As we conceptualize the change metadata model in the form of an ontology, we use term "change metadata ontology" to represent the change metadata model. In order to maintain a fine granular ontology change representation, we distinguish five different knowledge gathering aspects (five W's), i.e.

- WHO performed the ontology change (User)
- WHEN the change is performed (Timestamp)
- HOW one can find the particular change in the change list (Session & Change Id)
- WHAT is the change (Operation & Element)
- WHERE particular change is applied in the ontology (Parameters)

[1] `http://trdf.sourceforge.net/provenance/ns.html`
[2] Available at http://www.computing.dcu.ie/~mjaved/MO.owl

The classes and properties available in the change metadata ontology assist the ontology engineer to construct the RDF triples, representing an applied ontology change. Similar to the approaches opted for by [Palma et al., 2009] and [Pedrinaci et al., 2007], the idea here is to provide a metadata model that is generic, independent and extendable to represent the changes of the domain ontologies. We used an RDF triple store to record the change logs, domain ontologies and change metadata ontology. Thus, all ontology changes, stored in the ontology change log, are in a form of triples. The main classes and properties of our change metadata model are given in Figure 5.2.

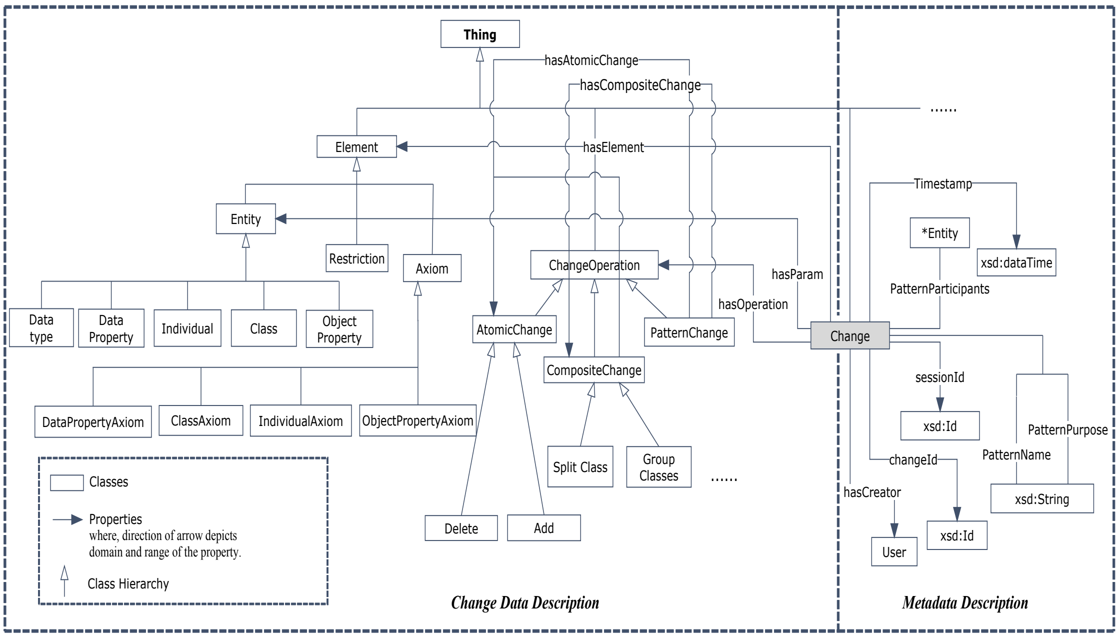


Figure 5.2: Change metadata ontology

The central class in the change metadata model is `Change`. Based on our proposed change operator framework (c.f. Section 4.2), where we recommended a layered representation of ontology changes, the class `ChangeOperation` is subdivided into `AtomicChange`, `CompositeChange` and `PatternChange`. The metadata details of an applied change, i.e. session Id, change Id, user and timestamp are given using properties `sessionId`, `changeId`, `hasCreator`, `Timestamp`, respectively. The core change data details of any applied ontology change, i.e. operation, element and parameters, are given using object properties hasOperation, hasElement (which is further subdivided into hasEntity and hasAxiom) and hasParam, respectively. The object property hasAxiom is further categorized into hasClassAxiom, hasObjectPropertyAxiom, hasDataPropertyAxiom and hasIndividualAxiom. The object property hasParam is further categorized into hasTargetParam, hasAuxParam1 and hasAuxParam2.

In order to express that the domain-specific change pattern is the combination of lower level change operations, the class PatternChange is associated to the class AtomicChange and the class CompositeChange using object properties hasAtomicChange and hasCompositeChange, respectively. Each stored domain-specific change pattern is an instance of (rdf:type) class PatternChange. The descriptive data of a change pattern, such as label, change Id, purpose are given using properties PatternName, changeId, PatternPurpose, respectively. For each composite or pattern change, its constituent atomic or composite changes are recorded; however, a complete decomposition is not intended. A reconstruction of lower levels can be facilitated through a pattern/composite change definition repository to be kept separate from the operational data.

We differentiate between *declaration/remover* axioms, *property restriction* axioms and other axioms. An atomic change can perform three kinds of actions, i.e. (i) addition/deletion of an entity (declaration/remover axioms), (ii) addition/deletion of a constraint (restriction axiom) or (iii) addition/deletion of an other (general) axiom. Therefore, we categorize the class Ontology_Elements in change metadata model into three subclasses, i.e. Entity, Axiom, Restriction. The class Axiom is further divided into ClassAxiom, ObjectPropertyAxiom, DataPropertyAxiom and IndividualAxiom.

The domain ontology changes can be logged either at the instance level (Abox) of the change metadata model or can be logged separately in the form of a change log. In our research, we separate the instance level data from the schema level data. Thus, the changes being applied on the domain ontologies are stored as RDF triples in ontology change logs. (<`Change`> <`hasParam`> <`Entity`>), (<`Change`> <`sessionId`> <`XSD:Id`>) are the examples of such RDF triple types.

5.3 Recording of ontology changes

Based on the utilized change operators, the applied ontology changes are recorded at two different levels of abstraction. If a user employs level one atomic change operators, the applied changes are recorded in the atomic change log (ACL). ACL is a sequential change log where each ontology change operation is executed one after the other. If a user makes use of higher level (level 2 or 3) change operators, the applied change is recorded as a change pattern in the pattern change log (PCL). Furthermore, for a complete representation of applied ontology changes, the applied change patterns are recorded as a sequence of atomic change operations in the atomic change log (Figure 5.3).

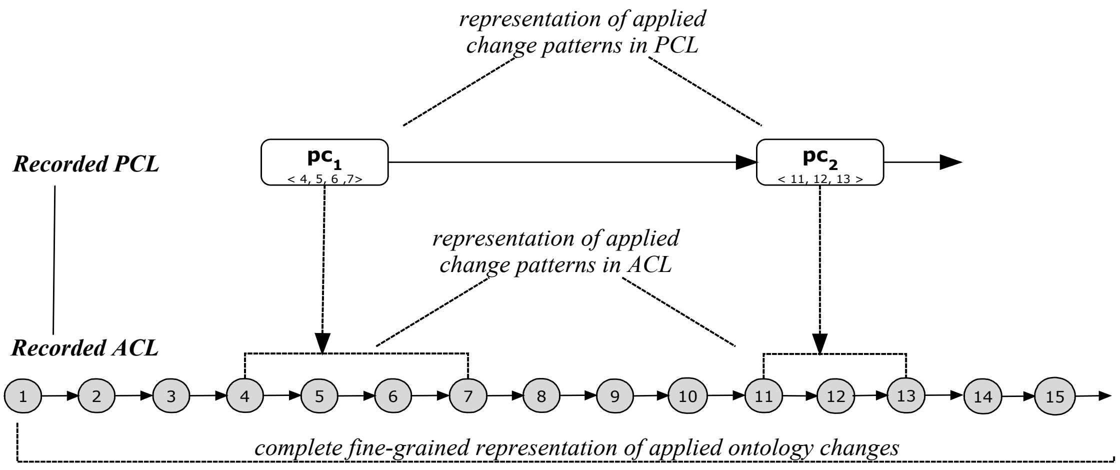

representation of applied change patterns in PCL
Recorded PCL
pc1
< 4, 5, 6 ,7>
pc2
< 11, 12, 13 >
representation of applied change patterns in ACL
Recorded ACL
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
complete fine-grained representation of applied ontology changes

Figure 5.3: Operational setup of ontology change logging

5.3.1 Recording of atomic changes in ACL

The atomic change log reflects the atomic level change representation of the applied ontology changes. The five aspects, mentioned in the previous section, are combined together to represent a single atomic level ontology change (Figure 5.4).

5.3.1.1 Formalization

Definition 5.1: An atomic change log consists of an ordered list of atomic ontology changes, $ACL = < ac_1,\ ac_2, ac_3 \cdots ac_n >$ where n refers to the sequence of ontology changes in a change log.

Each atomic ontology change is an instance of class `AtomicChange` of the change metadata ontology. The change consists of two types of data, i.e. *metadata* (M_A) and the *change data* (C_A) (Figures 5.2 and 5.4). As we can see in the change metadata ontology (given in Figure 5.2), the metadata provides the common details of the change, i.e. who performed the change, when the change was applied and how to identify such change from the change log. Thus, it can be given as $M_A = (id_s, id_c, u, t)$ where id_s, id_c, u and t represent `sessionId`, `changeId`, `User` and `Timestamp`, respectively. The change data contains the central information about the change request and can be given as $C_A = (o_p, e, p)$ where, o_p, e and p represent the `ChangeOperation`, `Element` and `Parameter Set` of a particular change. In Figure 5.2, such information is represented using object properties `hasOperation`, `hasElement` and `hasParam`.

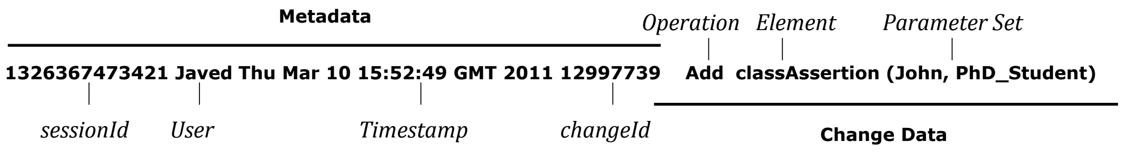

Metadata
Operation
Element
Parameter Set
1326367473421 Javed Thu Mar 10 15:52:49 GMT 2011 12997739 Add classAssertion (John, PhD_Student)
sessionId
User
Timestamp
changeId
Change Data

Figure 5.4: Sample representation of an atomic ontology change

5.3.1.2 Triple-based representation of atomic changes

Based on the atomic level change operation example given in Figure 5.4, the atomic change log entries for change operation `Add classAssertion` (`John`, `PhD_Student`) stored in the triple store are given in Table 5.1.

Table 5.1: Triple-based representation of an atomic change

Subject	Predicate	Object
MO:12997739	rdf:type	MO:AtomicChange
MO:12997739	MO:sessionId	“1326367473421”
MO:12997739	MO:hasCreator	MO:Javed
MO:12997739	MO:Timestamp	“Thu Mar 10 15:52:49 GMT 2011”
MO:12997739	MO:changeId	“12997739”
MO:12997739	MO:hasOperation	MO:Add
MO:12997739	MO:hasIndividualAxiom	MO:classAssertion
MO:12997739	MO:hasTargetParam	University:John
MO:12997739	MO:hasAuxParam1	MO:PhD_Student

5.3.2 Recording of change patterns in PCL

The pattern change log refers to the recorded change patterns being applied using higher level change operations of the layered change operator framework. Such specification of the applied change patterns help ontology engineers to i) distinguish between the applied similar changes and ii) in understanding the purpose and consequences of the changes.

5.3.2.1 Formalization

Definition 5.2: A pattern change log consists of an ordered list of ontology change patterns, $PCL = < pc_1, pc_2, pc_3 \cdots pc_n >$ where n refers to the number of change patterns available in a pattern change log.

The change patterns in PCL can either be level two generic composite change patterns or level three domain-specific change patterns (c.f. Figure 5.1). Similar to ACL, each ontology change pattern pc consists of two types of data, i.e. *Metadata (M_P)* and *Pattern data (C_P)*. The metadata provides meta-level details about the change pattern and can be given as M_P *= (id_s, id_c, u, t, p_u)* where, id_s, id_c, u, t and p_u represent the `sessionId`, `changeId`, `User`, `Timestamp` and `PatternPurpose`, respectively. The pattern data (C_P) provides a specification of the involved change operations. Here, C_P refers to the sequence of the change operations available in a change pattern $C_P = (c_1, c_2, \dots c_s)$

where s is the total number of change operations in a change pattern.

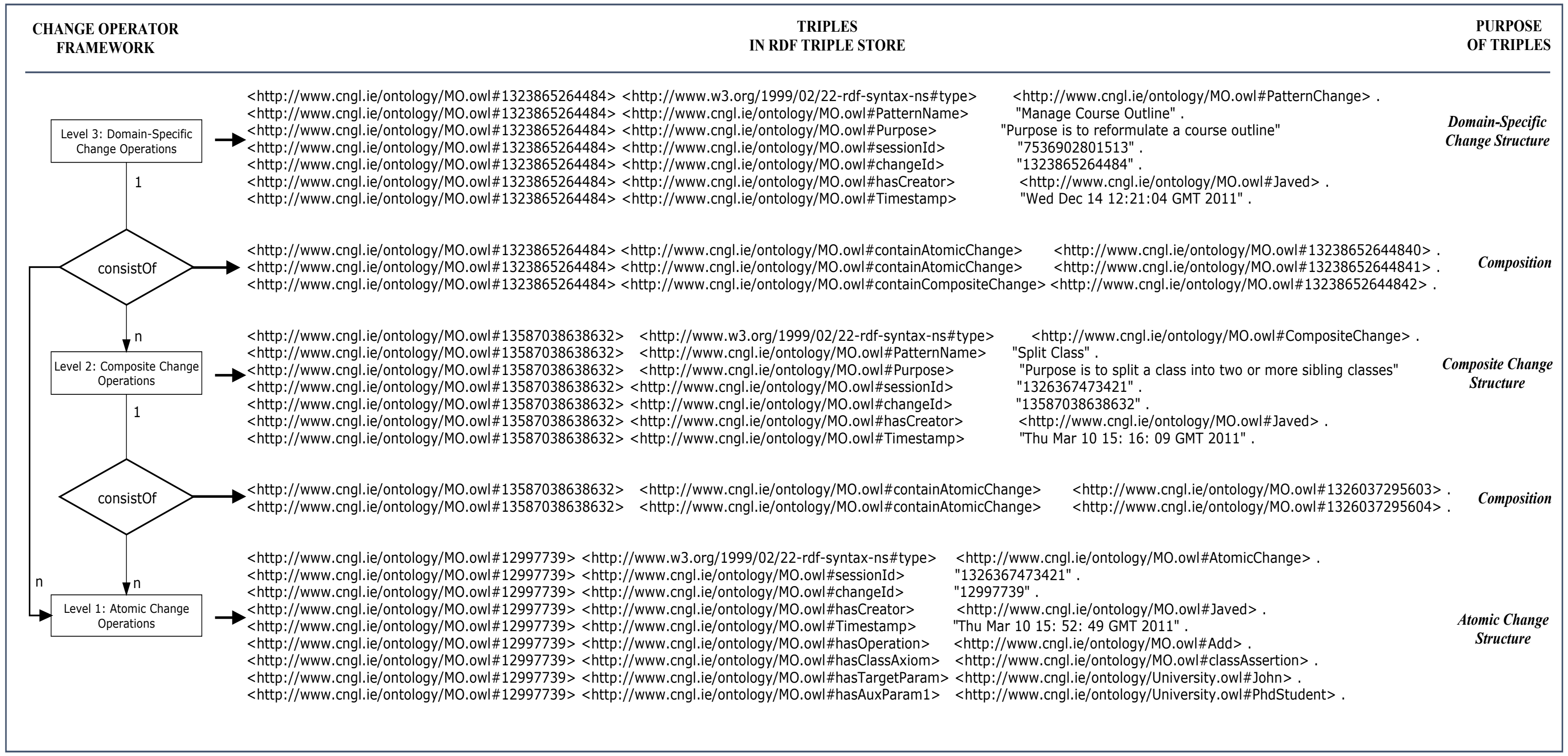
5.3.2.2 Triple-based representation of change patterns

Similar to the atomic change log, an RDF triple store can be used for recording of applied change patterns. In this sense, every applied change pattern is being recorded as an instance of either class `CompositeChange` or `PatternChange`, available in the change metadata ontology. Other common details, such as session Id, change Id, time of the applied change pattern are recorded using defined object and data properties in the change metadata ontology. Table 5.2 represents a section of a domain-specific change pattern log entries, representing the instantiation of a "PhD Student Registration" change pattern in the university administration domain. Such a change pattern consist of a number of atomic change operations. First, a new individual "John" is being added as an instance of class *PhD_Student*. Next, the details about the student Id, assigned supervisor etc. have been added to the specified student. The application of a change pattern is a single transaction on a domain ontology. In such a case, either the whole change pattern will be applied on the domain ontology, or will be discarded (rollback) completely if the pre and post conditions [Stojanovic et al., 2003] of any of the applied change pattern are not satisfied.

Figure 5.5 represents how the layered change operator framework (given in Figure 4.2) is mapped into the RDF Schema. The figure consists of three sections. On the left hand side, the layered change operator framework is given. In the middle, the mapping of ontology change operators in the form of RDF triples is given. On the right hand side, the purpose of the specific RDF triples is mentioned.

Table 5.2: Triple-based description of (domain-specific) change pattern

Subject	Predicate	Object
MO:13238651	rdf:type	MO:PatternChange
MO:13238651	MO:sessionId	“1326367473421”
MO:13238651	MO:hasCreator	MO:Javed
MO:13238651	MO:Timestamp	“Thu Mar 10 15:52:49 GMT 2011”
MO:13238651	MO:changeId	“13238651”
MO:13238651	MO:PatternName	“PhD Student Registration”
MO:13238651	MO:PatternPurpose	“Purpose is to register a new PhD student in school”
MO:13238651	MO:containAtomicChange	MO:12997738
MO:13238651	MO:containAtomicChange	MO:12997739
MO:13238651	MO:containAtomicChange	MO:12997740
MO:13238651	MO:containAtomicChange	MO:12997741
MO:13238651	MO:containCompositeChange	MO:12997742
MO:12997738	rdf:type	MO:AtomicChange
MO:12997738	MO:sessionId	“1326367473421”
MO:12997738	MO:hasCreator	MO:Javed
MO:12997738	MO:Timestamp	“Thu Mar 10 15:52:49 GMT 2011”
MO:12997738	MO:changeId	“12997738”
MO:12997738	MO:hasOperation	MO:Add
MO:12997739	MO:hasEntity	MO:Individual
MO:12997739	MO:hasTargetParam	“John”
MO:12997739	rdf:type	MO:AtomicChange
MO:12997739	MO:sessionId	“1326367473421”
MO:12997739	MO:hasCreator	MO:Javed
MO:12997739	MO:Timestamp	“Thu Mar 10 15:52:49 GMT 2011”
MO:12997739	MO:changeId	“12997739”
MO:12997739	MO:hasOperation	MO:Add
MO:12997739	MO:hasIndividualAxiom	MO:classAssertion
MO:12997739	MO:hasTargetParam	University:John
MO:12997739	MO:hasAuxParam1	MO:PhD_Student
MO:12997740	rdf:type	MO:AtomicChange
MO:12997740	MO:sessionId	“1326367473421”
MO:12997740	MO:hasCreator	MO:Javed
MO:12997740	MO:Timestamp	“Thu Mar 10 15:52:49 GMT 2011”
MO:12997740	MO:changeId	“12997740”
MO:12997740	MO:hasOperation	MO:Add
MO:12997740	MO:hasIndividualAxiom	MO:dataPropertyAssertionAxiom
MO:12997739	MO:hasTargetParam	University:John
MO:12997739	MO:hasAuxParam1	MO:studentId
MO:12997739	MO:hasAuxParam2	“5810638”


CHANGE OPERATOR FRAMEWORK
TRIPLES IN RDF TRIPLE STORE
PURPOSE OF TRIPLES
Level 3: Domain-Specific Change Operations
1
consistOf
n
Level 2: Composite Change Operations
1
consistOf
n
n
Level 1: Atomic Change Operations
Domain-Specific Change Structure
Composition
Composite Change Structure
Composition
Atomic Change Structure

```
<http://www.cngl.ie/ontology/MO.owl#1323865264484> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>        <http://www.cngl.ie/ontology/MO.owl#PatternChange> .
<http://www.cngl.ie/ontology/MO.owl#1323865264484> <http://www.cngl.ie/ontology/MO.owl#PatternName>         "Manage Course Outline" .
<http://www.cngl.ie/ontology/MO.owl#1323865264484> <http://www.cngl.ie/ontology/MO.owl#Purpose>            "Purpose is to reformulate a course outline"
<http://www.cngl.ie/ontology/MO.owl#1323865264484> <http://www.cngl.ie/ontology/MO.owl#sessionId>           "7536902801513" .
<http://www.cngl.ie/ontology/MO.owl#1323865264484> <http://www.cngl.ie/ontology/MO.owl#changeId>            "1323865264484" .
<http://www.cngl.ie/ontology/MO.owl#1323865264484> <http://www.cngl.ie/ontology/MO.owl#hasCreator>          <http://www.cngl.ie/ontology/MO.owl#Javed> .
<http://www.cngl.ie/ontology/MO.owl#1323865264484> <http://www.cngl.ie/ontology/MO.owl#Timestamp>           "Wed Dec 14 12:21:04 GMT 2011" .

<http://www.cngl.ie/ontology/MO.owl#1323865264484> <http://www.cngl.ie/ontology/MO.owl#containAtomicChange>     <http://www.cngl.ie/ontology/MO.owl#13238652644840> .
<http://www.cngl.ie/ontology/MO.owl#1323865264484> <http://www.cngl.ie/ontology/MO.owl#containAtomicChange>     <http://www.cngl.ie/ontology/MO.owl#13238652644841> .
<http://www.cngl.ie/ontology/MO.owl#1323865264484> <http://www.cngl.ie/ontology/MO.owl#containCompositeChange> <http://www.cngl.ie/ontology/MO.owl#13238652644842> .

<http://www.cngl.ie/ontology/MO.owl#13587038638632>   <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>      <http://www.cngl.ie/ontology/MO.owl#CompositeChange> .
<http://www.cngl.ie/ontology/MO.owl#13587038638632>   <http://www.cngl.ie/ontology/MO.owl#PatternName>     "Split Class" .
<http://www.cngl.ie/ontology/MO.owl#13587038638632>   <http://www.cngl.ie/ontology/MO.owl#Purpose>            "Purpose is to split a class into two or more sibling classes"
<http://www.cngl.ie/ontology/MO.owl#13587038638632> <http://www.cngl.ie/ontology/MO.owl#sessionId>            "1326367473421" .
<http://www.cngl.ie/ontology/MO.owl#13587038638632> <http://www.cngl.ie/ontology/MO.owl#changeId>             "13587038638632" .
<http://www.cngl.ie/ontology/MO.owl#13587038638632> <http://www.cngl.ie/ontology/MO.owl#hasCreator>           <http://www.cngl.ie/ontology/MO.owl#Javed> .
<http://www.cngl.ie/ontology/MO.owl#13587038638632> <http://www.cngl.ie/ontology/MO.owl#Timestamp>            "Thu Mar 10 15: 16: 09 GMT 2011" .

<http://www.cngl.ie/ontology/MO.owl#13587038638632>   <http://www.cngl.ie/ontology/MO.owl#containAtomicChange>     <http://www.cngl.ie/ontology/MO.owl#1326037295603> .
<http://www.cngl.ie/ontology/MO.owl#13587038638632>   <http://www.cngl.ie/ontology/MO.owl#containAtomicChange>     <http://www.cngl.ie/ontology/MO.owl#1326037295604> .

<http://www.cngl.ie/ontology/MO.owl#12997739> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>    <http://www.cngl.ie/ontology/MO.owl#AtomicChange> .
<http://www.cngl.ie/ontology/MO.owl#12997739> <http://www.cngl.ie/ontology/MO.owl#sessionId>          "1326367473421" .
<http://www.cngl.ie/ontology/MO.owl#12997739> <http://www.cngl.ie/ontology/MO.owl#changeId>           "12997739" .
<http://www.cngl.ie/ontology/MO.owl#12997739> <http://www.cngl.ie/ontology/MO.owl#hasCreator>         <http://www.cngl.ie/ontology/MO.owl#Javed> .
<http://www.cngl.ie/ontology/MO.owl#12997739> <http://www.cngl.ie/ontology/MO.owl#Timestamp>         "Thu Mar 10 15: 52: 49 GMT 2011" .
<http://www.cngl.ie/ontology/MO.owl#12997739> <http://www.cngl.ie/ontology/MO.owl#hasOperation>      <http://www.cngl.ie/ontology/MO.owl#Add> .
<http://www.cngl.ie/ontology/MO.owl#12997739> <http://www.cngl.ie/ontology/MO.owl#hasClassAxiom>    <http://www.cngl.ie/ontology/MO.owl#classAssertion> .
<http://www.cngl.ie/ontology/MO.owl#12997739> <http://www.cngl.ie/ontology/MO.owl#hasTargetParam> <http://www.cngl.ie/ontology/University.owl#John> .
<http://www.cngl.ie/ontology/MO.owl#12997739> <http://www.cngl.ie/ontology/MO.owl#hasAuxParam1>    <http://www.cngl.ie/ontology/University.owl#PhdStudent> .
```

Figure 5.5: RDF triple store data schema (operational change log data)

5.4 Incompleteness in the structure of analytical PCL data

Pattern change logs not only capture the explicitly applied ontology change patterns, but also the implicit change patterns. In this sense, we distinguish between the *operational* (recorded) and the *analytical* (mined) data of the change log. In terms of the PCL, the operational data of PCL records the applied pre-defined ontology change patterns, whereas analytical data of PCL records the ontology change patterns that are mined from the atomic change log. Figure 5.6 shows the joint information model as a class diagram, consisting of domain ontology, change metadata ontology, change log and triple store as core classes of the model. In this section, we focus on the analytical data of the ontology change log only.

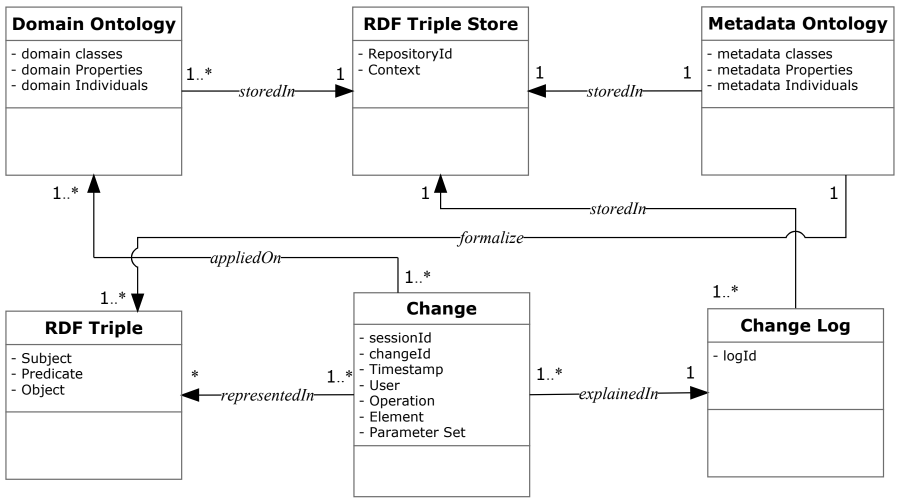


Figure 5.6: Core classes and their relationships

We utilize graph-based algorithms for change patterns discovery and matching from the atomic change log and allowed a gap between two adjacent change operations of a change pattern (i.e. permissible node-distance - c.f. Section 6.3.2). The analytical data of the pattern change log can be complex due to the *overlapping* of change patterns and possible *gaps* (Figure 5.7).

5.4.1 Evolution gaps in PCL

The objective of PCL analytical data is to identify those segments of ontology evolution where an implicit change pattern is being applied. This leads to the ontology evolution gaps in PCL. For a comprehensive understanding of the evolution of a domain ontology, where a user can visualize each step being taken during the ontology evolution, the user needs to go through the atomic change log. For a higher level of understanding of ontology evolution, where the change patterns capture the semantics of the ontology evolution steps, the user can make use of the pattern change log.

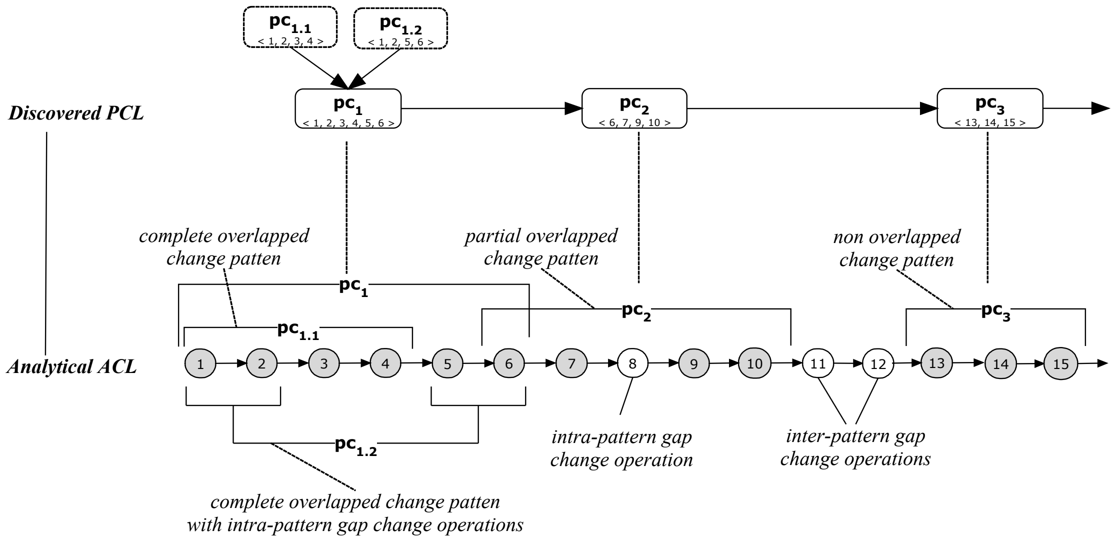

pc1.1 < 1, 2, 3, 4 >
pc1.2 < 1, 2, 5, 6 >
Discovered PCL
pc1 < 1, 2, 3, 4, 5, 6 >
pc2 < 6, 7, 9, 10 >
pc3 < 13, 14, 15 >
complete overlapped change patten
partial overlapped change patten
non overlapped change patten
pc1
pc1.1
pc2
pc3
Analytical ACL
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
pc1.2
intra-pattern gap change operation
inter-pattern gap change operations
complete overlapped change patten with intra-pattern gap change operations

Figure 5.7: Incompleteness in the structure of PCL (analytical change log data)

PCL contains two types of evolution gaps, i.e. inter-pattern evolution gaps and intra-pattern evolution gaps. The change operations exist in such evolution gaps are termed as "evolution gap change operations" - formulating an evolution gap change subsequence. Identification of such evolution gap change subsequences is necessary for the completion of the mined pattern change log. In Figure 5.7, each node represents an atomic change operation recorded in the atomic change log.

5.4.1.1 Inter-pattern evolution gaps

As all the atomic change operations recorded in ACL will not be extracted as a part of any identified change pattern, one can find the gaps between the adjacent change patterns of an analytical PCL. In Figure 5.7, change operations 11 and 12 are member of such an inter-pattern evolution gap change subsequence as they are not part of any identified change pattern of the PCL.

Definition 5.3: For the given atomic change log $ACL = < ac_1, ac_2, ac_3 \cdots ac_n >$ and the pattern change log $PCL = < pc_1, pc_2, pc_3 \cdots pc_m >$, an atomic change operation ac_k is the member of an inter-pattern evolution gap change subsequence if

- $ac_k \in ACL$ for $k = 1, 2 \cdots n$, then
- $ac_k \notin pc_i$ for all values of i, where, $i = 1, 2 \cdots m$

i.e. $ac_k \notin PCL$

5.4.1.2 Intra-pattern evolution gaps

One of our main objectives of ACL mining is to identify and capture semantically identical change sequences (in the form of change patterns). To achieve this, we allow a gap between two adjacent change operations of an identified change pattern sequence, called "node-distance" (c.f. Section 6.3.2). The maximum allowed node-distance is a user input to the change pattern discovery algorithms. Based on the permissible evolution gap, there may exist a change operation subsequence in ACL, between two adjacent change operations of a pattern, which is not part of the identified change pattern sequence. For example in Figure 5.7, change operation 8 is a member of such an intra-pattern gap change subsequence as it exists within pattern pc_2 in ACL, but is actually not a part of it (in PCL).

Definition 5.4: For the given atomic change log $ACL =< ac_1, ac_2, ac_3 \cdots ac_n >$ and an identified change pattern $pc_k =< ac_{p1}, ac_{p2}, ac_{p3} \cdots ac_{pm} >$, an atomic change operation ac_{pi} is a member of an intra-pattern evolution gap change subsequence if

- $ac_{pi} \in ACL$ for $i = 2 \cdots m - 1$, and
- $pos(ac_{p1}) < pos(ac_{pi}) < pos(ac_{pm})$ in ACL, and
- $ac_{pi} \notin pc_k$ in PCL

Things become more complex when an atomic change operation acts as an element of an identified change pattern in one case but acts as a evolution gap change operation in another identified change pattern. For example in Figure 5.7, change operations 3 and 4 act as an element of identified change pattern $pc_{1.1}$ but as an evolution gap change operations in case of change pattern $pc_{1.2}$.

5.4.2 Pattern overlapping in PCL

The change patterns available in PCL can overlap each other. That means, a set of atomic change operations can exist as part of two or more identified change patterns. Such overlapping can be either *complete* or *partial*.

5.4.2.1 Complete change pattern overlapping

If the sequence of atomic change operations in a change pattern p is actually a subsequence of atomic change operations available in an another change pattern q, we can say that change pattern p is completely overlapped by change pattern q (i.e. change pattern p is a subpattern of q).

Definition 5.5: For the given two change patterns $p_1 =< ac_1, ac_2, ac_3 \cdots ac_n >$ and $p_2 =< bc_1, bc_2, bc_3 \cdots bc_m >$, the change pattern p_1 is completely overlapped by change pattern p_2 if

- $ac_k \in p_1$ for $k = 1, 2 \cdots n$, then
- $ac_k \in p_2$ for all values of k.

i.e. $p_2 \cap p_1 = p_1$.

Example 5.1: In Figure 5.7, change patterns $pc_{1.1}$ and $pc_{1.2}$ are completely overlapped by change pattern pc_1.

5.4.2.2 Partial change pattern overlapping

Two change patterns are partially overlapped if they share one or more atomic change operations.

Definition 5.6: For the given two change patterns $p_1 =< ac_1, ac_2, ac_3 \cdots ac_n >$ and $p_2 =< bc_1, bc_2, bc_3 \cdots bc_m >$, the change pattern p_1 is partially overlapped by change pattern p_2 if

- $ac_k \in p_1$ for $k = 1, 2 \cdots n$, then
- $ac_k \in p_2$ for at least one value of k.

i.e. $p_2 \cap p_1 \neq \emptyset$.

Example 5.2: In Figure 5.7, change patterns pc_1 and pc_2 are partially overlapped as they share the atomic change operation "6" among their change pattern sequences.

5.5 Summary

The impact of an applied change operation can affect the consistency of the ontology. Thus, a definition of the ontology change operators that supports preservation of consistency and maintenance of overall integrity at each level of granularity becomes vital. Most of the time, the intent of an ontology change is not explicitly defined in lower-level change operators, thus requires representation of changes at higher levels. A layered change log framework fills this gap and helps an ontology engineer in explicitly understanding ontology changes.

In this chapter, we introduced the layered change log model for explicit operational representation of applied ontology changes. These layered ontology change logs not only provide operational support during the evolution of an ontology, but can also be utilized to extract implicit knowledge, such as change patterns, composite change patterns, causal dependencies among the ontology taxonomy etc. The atomic changes that perform a single change on a single entity of the ontology are stored in a lower level atomic change log (ACL). We utilized five different knowledge gathering aspects (five W's) in order to represent a single atomic ontology change. We distinguished between the metadata and the change data of the atomic change. The metadata refers to the Id of the change, user and timestamp etc., whereas the change data refers to the change operation, element and the set of parameters.

The changes on a higher level of granularity are stored as change patterns in the pattern change log (PCL). The pattern change log represents and captures the semantics of the applied atomic change operations. To do so, the PCL includes only those segments of evolution of a domain ontology where a generic or domain-specific change pattern has been applied. Similar to the atomic changes, we distinguish between the descriptive data and the pattern data. The descriptive data refers to the metadata of the pattern change (which includes details of the user, label of the pattern, change Id, etc.) and pattern data refers to the involved atomic change operations. The structure of the pattern change log can get complicated due to the existence of overlapping among the identified change patterns (from ACL) and the evolution gaps.

Ontology changes can be stored in RDF triple format. Such triple-based storage facilitates the storage of ontology changes at fine-grained level and can easily be queried using a query language such as SPARQL. One can utilize our pattern mining mechanism in order to extract higher level change patterns from atomic change logs. We extract the composite changes and the domain-specific change patterns from the atomic change log, which is discussed in detail in Chapters 7 and 8.

Chapter 6

Knowledge extraction from the atomic change log (ACL)

This chapter acts as an inter-linkage between operational aspects of ontology evolution (discussed in Chapter 4 and 5) and the analytical aspects (discussed in Chapter 7 and 8). The aim here is to give an overview of our approach and discuss the metrics used for mining of change patterns from atomic change log (ACL). We discuss a graph-based formalization of atomic change log data and introduce the metrics utilized for the identification of sequential abstractions from the change log graph. The algorithms for the identification of higher level (level two and three) change patterns are not given here and are discussed in detail in Chapter 7 and 8.

The chapter is structured as follows: In Section 6.1, we discuss the atomic change log data processing steps. It consists of various stages including data cleaning, session identification and data transformation. In Section 6.2, we discuss the formalization of processed atomic change log data into a change log graph. In Section 6.3, we present the analysis of the ontology change log graph and discuss a number of metrics that are used for pattern mining. We discuss the mining of the sequential abstractions from the ontology change log graph in Section 6.4. A brief summary is given in Section 6.5.

6.1 Knowledge extraction process

In this section, we discuss the steps taken for the extraction of the higher level change patterns from the atomic change log. One of the main stages is the *data preparation*, which overall consists of several steps including necessary data cleaning and filtering and data transformation (Figure 6.1). Researchers use different steps in different arrangements based on their viewpoints and requirements. We adopt the main data preparation steps from knowledge discovery for web log data [Ivancsy et al., 2006, Pabarskaite et al., 2007]. For our discussion here, we assume that the unnecessary and incorrect entries from the atomic change log data have been removed.

6.1.1 Data cleaning and filtering

The first task in the data preparation step is the *data cleaning and filtering*, which includes filtering and extracting essential features from the ontology change logs and removing the unwanted data. The definition of *unwanted data* may vary depending on the output goals of the change log mining framework. In our case, it includes the exclusion of auxiliary RDF triples, such as those which provide information about the number of change operations and parameters in a change request and triples linking the ACL with the PCL recorded data.

Although the change patterns can be stretched across the session boundaries, we opt for a restrictive approach, where a change pattern is applied completely within a session. In this sense, we divide the sequential changes available in an atomic change log based on their session ids (c.f. Figure 5.4). We used SPARQL queries for extracting the change log sessions.

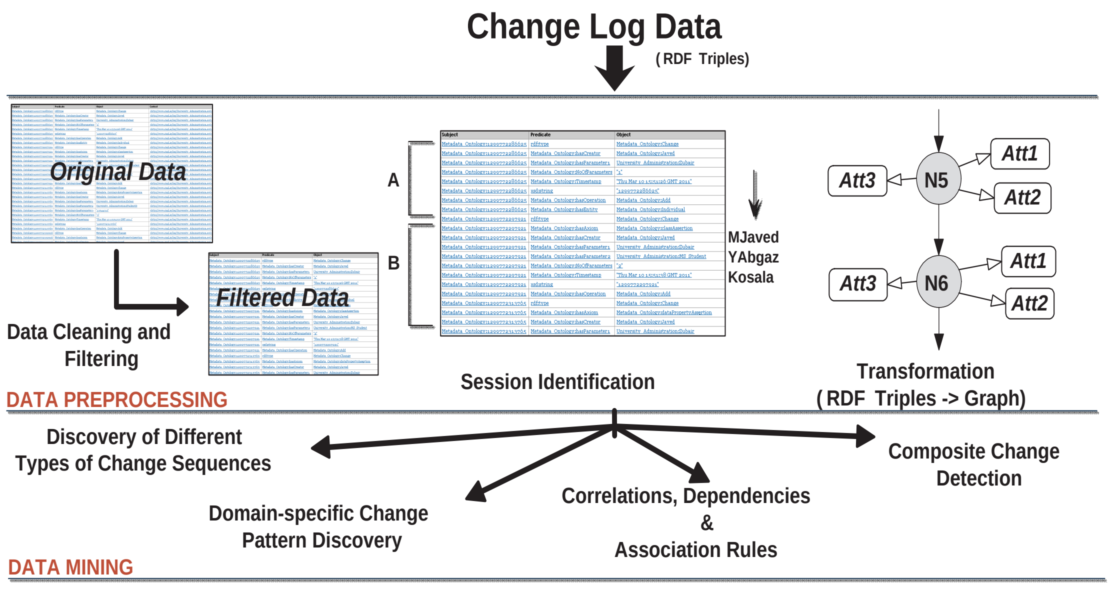


Figure 6.1: Knowledge identification process from ontology change log data

6.1.2 Data transformation

The last step in the knowledge extraction process is the data transformation. Atomic change log data provides operational as well as analytical support in the ontology evolution process. One can mine the higher level change patterns from the ontology change log data. For this purpose, graphs enable efficient search and analysis and can also communicate information visually. We transformed the change log triples into an attributed graph [Ehrig et al., 2004] (formalization is discussed in next section). SPARQL queries are used for filtering the ACL recorded data. Below, three sample SPARQL queries have been given that extract different attributes of an atomic change from the atomic change log.

```
1. select ?x   from <http://www.cngl.ie/log/university.owl>
where {
?x  rdf:type  MO:AtomicChange .
}
```

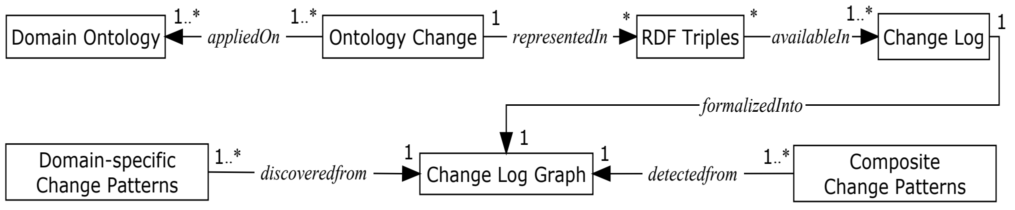


Figure 6.2: Data transformation and mining of higher level change patterns

```
2. select ?c  from <http://www.cngl.ie/log/university.owl>
where {
MO:12383422 rdf:type   MO:AtomicChange .
MO:12383422   MO:hasCreator ?c .
}
3. select ?t  from <http://www.cngl.ie/log/university.owl>
where {
MO:12383422 rdf:type   MO:AtomicChange .
MO:12383422   MO:TimeStamp ?t .
}
```

A linear directed graph (in GraphML[1] format) is then generated from the extracted ACL data using a graph API.

A graph-based formalization is an operational representation for the ontology changes. We considered identifying change patterns from an atomic change log as a problem of recognition of a pattern in a graph. The identification of change patterns from an atomic change log is operationalized in the form of graph-based algorithms (discussed in next two chapters).

[1]http://graphml.graphdrawing.org/primer/graphml-primer.html

6.2 Graph-based ontology change formalization

One of the benefits of the RDF triple format is its fine-grained level representation and interoperability (i.e. conversion from triple format to others standard formats such as RDF and XML). The fine-grained representation of ontology changes helps the ontology engineer to construct complex queries and extract different types of knowledge from the change log. However, as RDF triples represent the ontology changes at fine-grained level (1 ontology change is represented by 8 to 10 triples), visualizing and navigating through the change log alone is time consuming. Graphs can cover this gap. Graph techniques provide the ability to visualize and navigate through large network structures. They enable efficient search and analysis and can also communicate information visually. Moreover, the benefit of a graph-based representation is the availability of well established algorithms/metrics (for pattern discovery and detection) and its well-known characteristics such as performance (for querying the ontology changes effectively). A data warehouse mechanism can be applied here, where the data warehouse collects the operational domain ontologies and the change log data from different distributed locations and reformulates it into a graph on a periodic basis for analytical processing.

A graph-based formalization is an operational representation for the ontology changes. In order to identify the higher level change patterns from the atomic change log, we reformulate the triple-based representation of atomic changes using a graph-based approach. We use attributed graphs [Ehrig et al., 2004]. Graphs with node and edge attribution are typed over an attribute type graph (ATG). Attributed graphs (AG) ensure that all edges and nodes of a graph are typed over the ATG and each node is either a source or target, connected by an edge (Figure 6.3). Each graph node represents an atomic ontology change and the attributes of such graph node provide meta-level and change data details. The benefit of using ATGs and AGs is their similarity with object oriented programming languages, where one can assign each element of the graph a type. Similar to the objects of any class, having a number of class variables, one can attach a number of attributes to a graph node in an AG. The data types of such attributes can be defined in an ATG. Furthermore, one can borrow other object oriented concepts, such as inheritance relations, for any defined element in an ATG.

6.2.1 Formalization

Based on the idea of attributed graphs, a change log graph G can be given as $G = (N_G, N_A, E_G, E_{NA}, E_{EA})$ where:

- $N_G = \{n_g^i | i = 1, \ldots, p\}$ is the set of graph nodes. Each node represents a single ontology change log entry (i.e. representing a single atomic ontology change). The term p refers to the total number of atomic change operations present in the atomic change log. Our overall assumption is that the concurrent ontology change operations (if any) are sequenced, where each ontology change operation is executed one after the other (i.e. sequenced change log).

- $N_A = \{n_a^i | i = 1, \ldots, q\}$ is the set of attribute nodes. Attribute nodes are of two types, i) attribute nodes which symbolize the metadata (e.g. change Id, user, timestamp) and ii) attribute nodes which symbolize the change data and its subtypes (e.g. operation, element, target parameter, auxiliary parameters) - c.f. Figure 5.4. The term q refers to the total number of attribute nodes in a change log graph.

- $E_G = \{e_g^i | i = 1, \ldots, p-1\}$ is the set of graph edges which connects two graph nodes n_g. The graph edges e_g represent the sequence of the ontology change operations in which they have been applied on the domain ontology.

- $E_{NA} = \{e_{na}^i | i = 1, \ldots, r\}$ is the set of node attribute edges which join an attribute node n_a to a graph node n_g. Term r refers to the total number of node attribute edges in a change log graph.

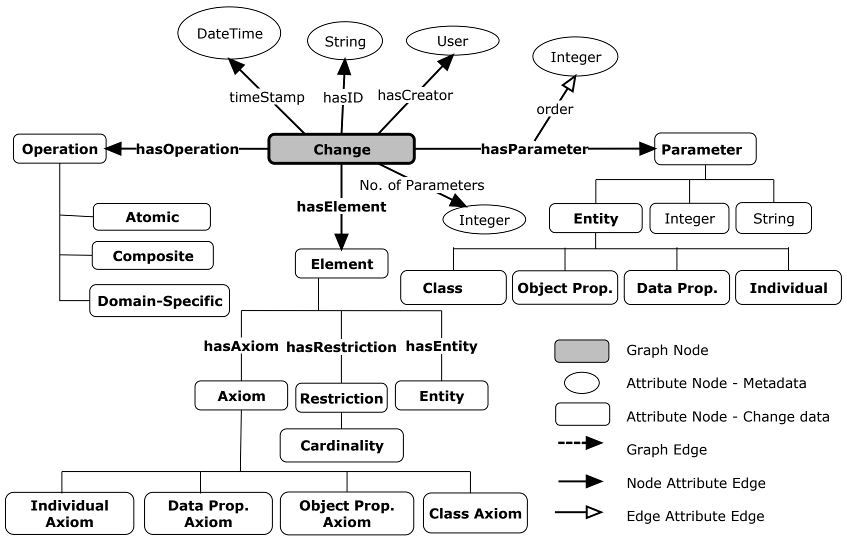


Figure 6.3: Attribute Type Graph (ATG) for an ontology change

- $E_{EA} = \{e^i_{ea} | i = 1, \ldots, s\}$ is the set of edge attribute edges which joins an attribute node n_a to a node attribute edge e_{na}. Term s refers to the total number of edge attribute edges in a change log graph.

Example 6.1: Two graph nodes of an attributed graph (AG) typed over an ATG are given in Figure 6.4. The first graph node, having change Id "1299732463318", represents the atomic change operation `Add NamedIndividual (‘‘John’’)`. The second graph node, having change Id "1299732463423", represents the atomic change operation `Add classAssertion` (`John`, `PhD_Student`). The attribute nodes n_a (representing metadata and change data properties (c.f. Section 5.3.1)) are linked to the graph nodes n_g using node attribute edges e_{na}. The graph nodes are linked to each other using a graph edge e_g, constructing a sequenced ontology change log graph. The types defined on the attribute nodes can be given as $t(Add)$=$Operation$, $t(classAssertion) = Element$, $t(John) = Individual$ and $t(PhD\_Student) = Class$.

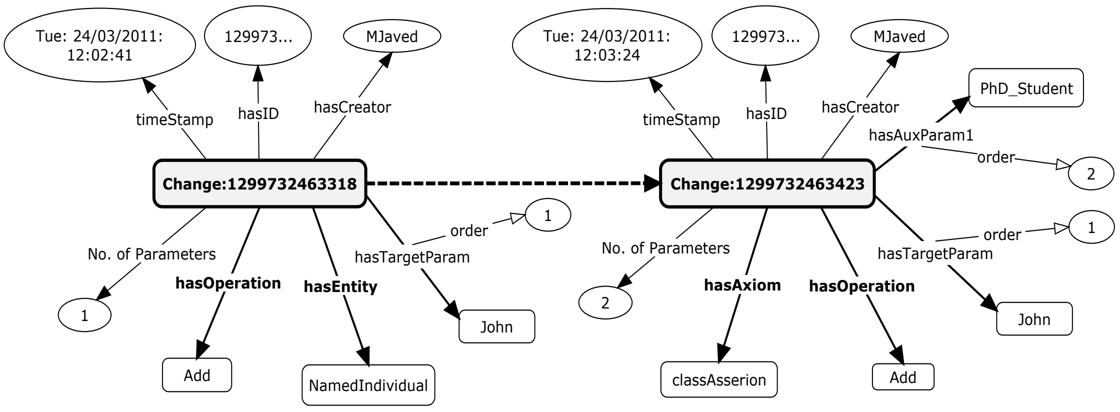


Figure 6.4: Nodes of Attributed Graph (AG) typed over ATG

6.3 Analysis of change log graph

In this section, we discuss our methods to analyze the change log graph by using examples from the *university ontology* case study domain (c.f. Section 4.1.1). A small portion of the university ontology change log graph (in a form of listing) is given in Table 6.1. Each line in the table represents a single graph node (n_g), representing a single atomic change request. The id in each line represents the graph node identification key, representing the order of the change operations in which they have been applied. Note, that the example here was chosen as it can fit on a small scale and metadata attributes (e.g. `sessionId`, `User` and `Timestamp`) attached to each graph node, have not been mentioned. Below we discuss the metrics that lead us to the mining of (generic and domain-specific) change patterns.

6.3.1 Ordered/Unordered change sequences

A *change in an ontology* can be modeled and performed in different ways using different names and different steps [Palma et al., 2009]. In Table 6.1, one can identify multiple occurrences of the process of “PhD student registration”. Such registration processes have been extracted from the change log graph table and are given in the form of a node set in Table 6.2. The numbers in Table 6.2 represent the graph node ids and the order in which they are listed in the change log graph (from left to right). The user performed the identical change by using a different order of atomic change operations at different times. For example, in Table 6.1, sequences s_1 and s_2 have the same order of change operations. First, the user adds the individual x to the domain ontology. In the next step, s/he adds individual x as an instance of class `PhD_Student` and in the last step, s/he adds an individual x as a member of a university department y and assigns a student id. On other hand, change sequence s_4 is unordered with respect to the change sequence s_1. In change sequence s_4, the user assigns a university department to the individual first (node 18) and later adds the individual as an instance of class `PhD_Student` (node 19). As the overall intent behind the two applied change sequences is same, we can say that, although change sequences s_1 and s_4 are structurally different (due to different order of atomic change operations), they are semantically identical.

Table 6.1: A section of university ontology change log graph

	Id	Change Operations (extracted from ontology change log graph)
	01	Add class ("PhD_Student")
	02	Add subclassOf (PhD_Student, Student)
	03	Add NamedIndividual ("Javed")
	04	Add classAssertion (Javed, PhD_Student)
s_1	05	Delete subclassOf (PostGraduate , Student)
	06	Delete class (PostGraduate)
	07	Add objectPropertyAssertion (Javed, isMemberOf , Computing)
	08	Add dataPropertyAssertion (Javed, studentId , "58120348")
	09	Add NamedIndividual (" Yalemisew ")
s_2	10	Add classAssertion (Yalemisew , PhD_Student)
	11	Add objectPropertyAssertion (Yalemisew , isMemberOf , ElectricalEngineering)
	12	Add dataPropertyAssertion (Yalemisew , studentId , "58123857")
	13	Add NamedIndividual (" Kosala ")
	14	Add classAssertion (Kosala , PhD_Student)
s_3	15	Add NamedIndividual (" ECOWS2009 ")
	16	Add objectPropertyAssertion (Kosala , isMemberOf , ElectronicEngineering)
	17	Add NamedIndividual (" Aakash ")
	18	Add objectPropertyAssertion (Aakash , isMemberOf , MechanicalEngineering)
s_4	19	Add classAssertion (Aakash , PhD_Student)
	20	Delete inverseObjectProperty (hasSupervisor , isSupervisorOf)
	21	Add dataPropertyAssertion (Aakash , studentId , "58121143")
c	22	Add domainOfObjectProperty (studentId , Student)
	23	Delete domainOfObjectProperty (studentId , PhD_Student)
	24	Add NamedIndividual ("Wong")
s_5	25	Add dataPropertyAssertion (Wong, studentId , "58129070")
	26	Add classAssertion (Wong, PhD_Student)
	27	Add NamedIndividual (" Pooyan ")
s_6	28	Add classAssertion (Pooyan , PhD_Student)
	29	Add objectPropertyAssertion (Pooyan , isMemberOf , ElectronicEngineering)
	30	Add objectPropertyAssertion (Pooyan , isMemberOf , ElectricalEngineering)

Table 6.2: Node sets for PhD student registration

Seq .	Node Set (*PhD Student Enrolment*)
s_1	*03 - 04- 07 - 08*
s_2	*09 - 10- 11 - 12*
s_3	*13 - 14- 16*
s_4	*17 - 18- 19 - 21*
s_5	*24 - 25- 26*
s_6	*27 - 28- 29 - 30*

6.3.2 Node-distance value

As we discussed in the previous section, users adopt different orders of atomic change operations to perform semantically equivalent tasks. For example, sequence s_4 is a semantically equivalent sequence to s_1. However, due to the different order to atomic change operations in these sequences, comparing their two graph node sequences (i.e. step-by-step graph node comparison), we find a mismatch at step two. The first graph node of sequence s_1 will match to the first graph node of sequence s_4. However, at step two, the second graph node of sequence s_1 will find a mismatch with the second graph node of sequence s_4. In such cases, to find the matching graph node, we need to discard mismatching graph nodes, called "intra-pattern gap node"(c.f. Section 5.4.1), and move to the next nodes of the sequence to find a match, if available. In the above example, a matching graph node will be found at step three of the sequence s_4. Thus, we can say that in comparison to the graph nodes 03 and 04 of sequence s_1, the distance between the type equivalent graph nodes (i.e. nodes 17 and 19) of sequence s_4 is 1 (as there is one additional graph node between them, i.e. node 18). We call it the *node-distance* (or in short *n-distance*), as it refers to the distance (gap) between two adjacent nodes of a sequence in comparison to another change sequence. We adopted the node-distance approach from sequential pattern mining in the biology domain where subsequences are restricted by a predefined gap constraint range [Li et al., 2008, Zhu et al., 2007]. In our case, the gap between two nodes of a sequence is not defined as a constraint but as a permissible and the maximum allowed node-distance between two adjacent nodes is a user-defined value. Defining the node-distance value as a user input allows users to mine the ontology change patterns with zero, one or multiple intra-pattern gaps among the adjacent ontology change operations of a sequence.

Though in our case the permissible node-distance value is a user input value, the node-distance value can also be computed by performing an iterative pattern mining process. For the fixed user input values of pattern support and pattern length, node-distance value can be varied from value 0 to a specific threshold value x such that the increase in value x does not make any difference in the output result. Further, a user may perform the same step in different settings, i.e. for different pattern support and pattern length values.

For a change sequence s in a change log graph G, the node gap between two adjacent nodes can be represented by a series of wild-cards (denoted by symbol x), where a wild-card x is a special symbol that represents an intra-pattern gap node and matches a graph node in the change log graph. Thus, given a change sequence $g = n_1, x_1, x_2 \cdots x_n, n_2, \cdots n_p$, where p is the total number of graph nodes in g and n_1, n_2 are two adjacent graph nodes in an identified graph node change sequence s, the node distance between adjacent graph nodes n_1 and n_2 in change sequence s ($s = n_1, n_2 \cdots n_m$) can be given as $\alpha^{1,2} = |x_1 \cdots x_n|$. For example, sequence s_4 of Table 6.2 can be written as $s_4 = \{n_{17}, n_{18}, n_{19}, x_1, n_{21}\}$ and $\alpha^{19,21} = +1$.

The overall n-distance of the sequence is denoted by upper case alpha A. It represents the number of wild-cards present in the whole sequence, whereas lower case alpha α refers to the distance between two distinct adjacent graph nodes of a sequence. An ordered sequence s with n-distance $A_s = \sum_{i=1}^{m} \alpha_i^{a,b}$ (where a and b refer to the Ids of the adjacent graph nodes and m refers to the number of graph nodes in the change sequence) can be given as $s = (n_1, \alpha_1^{1,2}, n_2, \alpha_2^{2,3}, \cdots, n_m)$. Thus, sequence s_4 in Table 6.2 can also be written as $s_4 = \{n_{17}, +0, n_{18}, +0, n_{19}, +1, n_{21}\}$. Here, '+' sign refers to the order of the

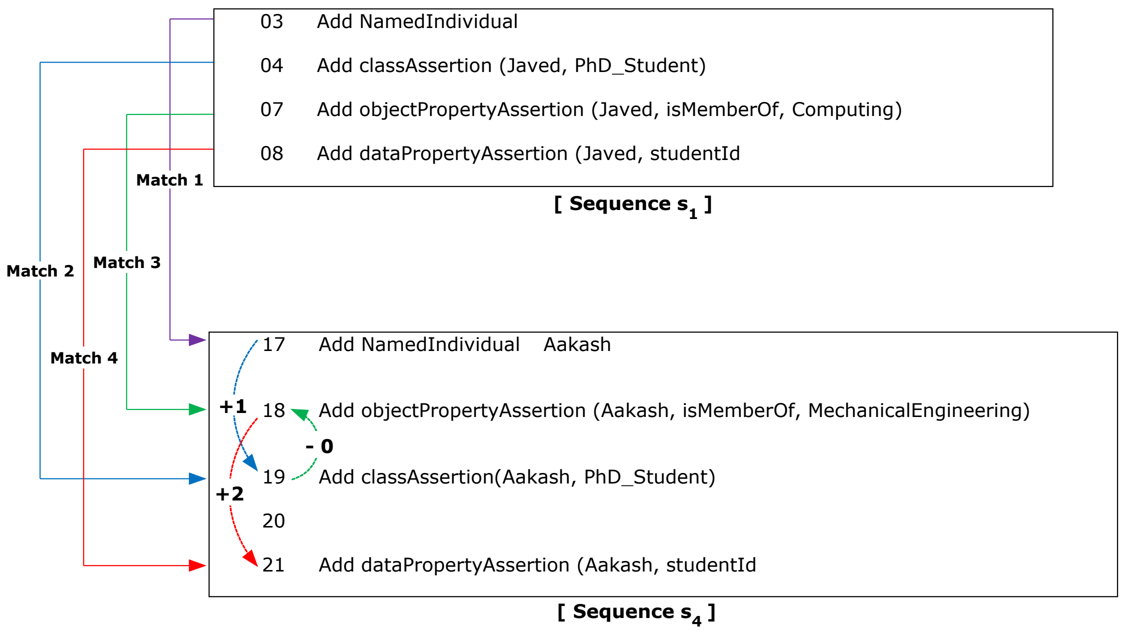

03 Add NamedIndividual
04 Add classAssertion (Javed, PhD_Student)
07 Add objectPropertyAssertion (Javed, isMemberOf, Computing)
08 Add dataPropertyAssertion (Javed, studentId
[Sequence s1]
Match 1
Match 2
Match 3
Match 4
17 Add NamedIndividual Aakash
+1
18 Add objectPropertyAssertion (Aakash, isMemberOf, MechanicalEngineering)
- 0
19 Add classAssertion(Aakash, PhD_Student)
+2
20
21 Add dataPropertyAssertion (Aakash, studentId
[Sequence s4]

Figure 6.5: Step-by-step graph node comparison of sequence s_1 and s_4

graph nodes in a change sequence. Sequence s_4 is an unordered sequence in comparison to the reference sequence s_1. In such a case, the node distance between two adjacent graph nodes can be either positive or negative (Figure 6.5). Thus, in a step-by-step comparison to s_1, sequence s_4 can also be written as $s_4 = \{n_{17}, +1, n_{19}, -0, n_{18}, +2, n_{21}\}$. Here, -0 node-distance means that the node id of the next matched node is one less than the node id of the current graph node. In other words, the position of the next matched node n_{18} in ACL is prior to the current matched node n_{19}.

Node-distance is a measure for capturing the structural differences between two identical change sequences. The structural differences between two semantically identical change sequences are caused by the existence of additional change operations within the sequence (which must be discarded during matching) or by the unordered change operations [Javed et al., 2011b]. The benefit of the metric is clearly visible in the case of change patterns discovery (c.f. Section 8.3.2), where one can map a given change sequence with other identified ordered change sequences with additional change operations or with semantically identical unordered change sequences. The gap nodes, that were discarded earlier, may either be mapped with other subsequent graph nodes of the referenced change sequence (in the case of unordered change sequences) or may be discarded completely (in the case of ordered change sequences) (discussed more in Section 6.4.1).

6.3.3 Type categorization of change operations

Identifying the semantically identical changes of a domain ontology is an important part of the ontology change mapping and clustering [Tury et al., 2006]. We distinguish between type-equivalent and distinct (non-type equivalent) change operations.

Definition 6.1: Two ontology change operations can be type-equivalent (or non-type equivalent) based on the type of their operations (o), elements (e) and parameters (p). Given two atomic change operations $a = (o_a, e_a, p_a)$ and $b = (o_b, e_b, p_b)$, the change operations a and b are distinct if they are type-distinct in at least one of their components, i.e. if $type(o_a) \neq type(o_b)$ or $type(e_a) \neq type(e_b)$ or $type(p_a) \neq type(p_b)$.

The type categorization metrics benefits us in terms of measuring the variations between two change sequences (discussed in Section 6.3.4). Below, three change operations are given.

```
1- Add DataPropertyAssertion(John, hasTitle, ''Prof.'')
2- Add DataPropertyAssertion(Conor, hasTitle, ''Engr.'')
3- Add DataPropertyAssertion(RefBook, hasTitle, ''Intro to Java'')
```

Each change operation instantiates a data property "hasTitle" for a certain individual. Here, operations 1 and 2 are type-equivalent as both of them have the same operation type (Add), the same element type (DataPropertyAssertion) and the same type of individuals (i.e. `John` and `Conor` are instances of class `Person`). Operation 3 is distinct in comparison to operations 1 and 2, as parameter `RefBook` is not an instance of class `Person`, but of class `Book`, which does not exist in the class hierarchy of `Person`.

6.3.4 Variation between change sequences

Two change sequences can be different from each other in terms of their length, order or type of operations involved. We identified three types of variation which can occur between two change sequences, i.e. Len-variation, ST-variation and DT-variation.

Len-Variation: The length variation (Len-variation) captures the variation between two change sequences based on the number of graph nodes present in them.

Definition 6.2: Given two change sequences $s = s_1, s_2 \cdots s_n$ and $t = t_1, t_2 \cdots t_m$, the Len-variation of change sequence s in relation to t can be given as $Len(s,t) = n - m$, where n and m are the number of graph nodes in change sequences s and t, respectively.

Example 6.2: In Table 6.2, the Len-variation of change sequence s_1 in relation to s_3 is $Len(s_1, s_3) = (4 - 3) = +1$. The '+' sign refers that the latter sequence is shorter (in length) in relation to the former sequence and a '-' sign refers that former sequence is shorter (in length) in relation to the latter sequence.

ST-Variation: The same-type variation (ST-variation) is a measure to capture the differences between two change sequences based on the sets of type-equivalent change operations that exist in both change sequences, but in different numbers (see Figure 6.6).

Definition 6.3: Given two change sequences $s = s_1, s_2 \cdots s_n$ and $t = t_1, t_2 \cdots t_m$, the ST-variation can be given as

$$ST(s,t) = \sum_{i=1}^{type\_max} |s_i^{ST} \backslash t_i^{ST}| \tag{6.1}$$

We describe the above given equation with an example. The change operations of the two change sequences (s and t) along with their types can be given as,

$$s = s_1 : type_1, s_2 : type_2, s_3 : type_2, s_4 : type_3, s_5 : type_4 \tag{6.2}$$

$$t = t_1 : type_1, t_2 : type_1, t_3 : type_2, t_4 : type_4, t_5 : type_5 \tag{6.3}$$

Let, s^{ST} be a type-sorted subset of types of s with $type(s_i)$ in $types(t)$ and t^{ST} be a type-sorted subset of types of t with $type(t_i)$ in $types(s)$. Based on the change sequences given in equation 6.2 and 6.3, the s^{ST} and t^{ST} can be given as

$$s^{ST} = \{\{type_1{}^{s_1}\}, \{type_2{}^{s_2}, type_2{}^{s_3}\}, \{type_4{}^{s_5}\}\}$$

$$t^{ST} = \{\{type_1{}^{t_1}, type_1{}^{t_2}\}, \{type_2{}^{t_3}\}, \{type_4{}^{t_4}\}\}$$

The set difference $s^{ST} \backslash t^{ST}$ (i.e. complement of t^{ST} in s^{ST}) can be identified as

$$s_{type_1} \backslash t_{type_1} = \{\{type_1{}^{s_1}\} \backslash \{type_1{}^{t_1}, type_1{}^{t_2}\}\} = \{\}$$

$$s_{type_2} \backslash t_{type_2} = \{\{type_2{}^{s_2}, type_2{}^{s_3}\} \backslash \{type_2{}^{t_3}\}\} = \{type_2{}^{s_3}\}$$

$$s_{type_4} \backslash t_{type_4} = \{\{type_4{}^{s_5}\} \backslash \{type_4{}^{t_4}\}\} = \{\}$$

Thus, $ST(s,t) = |\{\}| + |\{type_2{}^{s_3}\}| + |\{\}| = 0 + 1 + 0 = 1.$

Similarly, the set difference $t^{ST} \backslash s^{ST}$ can be given as

$$t_{type_1} \backslash s_{type_1} = \{\{type_1{}^{t_1}, type_1{}^{t_2}\} \backslash \{type_1{}^{s_1}\}\} = \{type_1{}^{t_2}\}$$

$$t_{type_2} \backslash s_{type_2} = \{\{type_2{}^{t_3}\} \backslash \{type_2{}^{s_2}, type_2{}^{s_3}\}\} = \{\}$$

$$t_{type_4} \backslash s_{type_4} = \{\{type_4{}^{t_4}\} \backslash \{type_4{}^{s_5}\}\} = \{\}$$

Thus, $ST(t,s) = |\{type_1{}^{t_2}\}| + |\{\}| + |\{\}| = 1 + 0 + 0 = 1.$

Example 6.3: In Table 6.2, the type-sorted subsets $s_1{}^{ST}$ and $s_6{}^{ST}$ can be given as

$$s_1{}^{ST} = \{\{type_1{}^{n_3}\}, \{type_2{}^{n_4}\}, \{type_3{}^{n_7}\}\}$$

$$s_6{}^{ST} = \{\{type_1{}^{n_{27}}\}, \{type_2{}^{n_{28}}\}, \{type_3{}^{n_{29}}, type_3{}^{n_{30}}\}\}$$

The set difference $s_1{}^{ST} \backslash s_6{}^{ST}$ can be identified as

$$s_{1type_1} \backslash s_{6type_1} = \{\{type_1{}^{n_3}\} \backslash \{type_1{}^{n_{27}}\}\} = \{\}$$

$$s_{1type_2} \backslash s_{6type_2} = \{\{type_2{}^{n_4}\} \backslash \{type_2{}^{n_{28}}\}\} = \{\}$$

$$s_{1type_3} \backslash s_{6type_3} = \{\{type_3{}^{n_7}\} \backslash \{type_3{}^{n_{29}}, type_3{}^{n_{30}}\}\} = \{\}$$

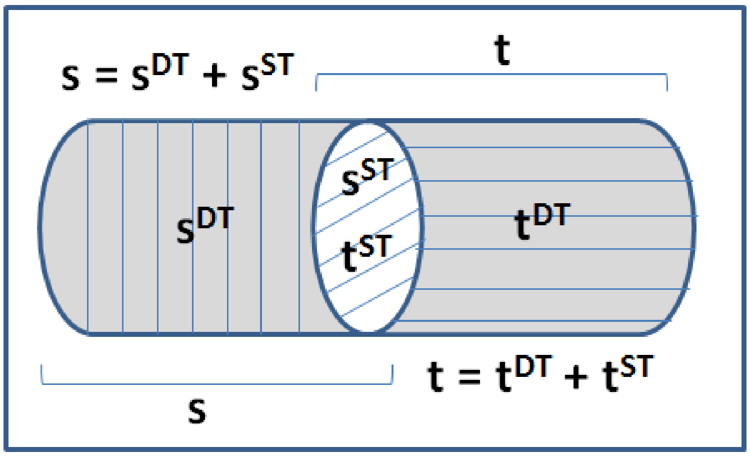


Figure 6.6: Type sets of s and t

Thus, $ST(s_1, s_6) = |\{\}| + |\{\}| + |\{\}| = 0 + 0 + 0 = 0.$

On the other hand, the set difference ${s_6}^{ST} \backslash {s_1}^{ST}$ can be identified as

$$s_{6type_1} \backslash s_{1type_1} = \{\{{type_1}^{n_{27}}\} \backslash \{{type_1}^{n_3}\}\} = \{\}$$

$$s_{6type_2} \backslash s_{1type_2} = \{\{{type_2}^{n_{28}}\} \backslash \{{type_2}^{n_4}\}\} = \{\}$$

$$s_{6type_3} \backslash s_{1type_3} = \{\{{type_3}^{n_{29}}, {type_3}^{n_{30}}\} \backslash \{{type_3}^{n_7}\}\} = \{{type_3}^{n_{30}}\}$$

Thus, $ST(s_6, s_1) = |\{\}| + |\{\}| + |\{{type_3}^{n_{30}}\}| = 0 + 0 + 1 = 1.$

DT-Variation: The distinct-type variation (DT-variation) captures the differences between two change sequences based on the sets of distinct (non-type equivalent) change operations that are present in one change sequence, but are missing from the other.

Definition 6.4: Given two change sequences $s = s_1, s_2 \cdots s_n$ and $t = t_1, t_2 \cdots t_m$, the DT-variation of change sequence s in relation to change sequence t can be given as $DT(s, t) = |s \backslash t|$. Here $s \backslash t$ refers to the set difference of s and t. In comparison to change sequence t, s is actually a combination of change operations that exist or do not exist in t, and vice versa (see Figure 6.6). Hence, the DT-variations can also be given

$DT(s,t) = |s - s^{ST}|$ and $DT(t,s) = |t - t^{ST}|$.

For explanation, we make use of changes sequences given above in equations 6.2 and 6.3. Let, s^{DT} be a type-sorted subset of types of s with $type(s_i)$ not in $types(t)$ and t^{DT} be a type-sorted subset of types of t with $type(t_i)$ not in $types(s)$. The s^{DT} and t^{DT} can be given as

$$s^{DT} = \{type_3{}^{s_4}\}$$
$$t^{DT} = \{type_5{}^{t_5}\}$$

The set difference $s^{DT} \backslash t^{DT}$ (i.e. complement of t^{DT} in s^{DT}) can be given as

$s^{DT} \backslash t^{DT} = \{type_3{}^{s_4}\}$ and $DT(s,t) = 1$.

Similarly, the set difference $t^{DT} \backslash s^{DT}$ can be given as

$$t^{DT} \backslash s^{DT} = \{type_5{}^{t_5}\} \text{ and } DT(t,s) = 1.$$

Example 6.4: In Table 6.2, the ${s_1}^{DT}$ and ${s_6}^{DT}$ can be given as

$${s_1}^{DT} = \{type_4{}^{n_8}\}$$
$${s_6}^{DT} = \{\}$$

The set difference ${s_1}^{DT} \backslash {s_6}^{DT}$ can be given as

$$s_1^{DT} \backslash s_6^{DT} = \{type_4{}^{n_8}\} \text{ and } DT(s_1, s_6) = 1.$$

On the other hand, the set difference ${s_6}^{DT} \backslash {s_1}^{DT}$ can be given as

$$s_6^{DT} \backslash s_1^{DT} = \{\} \text{ and } DT(s_6, s_1) = 0.$$

6.4 Mining of sequential abstractions

Several data mining methods are used to discover the implicit information in the log data, especially in web log data [Ivancsy et al., 2006, Pabarskaite et al., 2007, Yu, 2009, Jiang et al., 2010, Agostiet al., 2007]. In this section, we discuss the mining of sequential abstractions from the preprocessed change log data. Based on our analysis of change log graph data, we categorize the ontology change sequences in two basic divisions and use them as the basis for change pattern discovery algorithms. Furthermore, we exploit the change log graph to detect the generic composite change patterns across the ontology taxonomy. Here, we distinguish between the terms *"discovery"* and *"matching"*. The term *"change pattern discovery"* refers to identifying the change patterns from the change log graph without having any prior knowledge about them and is based on their size and frequency of occurrence. The term *"change pattern matching"* refers to identifying a pre-defined (existing) change pattern from the ontology change log graph. Here, Sections 6.4.2 and 6.4.3 provide only an overview of the approaches and details are given in Chapter 7 and 8.

6.4.1 Identification of change sequences

As we have seen in Table 6.1, users can use a different order of atomic change operations to perform the task "PhD student registration". For change sequences s_1 and s_2, the user first adds a newly created individual as an instance of class `PhD_Student`, assigns a university department to him/her and then assigns a student Id. Comparing these change sequences with sequence s_4, the user first assigns the university department to the newly created individual and later make it an instance of class `PhD_Student`. This makes the change sequence s_4 an unordered change sequence in comparison to the change sequences s_1 and s_2.

Based on our ordered/unordered change sequence observation (c.f. Section 6.3.1), we identified four types of the change sequences (in comparison to a referenced candidate sequence) based on the ordering of the graph nodes and completeness (i.e. *Len-Variation*). We merged these different types of change sequences into two basic divisions:

- ***Ordered change sequences (OS)***

  Type 1 - Ordered complete change sequence (OCS)

  Type 2 - Ordered partial change sequence (OPS)

- ***Unordered change sequences (US)***

Type 3 - Unordered complete change sequence (UCS)

Type 4 - Unordered partial change sequence (UPS)

6.4.1.1 Ordered change sequences (OS)

OSs comprise ordered graph node change sequences from a change log graph. Such change sequences may only have a positive node distance between two adjacent graph nodes (w.r.t. a referenced change sequence). Ordered change sequences can be complete (OCS) or partial (OPS).

- *Ordered complete change sequence (OCS)*: Given two graph node change sequences a and $\bar{a}$, the change sequence $\bar{a}$ is an ordered complete change sequence in relation to the referenced change sequence a, if

  i. each graph node of the change sequence $\bar{a}$ is type-equivalent to the *same indexed graph node* of the change sequence a (i.e. $DT(\bar{a}, a) = 0$, $ST(\bar{a}, a) = 0$ and $\alpha = +ve$ value) and

  ii. both change sequences have the *same number of graph nodes* (i.e. $Len(\bar{a}, a) = 0$).

Definition 6.5: Given two change sequences $a = a_1, a_2, \cdots a_p$ and $\bar{a} = \bar{a}_1, \bar{a}_2, \cdots \bar{a}_q$, the change sequence $\bar{a}$ is an OC-match of change sequence a, if $q = p$ and there exist integers $1 \leq j_1 < j_2 \cdots j_q \leq p$ such that $a_1 \equiv \bar{a}_{j1}, a_2 \equiv \bar{a}_{j2}, \cdots a_p \equiv \bar{a}_{jq}$. Here $\equiv$ refers to a type equivalence between two graph nodes [Spiliopoulos et al., 2010].

Example 6.5: In Table 6.2, change sequence s_2 is an OC-match of change sequence s_1, as the sequence s_2 is *complete* (i.e. $Len(\bar{a}, a) = 0$) and type-equivalent change operations are in the *same order* (i.e. $\alpha = +ve$ value).

- *Ordered partial change sequence (OPS)*: Given two graph node change sequences a and $\bar{a}$, the change sequence $\bar{a}$ is an ordered partial change sequence (OPS) in relation to the referenced change sequence a, if

    i. each graph node of the change sequence $\bar{a}$ is type-equivalent to the *same indexed graph node* of the change sequence a (i.e. $DT(\bar{a}, a) = 0$, $ST(\bar{a}, a) = 0$ and $\alpha = +ve$ value) and

    ii. change sequence $\bar{a}$ is a *subset* of the change sequence a (i.e. $Len(\bar{a}, a) \neq 0$).

Definition 6.6: Given two change sequences $a = a_1, a_2, \cdots a_p$ and $\bar{a} = \bar{a}_1, \bar{a}_2, \cdots \bar{a}_q$, the change sequence $\bar{a}$ is an OP-match of change sequence a, if $q < p$ and there exist integers $1 \leq j_1 < j_2 \cdots j_p \leq q$ such that $\bar{a}_1 \equiv a_{j1}, \bar{a}_2 \equiv a_{j2}, \cdots \bar{a}_q \equiv a_{jp}$.

Example 6.6: In Table 6.2,, sequence s_3 is an OP-match of change sequence s_1, as the type-equivalent graph nodes in change sequence s_3 are in the *same order* as change sequence s_1 (i.e. $\alpha = +ve$ value), but the sequence is *partial* (i.e. $Len(\bar{a}, a) \neq 0$) (due to the absence of a change operation assigning a student id to the PhD student).

6.4.1.2 Unordered change sequences (US)

USs comprise unordered graph node change sequences from a change log graph. These change sequences (complete or partial) may have positive or negative node distances between two adjacent graph nodes (w.r.t the referenced change sequences).

- *Unordered complete change sequence (UCS)*: Given two graph node change sequences a and $\bar{a}$, the change sequence $\bar{a}$ is an unordered complete change sequence in relation to the referenced change sequence a, if

i. each graph node of the change sequence $\bar{a}$ is type equivalent to *one of the graph nodes* in the change sequence a (i.e. $DT(\bar{a}, a) = 0$ and $\alpha = +ve/-ve$ value) and

ii. both change sequences have the *same number of graph nodes* (i.e. $Len(\bar{a}, a) = 0$).

Definition 6.7: Given two change sequences $a = a_1, a_2, \cdots a_p$ and $\bar{a} = \bar{a}_1, \bar{a}_2, \cdots \bar{a}_q$, the change sequence $\bar{a}$ is an UC-match of change sequence a, if $p = q$ and $(\bar{a}_i \in \bar{a} \Rightarrow a_j \in a$ with $\bar{a}_i \equiv a_j)$, where $1 \leq i \leq q$, $1 \leq j \leq p$.

Example 6.7: In Table 6.2, sequence s_4 is a UC-match of change sequence s_1, as s_4 is *complete* (i.e. contains all the type-equivalent graph nodes in relation to change sequence s_1) and the graph nodes are *unordered* (i.e. $\alpha = +ve/-ve$ value).

- *Unordered partial change sequence (UPS)*: Given two graph node change sequences a and $\bar{a}$, the change sequence $\bar{a}$ is an unordered partial change sequence in relation to the referenced change sequence a, if

  i. each graph node of the change sequence $\bar{a}$ is type equivalent to *one of the graph nodes* in the change sequence a (i.e. $DT(\bar{a}, a) = 0$ and $\alpha = +ve/-ve$ value) and

  ii. change sequence $\bar{a}$ is a *subset* of the change sequence a (i.e. $Len(\bar{a}, a) \neq 0$).

Definition 6.8: Given two change sequences $a = a_1, a_2, \cdots a_p$ and $\bar{a} = \bar{a}_1, \bar{a}_2, \cdots \bar{a}_q$, the change sequence $\bar{a}$ is an UP-match of change sequence a, if $p > q$ and $(\bar{a}_i \in \bar{a} \Rightarrow a_j \in a$ with $\bar{a}_i \equiv a_j)$, where $1 \leq i \leq q$, $1 \leq j \leq p$.

Example 6.8: In Table 6.2, change sequence s_5 is an UP-match of change sequence s_1, as the change operations are in *different order* as well as the change sequence is *partial*.

The identification of the above mentioned change sequences has a number of benefits. First, it helps documenting evolving ontologies, i.e. representing how entities evolve over time (entity evolution). Second, these change sequences can be used to discover the correlations and the causal dependencies between different ontological entities which evolve together. Third, and most importantly, the identified change sequences can be used in discovering usage-driven change patterns.

6.4.2 Detection of composite change patterns

Many evolution tasks cannot be done by applying a single atomic change operation on a domain ontology. A set of related atomic change operations is required. In this sense, composite change operations are the aggregated changes to represent a composite task. Composite change patterns are represented in the atomic change log (ACL) as an ordered/unordered list of atomic level change operations. Composite change patterns are generally applied at the entity level and, thus, are generic (non domain-specific). As different atomic level change operations can be combined together in different order, providing an exhaustive list of composite level change patterns is not feasible. `Split class`, `Pull up property`, `Group classes` are examples of commonly used generic composite change patterns.

The composite changes can be mined from the atomic change log using a pattern matching approach. Here, the term "pattern matching" refers to the mining of occurrences of a pre-defined change pattern sequence from an atomic change log. The aim here is the identification of already defined composite change patterns [Stojanovic, 2004, Klein, 2004] from the atomic change log. Identifying the composite changes from the atomic change log gives an ontology engineer an indication about the intent of the applied changes. Once a user has a clear understanding of semantics of a change, s/he can select appropriate evolution strategy [Javed et al., 2012a] for consistent ontology change management. One may find (complete or partial) overlapping among the mined change patterns (c.f. Section 5.4.2). This is due to the possibility that a subset of a change pattern satisfies the conditions (to be identified) of another change pattern.

Our solution is based on the identification of a graph node change sequence that matches a referenced composite change sequence. In the next step, we ensure that the identified graph nodes in a candidate change sequence fulfill the conditions of the referenced composite change. Here, the term *conditions* refers to the existence and the correlations among the change parameters (discussed in detail in Section 7.1). The correlations, among the change operations of the change log, are not visible at atomic level and are defined at a higher level of abstraction. For example, by identifying two atomic change operations that replace the domain class of an object property are not enough to declare it as a detection of "Pull up property" composite change. To confirm it, one needs to ensure that the newly added domain class is actually a superclass of the previous domain class (i.e. correlation among the parameters of the two atomic change operations).

A candidate change sequence s_c is a detected composite change pattern if each of the graph nodes in the candidate change sequence s_c has a type equivalent graph node in the referenced change sequence s_r and the candidate change sequence s_c fulfills all the correlation conditions ϕ_c (c.f. Section 7.1) defined for the referenced composite change sequence s_r.

Definition 6.9: Given two graph node change sequences, candidate change sequence $s_c = n_g^1, n_g^2, \cdots, n_g^p$ and referenced change sequence $s_r = (\bar{n}_g^1, \bar{n}_g^2, \cdots, \bar{n}_g^q)$, the change sequence s_c is a detected (ordered/unordered) composite change pattern w.r.t the reference change sequence s_r if, $n_g^1 \equiv \bar{n}_g^x, n_g^2 \equiv \bar{n}_g^x \cdots n_g^m \equiv \bar{n}_g^x$, where $p = q$ and ϕ_c are satisfied.

Detection of composite change patterns supports the representation of the intent of the changes at a higher level. Below in Table 6.3, for the detection of the `Pull up property` composite change pattern, a referenced change sequence s_r and the correlation condition ϕ_c for the referenced change sequence are given. In order to detect the `Pull up property` composite change pattern from the change log graph, the given referenced change sequence and the conditions are given as an input to the composite change detection algorithm. The algorithm detects the type equivalent candidate graph nodes from the change log graph and ensures that the conditions are satisfied by the identified candidate change sequence. For example, in Table 6.1, change sequence c, consisting of graph nodes 22 and 23, is actually a "`Pull up property` (`studentId`, `PhD_Student`, `Student`)" composite change operation, where class `PhD_Student` is a subclass of `Student`. The algorithms for the composite change pattern detection are discussed in detail in Chapter 7.

Table 6.3: Referenced change sequence and conditions for `Pull up property`

s_r	- Add domainOfObjectProperty (P, X_1) - Delete domainOfObjectProperty (P, X_2)
ϕ_r	- X_2 subClassOf X_1

6.4.3 Discovery of domain-specific change patterns

An atomic change log can also be utilized to discover the new domain-specific change patterns using a pattern discovery approach. Such usage-driven domain-specific change patterns provide guidelines to content change management systems in a domain and support in the evolution process [Gruhn et al., 1995]. The discovered domain-specific change patterns can also be utilized in the future as once-off change pattern specifications that can be instantiated whenever a user needs to apply similar changes on the underlying domain ontology. Here, the term "pattern discovery" (as opposed to "pattern matching") refers to the mining of a change pattern from the atomic change log without having any prior knowledge about them. One of the main objective here is, not only to capture the recurrent change subsequences from the ACL that are applied in the same order (i.e. *structurally* identical change patterns), but also the change subsequences that may contain different orders of change operations, but have the same effect on the domain ontology (i.e. *semantically* identical change patterns).

We consider identifying domain-specific change patterns from the atomic change log as a problem of recognizing frequent patterns from a graph [Kuramochi et al., 2001, De Leenheer et al., 2007, Rudolf et al., 2000, Yan et al., 2002]. The discovery of change patterns from the ontology change log graph has been formalized in the form of discovery algorithms (discussed in detail in Chapter 8). There are two main criteria used in the discovery of domain-specific change patterns, i.e. sequence support and length. The *support (supp)* of a change sequence s refers to the number of occurrences of ordered/unordered complete change sequences (in the change log graph) that are type equivalent to the change sequence s, whereas the *length (len)* of a change sequence s refers to the number of graph nodes in such change sequence.

Definition 6.10: Given a candidate change sequence $s_c = n_g^1, n_g^2, \cdots, n_g^p$ (where p is the number of graph nodes in sequence s_c), the change sequence s_c is an instantiation of a discovered change pattern if i) the length of the change sequence s_c and each identified type equivalent change sequence s_t (i.e. $s_t \equiv s_c$) is equal to or greater than the threshold value set for the pattern length len_{min}, (i.e. $len(s_c) \geq len_{min}$) and ii) the support of the candidate change sequence s_c is above the threshold value set for the pattern support of a pattern $supp_{min}$, (i.e. $supp(s_c) \geq supp_{min}$).

Example 6.9: Now, we illustrate the above given formalization using an example from Table 6.1. Let the minimum change pattern support ($supp_{min}$) and the minimum change pattern length (min_{len}) be 2 and 4, respectively, and the change sequence s_1 make the candidate change sequence. In this case scenario, the change sequence s_1 will be selected as a change pattern instantiation because

- the change sequences s_2 and s_4 are identified as (ordered/unordered) type equivalent change sequences (i.e. the $supp(s_1) = 2$) and
- the length of each change sequence is equal to the minimum length threshold value (i.e. $len(s_1)$, $len(s_2)$ and $len(s_4) \geq 4$).

The basic idea of the domain-specific change pattern discovery algorithms is to i) start an iteration process on each graph node, ii) construct the candidate change pattern sequence (s_c) starting from that particular graph node and iii) search the ordered/unordered type equivalent change sequences (s_t) within the change log graph. The discovered change patterns are based on the operations that have been utilized frequently by the user and reduce the effort (in terms of time) required to apply similar change operations on the domain ontology. Once the change patterns are associated with the user category, patterns will be more effective since classified patterns are often more useful [Pinto et al., 2001].

6.5 Summary

Activity log mining is not restricted to creating new formal process models [Jiang et al., 2010, Pabarskaite et al., 2007, Tao et al., 2000] but can be extended to extract other implicit knowledge. Ontology change logs play a significant role and can provide operational as well as analytical support in the ontology evolution process. In this sense, atomic change log data can be re-used to capture change patterns, frequently evolving areas of the ontology and implicit dependencies between ontological entities.

In this chapter, we discussed the different steps taken in order to identify the implicit knowledge from the atomic change log. We adopted a graph mining approach to identify knowledge from the atomic change log. As the ontology changes are stored in the form of RDF triples in an atomic change log, we first discarded the unwanted RDF triples and extracted the required change log triples only. Second, the change log triples are then formalized into a change log graph. We selected the attributed graph here that is typed over an attribute type graph. The benefit of attribute type graphs is its similarity with the object oriented programming (OOP) formalism where one can attach a number of attributes (i.e. *class variables*) to each node (i.e. *object*) of the attributed graph (i.e. *class*).

We utilized ontology change log graphs to empirically identify different types of change sequences. These identification of change sequences lead us to the identification of pre-defined generic (i.e. composite) and usage-driven (i.e. domain-specific) change patterns. The detected composite change patterns can be utilized for ontology change logging at a higher level of granularity with a clear representation of the intent of changes and for a better ontology change management in ontology evolution process. The discovered domain-specific change patterns can be used as pre-defined ontology change operators (to perform similar tasks in the future), pattern redesign (in case, similar change patterns already exist), documentation of changes at a higher level, classification of ontology users etc. The algorithms for identification of composite and domain-specific change patterns are given in Chapter 7 and 8, respectively.

Chapter 7

Composite change detection algorithms

Ontology change log data is a valuable source of information that reflects the changes in the domain, the user requirements, flaws in the initial design or the need to incorporate additional information [Haase et al., 2003]. Ontology change logs can play a significant role and can provide operational as well as analytical support in the ontology evolution process. Representing the ontology changes at a higher level is beneficial as it is more concise, more intuitive and the intent of change is more visible [Papavassiliou et al., 2009]. The composite change operations provide more explicit information about how an ontology changes as well as the specific reasons and consequences of the change operations. Higher level composite change operators are more powerful since an ontology engineer does not have to go through each step of the atomic change in order to achieve the desired effect [Stojanovic, 2004].

Representation of ontology changes at a higher level also help in certifying the validity[1] of the instances at any specific time. For example, it would be more useful for an

[1]The term *validity of instances* indicates that certain individuals, available in the ontology, can consistently be inferred as instances of a specific class.

ontology engineer to know that a domain of an object property is generalised to a higher class in the class hierarchy than to know that domain of a property is detached from one class and is attached to another. In this case, knowing the semantics of a change (through higher level composite change description) one can assure that the validity of the instances is not violated [Klein, 2004].

We consider the composite change operations as pre-defined generic change patterns. In this chapter, we give a graph-based specification of composite ontology changes and present the composite change pattern detection algorithms. We discuss how we exploited a graph transformation approach and utilized it for a graph-based composite change specification. Identification of composite change patterns from an ontology change log

- helps in formulating the ontology change log data in a more concise manner.
- assists in realizing the intuition behind any applied change.
- facilitates in realizing the consistency of an evolving domain ontology.

The chapter is structured as follows: In Section 7.1, we give a formal definition of a composite change. We discuss the graph-based specification of a composite change (using a graph transformation approach) in Section 7.2. In Section 7.3, we discuss how we adapt the graph transformation approach to our needs. We present a composite change scenario for a detailed explanation. The composite change detection algorithms are given in Section 7.4. We end with a brief summary of the chapter.

7.1 Composite change

A composite change is a sequence containing a group of atomic (level one) change operations that are applied on a domain ontology, where the change operations can be of inclusion or exclusion type. The inclusion type change operations add new knowledge to the domain ontology, whereas the exclusion type change operations remove some knowledge from the domain ontology. Thus, a composite change c can be given as a sequence of s_i, where each s_i is either

- an (atomic level) exclusion change operation (δ_1) or (exclusively)

- an (atomic level) inclusion change operation (δ_2).

An exclusion change operation deletes a certain axiom from the target ontology entity and an inclusion change operation adds some new axiom for the target ontology entity. These exclusion and inclusion change operations can be applied in different order - making the composite changes in ordered/unordered form in relation to a reference composite change operation sequence.

In terms of detection of a composite change from atomic change log, i.e. to consider a group of (add/delete) atomic change operations as a composite change, the change operations must satisfy certain conditions ϕ. The term ϕ refers to the *conditions* on the existence of any knowledge in the ontology. Such conditions can either be existential conditions (ϕ_e) or correlations (ϕ_c) among the parameters of the composite change operations. The existential conditions (ϕ_e) of any ontology change operation can be given in terms of pre and post conditions [Stojanovic et al., 2003]. For example, in case of change operation `Add class` (`Researcher`), `Researcher` must not exist in the current version (O_1) of the ontology (as a class) and must exist (as a class) in the next version of the ontology (O_2).

- Pre-Cond: (`Researcher` rdf:type owl:Class) $\notin O_1$

- Post-Cond: (`Researcher` rdf:type owl:Class) $\in O_2$

Next, for change operation `Add subclassOf` (`Researcher` , `Person`), class `Researcher` (and `Person`) must exist in the current version of ontology (Post-Cond (Op_1) $\implies$ Pre-Cond (Op_2)) and `Researcher` should be a subclass (rdfs:subClassOf) of `Person` in the subsequent version of the ontology (O_2).

- Pre-Cond: (`Researcher` rdf:type owl:Class) $\in O_1$
  (`Person` rdf:type owl:Class) $\in O_1$
- Post-Cond: (`Researcher` rdfs:subClassOf `Person`) $\in O_2$

The correlations (ϕ_c) refer to the relationships among the parameters of the existing atomic change operations and works as invariants during the operationalization of a composite change. Such relationships are not explicitly given in the atomic change log. For example, in case of composite change operation `Pull up class` (`Researcher`, `Student`), where the class `Researcher` is being pulled up in the class hierarchy and becomes a sibling class to its previous parent `Student`, the change is actually a group of two atomic change operations, i.e.

- `Delete subclassOf(Researcher, Student).` (δ_1)
- `Add subclassOf(Researcher, Person).` (δ_2)

the invariant correlation can be given as

- `Student subclassOf Person.` (ϕ_c)

If the above given correlation among the parameters is satisfied, we can consider the given two atomic change operations as a `Pull up class` composite change. We utilized the given definition of a composite change in defining the graph transformation rules and the conditions. In other words, we can say that a source ontology subgraph has been transformed into a target ontology subgraph (by applying a composite change) based on the given conditions, i.e. existential and correlation conditions.

7.2 Graph-based specification of a composite change

We specify the composite ontology changes using a graph transformation approach where a source ontology subgraph is transformed into a target ontology subgraph, while preserving the defined conditions. Below, we present the graph-based specification of an ontology and a composite change one after the other.

7.2.1 Graph-based ontology specification

OWL 2.0 structural specification can be found in Chapter 2. Here, we are mainly interested in representing how different ontology entities (i.e. classes, object properties, data properties and individuals) are linked to each other in OWL 2.0, constructing a domain ontology hierarchy. Similar to the approach adopted by few researchers [D'Aquin et al., 2007, Mitra et al., 2000, Patil et al., 2004, De Leenheer et al., 2007], we use directed typed graphs (where ontology nodes and edges are labeled) to represent the ontology entity relationships (Figures 7.1 - 7.2). A *class* is the central element of the ontology hierarchy. Different classes link each other using the subclassOf relationship, constructing a *class hierarchy.* The features (characteristics) of an ontology class are given using object and data properties. Each object property links two classes using a domainOf and rangeOf relationship. For example, the classes `PhD_Student` and `Faculty` can be domain and range (respectively) of the object property `hasSupervisor`. The data property links an ontology class to an XML schema datatype. For example, the class `PhD_Student` can be domain of the data property `hasFirstName` and XML schema datatype `String` can be range of such a property. Furthermore, each ontology class can have a number of instances (termed as "Individuals") linked to it. For example, the class `PhD_Student` can have an individual `John` as its instance, representing that John is a PhD student.

The (object and data) properties can be represented as nodes [Flury et al., 2004, Burleson et al., 2007] or edges [Shaban-Nejad et al., 2011, Trinkunas et al., 2007]. If properties are represented as nodes, one may have orphaned nodes in the ontology graph due to properties without any domain and range. On other hand, if properties are represented as edges, one may find orphaned edges for the above case. Furthermore, each attachment of a (property) edge with a class will refer to addition of a domain/range axiom in the ontology. Though, it is easier to represent the object and data properties as edges at the instance level (to represent a property assertion axiom, one instance must be linked to another instance through given property), we represent properties as distinct nodes and allowed existence of orphaned node.

Based on the given description, an ontology graph G_O can be given as a set of nodes and edges $G_O = (N, E)$ where:

- $N = (C, O, D, I, X)$ is the set of ontology entities, represented as nodes in ontology graph. An ontology node $n \in N$ either represent a class C, object property O, data property D, individual I or an XML schema data type X.

- E is the set of edges that represents the ontology axioms connecting two ontology nodes where each edge $e \in E$ can be given as (n_s, e, n_t). Here, $n_s, n_t \in N$ refer to the source and target ontology nodes, respectively. As we have different types of axioms in an ontology, an ontology edge e can be of different types. It may link two classes to each other, a class to an individual, a property to a class etc. An ontology edge set can be given as $E = (E_c^c, E_c^o, E_o^c, E_o^o, E_c^d, E_d^x, E_d^d, E_i^c, E_i^i, E_o, E_d)$. The subscript and the superscript values here refer to the type of the source and target nodes of an edge, respectively.

  - The edge type E_c^c represents a class-class relationship. Such relationship can be either of $subclassOf$, $disjointClasses$ or $equivalentClasses$ type.

  - The edge type E_c^o represents a class-object property relationship, i.e. representing a $domainOf$ axiom for an object property O.

  - The edge type E_o^c represents an object property-class relationship, i.e. representing a $rangeOf$ axiom for an object property O.

  - The edge type E_o^o represents an object property-object property relationship. Such relationship can either be of $equivalentObjectProperties$, $subObjectP$-

$roperties$, $inverseObjectProperties$ or $disjointObjectProperties$ type.

- The edge type E_c^d represents a class-data property relationship, i.e. representing a $domainOf$ axiom for a data property D.

- The edge type E_d^x represents a data property-XML schema data type relationship, i.e. representing a $rangeOf$ axiom for a data property D.

- The edge type E_d^d represents a data property-data property relationship. Such relationship can either be of $equivalentDataProperties$, $subDataProperties$ or $disjointDataProperties$ type.

- The edge type E_i^c represents an individual-class relationship, i.e. representing an $classAsserion$ axiom for an individual I.

- The edge type E_i^i represents an individual-individual relationship. These edges are further divided into two categories. First, those edges (axioms) that link two ontology individuals directly. Such edges can either be of $sameIndividual$ or $differentIndividual$ type. The second type of edges are those that link two individuals through a ontology property instantiation. Such edges can either be of $objectPropertyAssertion$ or $dataPropertyAssertion$ type.

- The edge type E_o represents an attribute of an object property. Such edges can either be of $Functional$, $InverseFunctional$, $Transitive$, $Symmetric$, $Asymmetric$, $Reflexive$ or $Irreflexive$ type.

- The edge type E_d represents a $Functional$ attribute of a data property.

7.2.2 Graph-based composite change specification

Usage of graph-based transformation for the representation of atomic ontology changes has been suggested in the past [Mitra et al., 2000, Shaban-Nejad et al., 2011, De Leenheer et al., 2007]. In [De Leenheer et al., 2007], the authors adopt the idea of utilizing

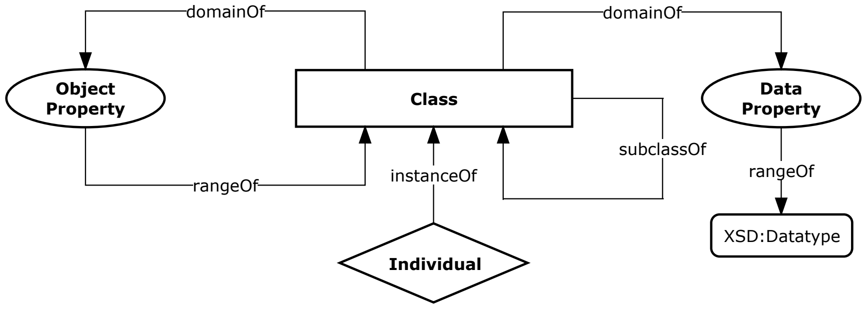


Figure 7.1: Type graph for ontology entity relationships

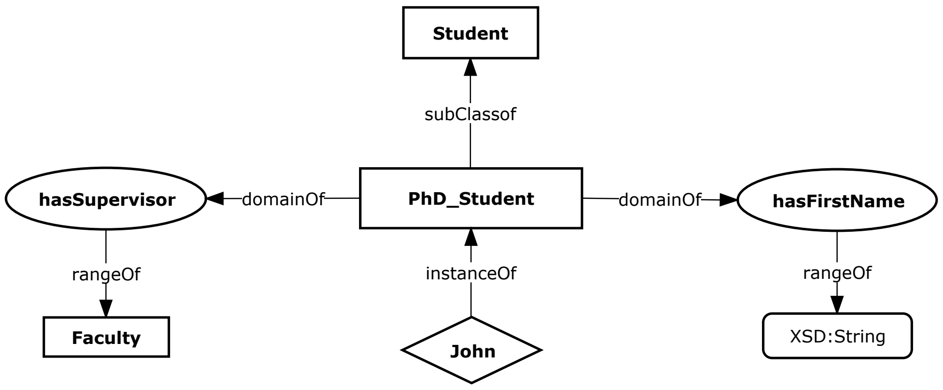


Figure 7.2: Typed ontology subgraph

a graph transformation approach for software evolution and applied it to ontology evolution. Similar to our work, they construct a metamodel for the domain ontologies in a form of type graph and domain ontologies itself are represented as typed graphs. The ontology changes are represented using graph transformation rules in AGG - a general purpose graph transformation tool. In [Shaban-Nejad et al., 2011], the authors present a graph-oriented double pushout (DPO) formalization and evolution of bio-ontologies. The authors made use of a rule-based hierarchical distributed graph transformation approach. In this case, the DPO approach has been extended from flat to hierarchical graphs [Drewes et al., 2002] where transformation rules can be applied on a hierarchical level.

For the composite ontology change specification, we follow the double pushout (DPO) approach, but adapt it to our needs. The DPO approach allows us to specify the graph transformation rules and gluing conditions (discussed below), for an applied composite ontology change, in a form of pairs of graph morphisms ($L \xleftarrow{l} K \xrightarrow{r} R$) – Figure 7.3. First, we describe the core DPO approach following Ehrig et al. [Ehrig et al., 1973].

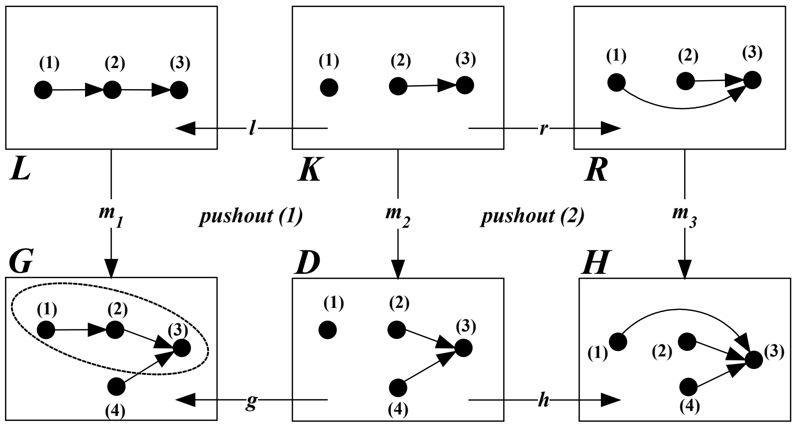


Figure 7.3: Double-pushout approach for graph transformation

Referenced and ontology subgraphs. The DPO approach is called "double pushout" as the complete transformation of an input ontology subgraph G into a target ontology subgraph H is translated into two types of changes, i.e. exclusion and inclusion change operations. The DPO approach uses a graph homomorphism approach where L, K and R represent the *referenced* subgraphs and G, D and H represent the *ontology* input subgraphs. If the match m_1 finds an occurrence of L in a given ontology subgraph G, then $G \overset{l,m_1}{\Longrightarrow} D$ denotes the derivation where l is applied to G leading to a derived graph D (Figure 7.3). Similarly, if the match m_2 finds an occurrence of K in derived ontology subgraph D, then $D \overset{r,m_2}{\Longrightarrow} H$ denotes the derivation where r is applied to D leading to the output subgraph H.

The graph L is the referenced input subgraph representing items (i.e. ontology nodes or edges) that must exist in the ontology input subgraph G for the application of the composite change. In other words, graph G represents the initial state of the ontology, i.e. the preconditions to be satisfied by the input ontology subgraph. The graph R is the referenced output subgraph representing the items that must exist in the resulting target ontology subgraph H, after the application of a composite change. In other words, graph H represents the final state of the ontology, i.e. the postconditions to be satisfied by the output ontology subgraph. The referenced graph K represents the "gluing graph" ($L \cap R$), also known as *interface graph*, representing the graph items that must be read during the transformation but are not consumed, i.e. representing the intermediate state after the application of exclusion type atomic change operations.

Note, the graph transformation here represents the transformation of an input ontology subgraph into a target ontology subgraph. Each node here represents an ontology entity; whereas the set of *graph nodes* of change log subgraph (discussed in Chapter 6), are mentioned here in the form of *productions* (discussed below).

Graph transformation rules. The graph transformation rules, also known as *productions* (p), refer to the change operations being applied to the subgraphs during the two pushouts. It defines the correspondence between the source and the target subgraph determining what is to be deleted, preserved or constructed. For example in Figure 7.3, the first production (represented as l) refers to the exclusion change operations of pushout 1 that deletes certain items (ontology nodes or edges) from the reference input subgraph L. The second production (represented as r) refers to the inclusion change operations of pushout 2 that adds certain items (ontology nodes or edges) into the reference gluing graph K. The productions representing the changes being applied to the input ontology subgraph G are known as *co-productions* and are given as g and h in Figure 7.3.

Match (m). In order to apply production l to the ontology graph, first we need to identify the occurrence of subgraph L in the ontology graph, called a "match". For example, $m_1 \colon L \longrightarrow G$ for a production l is a graph homomorphism, i.e. each ontology node/edge of subgraph L is mapped to a distinct ontology node/edge in subgraph G in such a way that graphical structure and labels are preserved [Corradini et al., 1996]. The context gluing graph D is obtained by deleting all items (ontology nodes and edges) from the subgraph G which have a match (image) in the subgraph L but not in subgraph K - pushout 1. Intuitively, we can say that if a match m_1 finds an occurrence of subgraph L in a given ontology subgraph G, then $G \overset{l,m_1}{\Longrightarrow} D$ represent the derivation (co-production) g where l is applied to G leading to a derived graph D. Informally, the subgraph D is achieved by replacing the occurrence of L in G by K. Similarly in pushout 2, the subgraph H is obtained by inserting distinct items (ontology nodes and edges) of subgraph R that do not have any match (image) in subgraph K ($h = D \overset{r,m_2}{\Longrightarrow} H$).

Gluing conditions. The possible conflicts in the graph matching step are resolved by applying certain matching constraints, known as "gluing conditions". A gluing condition consists of two parts, i.e. a dangling condition and an identification condition. The *dangling condition* (C_d) ensures that the graph D, obtained by applying the production l, contains no "dangling" edge, i.e. an edge without a source or a target node. For example, if an ontology node v is deleted from graph G, all the edges that contain ontology node v as a source or target node, will also be deleted. The *identification condition* (C_i) ensures that every item of graph G that has to be deleted by the application of production l, must have only one distinct match in the graph L, i.e. a 1:1 matching. Thus, we can say that the items from the left-hand side graph L may only be identified in resultant graph R if they also belong to the gluing graph (i.e. preserved items) [Heckel et al., 2002].

7.3 DPO adaptation - re-attachment of dangling edges

7.3.1 Motivation

A modification of the DPO approach is necessary to deal with the preservation of properties under change. A composite change is a combination of inclusion and exclusion type atomic change operations. In the DPO approach, exclusion and inclusion type change operations are applied in pushout 1 and pushout 2, respectively. The nodes and edges to be deleted (from an ontology graph) in pushout 1 can be given as $L \backslash K$ (i.e. the elements that are present in the subgraph L, but not in K) and the set of change operations of pushout 1 (l) can be given as $l = C_r^- + C_d$. Here, C_r^- refers to the exclusion type change operations of a user's change request and C_d refers to the change operations added in pushout 1 to satisfy dangling conditions. We take pushout 1 of the DPO approach as a "structural pushout", as the pushout (including gluing conditions) refers to the completeness and correctness of the structure of a graph. The dangling condition for edges in pushout 1 ensures that the interface graph D is a proper graph by deleting the dangling edges. However, the semantics behind the applied composite change may be lost in the case where newly added entities (of pushout 2) adopt properties from the deleted/edited entities (of pushout 1), e.g. in the case of the `Split class` change.

Let x be an ontology class that is split into two sibling classes x_1 and x_2 (Figure 7.4). In pushout 1 of the split class change, class x is removed from the class hierarchy and deleted. In order to satisfy the dangling condition, the *roles* of the class x are also being deleted. Here, the term "role" refers to the properties of an ontology entity (represented in the form of edges) that relates it to other entities of the domain ontology. In Figure 7.4, the class x (in the ontology input subgraph G) has three roles i.e. b_1 (*domainOf*), b_2 (*rangeOf*) and b_3 (*instanceOf*) that are deleted to satisfy the dangling condition.

In pushout 2, two new classes x_1 and x_2 are added, replacing the class x in the class hierarchy. As classes x_1 and x_2 adopt relationships from the split class x, the deleted edges (for satisfying dangling condition) are actually not the consumed entities in this graph transformation. Thus, the deleted edges must be added back to the newly added sibling classes x_1 and x_2.

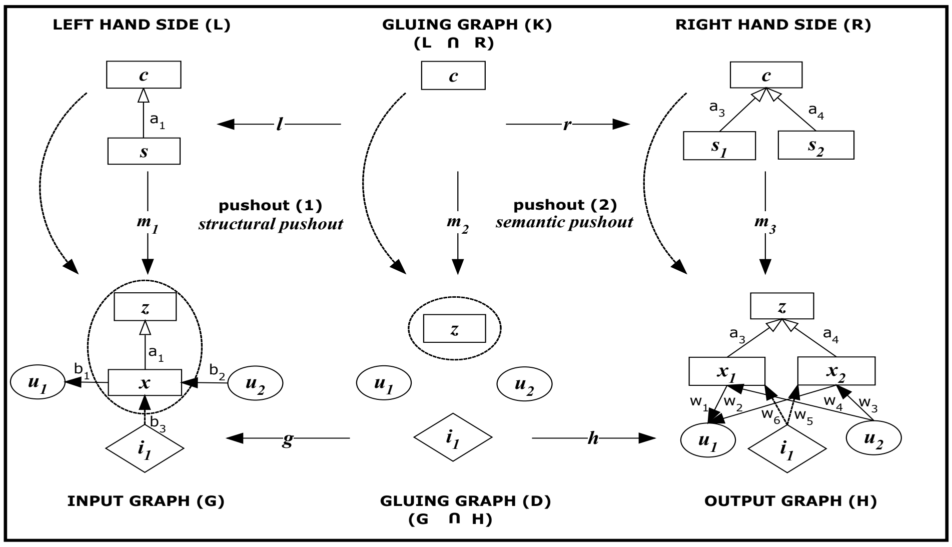


Figure 7.4: Split class (x, (x_1, x_2)) - double push out (DPO) approach

7.3.2 Extended DPO - definition

We extended the DPO approach by adding the re-attachment of the dangling edges (C'_d) that formulates the pushout 2 as a "semantic pushout" allowing a user to preserve the non-consumed entities that were deleted due to the dangling edge effect in pushout 1. Thus,

- the nodes and edges to be added (in ontology graph) in pushout 2 can be given as a set difference of R and K, denoted as $R \backslash K$ (i.e. the elements that are present in the subgraph R, but not in K), and

- the set of change operations of pushout 2 (r) can be given as $r = C_r^+ + C_d^{'}$. Here, C_r^+ refers to the inclusion type change operations included in a user's change request and $C_d^{'}$ refers to the change operations added in pushout 2 to re-attach the dangling edges that were deleted in pushout 1.

Evolution Strategies: The question that arises here is *how to make sure that intent of change is correctly achieved*, e.g. in cases where newly added classes adopt properties from deleted ones, a class becomes a subclass of its previous sibling disjoint class, properties are moved in the class hierarchy and inferred instances are not valid anymore etc. To resolve this issue, different sets of atomic change operations, in the form of an evolution strategy [Javed et al., 2012a], can be utilized to achieve a consistent state of the domain ontology. However, each solution may lead to a distinct consistent ontology version. Here, the term *consistent state* not only refers to a structural consistency, but also a semantic consistency [Qin et al., 2009]. For example, in the split change case scenario (given in Figure 7.4), where classes x_1 and x_2 adopt properties from deleted class x, a user can either

- distribute the deleted roles of class x among the newly added replacement classes, OR
- re-attach the roles to one of the newly added replacement class, OR
- re-attach roles to both the newly added replacement classes, OR
- do nothing.

As in our running example, we chose option 3, the nodes u_1, u_2 and i_1 are attached to the nodes x_1 and x_2 resulting into the output graph H (i.e. $h = D \overset{r,m_2}{\Longrightarrow} H$).

As different users may have different perspectives of a domain ontology and different objectives of an ontology change, given evolution strategies are customizable and extendable. A list of composite level evolution strategies is given in Appendix G.

7.3.3 Applying DPO to composite change patterns

We applied the DPO approach to composite change patterns. There exist no agreed standard set of composite change patterns. One can combine different atomic level change operations in order to construct new patterns. Thus, providing an exhaustive list of composite change patterns is not feasible. In our current work, we select the composite change patterns and their definitions from [Stojanovic, 2004] which are given in Table 7.1.

As we discussed earlier, a composite change pattern can be applied using different order of inclusion and exclusion type change operation. However, in order to adapt to DPO, we presume that exclusion type change operations have been applied prior to inclusion type change operations. This assumption allows us to define an association between pushout 1 and exclusion type change operations and between pushout 2 and inclusion type change operations. For example, for the `Pull up class` (x, x_1) change pattern, two pushouts of DPO can be given as:

Op.1: `Delete subclassOfAxiom` (x, x_1) – [*pushout 1*]

Op.2: `Add suclassOfAxiom` (x, y) – [*pushout 2*]

Op.3: `Add suclassOfAxiom` (x, z) – [*pushout 2*]

*assuming class x_1 has two super classes, i.e. y and z.

Similarly, for the `Pull down property` (p, x_1, x_2) change pattern, two pushouts of DPO can be given as:

Op.1: `Delete domainOfAxiom`(p, x_1) – [*pushout 1*]

Op.2: `Add domainOfAxiom` (p, x_2) – [*pushout 2*]

7.3.4 "Split class" change scenario

In this section, we provide the details of our extended DPO approach using the "split class" composite change as a case scenario. The composite change `split class` refers to splitting a class into two (or more) sibling classes (Table 7.1). For example in Figure 7.4, the class x ($x \in G$) has been split into two sibling classes x_1 and x_2 ($x_1, x_2 \in H$). The nodes and edges, given in Figure 7.4, represent the following ontology elements: square node $\longrightarrow$ class(c), oval node $\longrightarrow$ property (t), diamond node $\longrightarrow$ individual (i), edge [src(e) = c & tar(e) = c] $\longrightarrow$ is-a relationship, edge [src(e) = t & tar(e) = c] $\longrightarrow$ range of a property, edge [src(e) = c & tar(e) = t] $\longrightarrow$ domain of a property and edge [src(e) = i & tar(e) = c] $\longrightarrow$ instanceOf relationship.

Table 7.1: List of composite change patterns and their definitions

Composite Change	**Description**
Split class (x, (x_1, x_2))	Split a class x into two newly created sibling classes x_1 and x_2.
Merge classes ((x_1, x_2), x)	Merge two existing classes x_1 and x_2 into one newly created class x and cumulate all roles of x_1 and x_2 into x.
Pull up class (x, x_1)	Pull class x up in its class hierarchy and attach it to all parents of its previous parent x_1.
Pull up class (x)	Pull class x up in its class hierarchy and attach it to all parents of all its previous parents.
Pull down class (x, x_1)	Pull class x down in its class hierarchy and attach it as a child to its previous sibling class x_1.
Pull down class (x)	Pull class x down in its class hierarchy and attach it as a child to all its previous sibling classes.
Move class (x, x_1)	Detach class x from its previous superclass and attach it as a subclass to a class x_1 (which previously was not a direct/indirect superclass of class x).
Group classes (x, (x_1, x_2))	Create a common parent class x for sibling classes x_1 and x_2 and transfer the common properties to it.
Add Generalisation class (x, x_1)	Add a new class x between x_1 and all its super classes.
Add Specialization class (x, x_1)	Add a new class x between x_1 and all its subclasses.
Pull up property (p, x_1, x_2)	Pull a property p up in the class hierarchy and attach it to the superclass x_2 of its previous domain/range class x_1.
Pull down property (p, x_1, x_2)	Pull a property p down in the class hierarchy and attach it to the subclass x_2 of its previous domain/range class x_1.

Table 7.2 gives the formal definition of the `split class` composite change example, given in Figure 7.4, in terms of ontology and DPO graph changes and conditions. Now, we discuss each pushout and the involved change operations.

pushout 1 : First, we identify the occurrence of the reference subgraph L in the ontology graph (i.e. $m_1 \colon L \longrightarrow G$). Once the match is found, production l is being applied to the matched ontology subgraph G (through co-production g) resulting in a gluing graph D (i.e. $g = G \overset{l,m_1}{\Longrightarrow} D$). The co-production g represents the deletion of class x from the class hierarchy. Thus, in Figure 7.4, node x and edge a_1 are deleted from the input ontology subgraph G. Furthermore, to satisfy the dangling conditions, edges b_1, b_2 and b_3 are also deleted.

Table 7.2: Formal definition of composite change `Split class`(x, (x_1, x_2))

Split class (x, (x_1, x_2))	
Intuition: Splitting a class x into two sibling classes x_1 and x_2.	
Exclusion Changes (δ_1)	**Pushout–1** (Type)
x rdf:type OWL:Class	delete node x (m_2)
x rdfs:subClassOf z	delete edge a_1 (m_2)
u_1 rdfs:domain z	delete edge b_1 (C_d)
u_2 rdfs:range z	delete edge b_2 (C_d)
i_1 rdf:type z	delete edge b_3 (C_d)
Inclusion Changes (δ_2)	**Pushout–2** (Type)
x_1 rdf:type OWL:Class	add node x_1 (m_3)
x_1 rdfs:subClassOf z	add edge a_3 (m_3)
x_2 rdf:type OWL:Class	add node x_2 (m_3)
x_2 rdfs:subClassOf z	add edge a_4 (m_3)
u_1 rdfs:domain x_1, x_2	add edges w_1, w_2 (C'_d)
u_2 rdfs:range x_1, x_2	add edges w_3, w_4 (C'_d)
i_1 rdf:type x_1, x_2	add edges w_5, w_6 (C'_d)
Ontology Conditions (ϕ)	**Identification Conditions (C_i)**
$x_1, x_2 \notin O$ — $x_1, x_2 \in O'$	$x_1, x_2 \notin G$ — $x_1, x_2 \in H$
$x \in O$ — $x \notin O'$	$x \in G$ — $x \notin H$
$z \in (O, O')$	$z \in D$
(x rdfs:subClassOf z) $\in O$	src(a_1) = x & tar(a_1) = z in G
(x_1 rdfs:subClassOf z) $\in O'$	src(a_3) = x_1 & tar(a_3) = z in H
(x_2 rdfs:subClassOf z) $\in O'$	src(a_4) = x_2 & tar(a_4) = z in H

pushout 2 : Similar to pushout 1, first we identify the match of the reference gluing graph K in the ontology gluing subgraph D (i.e. $m_2 \colon K \longrightarrow D$). Once a match is confirmed, production r is applied to the ontology subgraph D (through co-production h). The co-production h represents the addition of two classes x_1 and x_2 in the ontology class hierarchy. Thus, in Figure 7.4, the nodes x_1 and x_2 are added to the gluing graph D and are linked to node z through edges a_3 and a_4.

In order to ensure that the non-consumed roles (edges) of the deleted class x have been transferred to the newly added classes, the deleted dangling edges of pushout 1 must be added back in pushout 2. To do this, a user can select different evolution strategies [Javed et al., 2012a] that guide in adopting the roles of the deleted class by the newly added classes.

7.4 Detection of composite changes

Little work has been done in the area of the detection of composite changes [Plessers et al., 2005, Papavassiliou et al., 2009]. Based on the work mentioned in [Plessers et al., 2005], ontology changes are recorded in the form of a version log[2] and each change is detected based on its comparison to a specified change definition. In [Papavassiliou et al., 2009], ontology changes are captured using different ontology versions. The authors focused on identifying composite changes by detecting the differences between (two) versions of the same ontology. In contrast to their work, we record ontology changes in the form of a change log and we operationalize the composite change detection in terms of graph matching algorithms.

The DPO approach can be applied directly, if one preserves the different versions of the ontology (such as in [Papavassiliou et al., 2009]). As we log the applied change operations, rather than the different versions of the ontology, we provide productions as an input to the composite change detection algorithm, rather than the ontology and referenced ontology subgraphs. Thus, the input to the composite change detection algorithm is the change log graph (representing the applied atomic changes on the domain ontology) and the referenced composite change graph (representing the sequence of atomic changes to be identified) along with the specified conditions (ϕ).

In terms of graph-based pattern matching, there exist a number of basic and frequently used algorithms [Wen et al., 2010, Van der Aalst et al., 2006, Baggenstos et al., 2006, Rudolf et al., 2000, Valiente et al., 1997]. However, as the composite changes are mainly detected from a sequential atomic change log, we adopt the ideas from *string pattern matching* and utilized the algorithms to our needs. The most prominent algo-

[2]a version log keeps record of different versions of an ontology entity during its lifespan

rithms in the area of string pattern matching include *Brute-force* exact pattern match, *Boyer-Moore* algorithm, *Karp-Rabin* and *Knuth-Morris-Pratt's* algorithm. Similar to a string, where a string is a sequence of characters, in our case, we have a sequence of atomic change operations stored in a sequential atomic change log. By symbolizing an atomic change operation as a single character, an existing string pattern matching algorithm can be directly applied.

7.4.1 Algorithm for composite change detection

The ontology change log graph is a collection of sessions (S), where each session ($s \in S$) consists of the change log entries, from the time the domain ontology is loaded into the ontology editor until the time it is closed. As we mentioned in the previous chapter, we opt for a restrictive approach where a composite change pattern is applied completely within a session. In this regard, we divided the change log graph into a sequence of sessions from where a composite change pattern can be identified.

The presented algorithm for the composite change pattern detection is similar to the *Brute-force* exact pattern matching algorithm, where the overall approach is based on Depth First Search (DFS) strategy. We try to match the first node of the referenced graph with the first node of a change log session. If the node is matched, we try to match second node, and so on. If we hit a failure, we slide the pattern over one graph nodes and repeat the process. In addition to the general matching of the graph nodes, our algorithm ensures that the conditions (i.e. existential and correlations) defined for any composite change pattern are satisfied. This is done by comparing the parameters of the change operations and their roles in the ontology class hierarchy. Furthermore, the algorithm is extendable to cover the unordered composite change patterns that are semantically identical to the reference composite change operation.

The basic idea of the presented composite change detection algorithm is to iterate over each session of the change log graph and find the location where an applied composite change may start. We pass the identified location and the reference graph G_r to a function that extracts the sequence of change nodes that completely map to G_r. In the mapping step, it ensures that the correlations among the parameters of the identified change operations are satisfied.

Description of algorithm : The composite change algorithm is given in listings 1.1 and 1.2, where listing 1.1 describes the main algorithm and listing 1.2 presents the algorithm for one of the functions (method). Below, we describe the algorithm in steps (and sub-steps):

Listing 1.1:

Step A: The algorithm takes the change log graph G and reference graph G_r as an input and groups the graph nodes into a set of sessions (line 1–2).

Step B: Once we have the session set S, the algorithm iterates over each session s (line 3–18).

Step B.1: Within each iteration over session s, first we get the range of the session by extracting the node ids of the first and the last node of the session. The parameter *currentId* (representing the id of the currently visited graph node) is initialized with the first node id (line 4–6).

Step B.2: We iterate over the graph nodes of the session, until the id of the currently visited node is less than the id of the last node of the session (line 7–17).

Step B.2.1: In each iteration, we extract the first node n_r from the reference graph $G - r$ and identify a matching node to n_r from the log session s (line 8–9).

Step B.2.2: If no matching node is identified from the session, the algorithm goes back to step 3 to selects the next session from the session set (line 10–11).

Step B.2.3: If a matching node is identified from the session, the algorithm passes the matched node n_g, reference composite change graph G_r and the session s to the method `matchPattern()`, that identifies the complete composite change sequence (line 13).

Step B.2.4: The method *matchPattern()* returns a list of change operations (representing a detected composite change operation) that is passed as an output of the algorithm or returns a *null* value (representing that a composite change was not identified at particular location of the session) (line 13–16).

Algorithm 7.4.1 Composite Change Detection Algorithm

```
Input: Change Log Graph ($G$) and Reference Graph ($G_r$)
Output: Set of Identified Composite Changes ($SC$)
 1: $set \leftarrow getGraphNodeSet(G)$
 2: $S \leftarrow getSessionSet(set)$
 3: for each session $s$ in session set $S$ do
 4:   $firstNodeId \leftarrow getFirstNodeId(s)$
 5:   $lastNodeId \leftarrow getLastNodeId(s)$
 6:   $currentId = firstNodeId$
 7:   while $currentId < lastNodeId$ do
 8:     $n_r \leftarrow getFirstNode(G_r)$
 9:     $n_g \leftarrow findMatchingNode(n_r, s)$
10:     if $n_g == null$ then
11:       go back to step 3.
12:     end if
13:     $list = matchPattern(n_g, G_r, s)$
14:     if $list \neq null$ then
```

```
15:         $SC \leftarrow list$
16:     **end if**
17:   **end while**
18: **end for**
```

Listing 1.2:

Step A: First, we save the passed graph node n_g in an extendable *list* (line 1).

Step B: We iterate over the session s, as long as the complete composite change reference graph is not identified (line 2–12).

Step B2.1: In each iteration, we select the subsequent nodes of the reference graph G_r and the session s (line 3–4).

Step B2.2: We match the selected nodes. If the nodes are matched and the correlations are satisfied, the selected node n_g is added into the *list* and the next subsequent node of the session s is selected as a current node (line 5–7).

Step B2.3: If the nodes do not match (in above step B2.2), the next subsequent node of the session s is selected as a current node (line 5–7) and the algorithm goes back to Listing 1.1 (from where this method was called) with a *null* value returned.

Algorithm 7.4.2 Method: matchPattern()

Input: Matched Graph Nodes n_g, n_r and session s

Output: List of Identified Composite Changes

```
1: $list \leftarrow n_g$
2: **while** $list$ is not complete **do**
3:   $n_r \leftarrow getNextNode(G_r)$
4:   $n_g \leftarrow getNextNode(s)$
```

```
 5:     if $matched(n_g, n_r)$ and correlation is satisfied then
 6:         $list.add(n_g)$
 7:         $currentNode = currentNode + 1$
 8:     else
 9:         $currentNode = currentNode + 1$
10:         return $null$
11:     end if
12: end while
13: return $list$
```

7.5 Limitations and illustration of results

7.5.1 Limitations

There are two main limitations of the presented algorithm.

1. The algorithm does not cover the complex classes created using logical class constructors (i.e. intersection, union and complement). The algorithm considers all defined classes as *simple classes*.

2. As we opt for the pattern *matching* approach here, unordered changes are not covered by the composite change detection algorithm. The algorithm only matches the change sequences that completely overlap (in terms of order and number of ontology change operations) with the referenced change sequence of the composite change operation.

7.5.2 Illustration of the results

Two examples from the identified composite changes are given in Figures 7.5 and 7.6.

The example given in Figure 7.5 represents an identified `Split class` change, where *"Distribute the roles"* was the selected evolution strategy. In the previous version of the ontology V_1, class `Student` was classified into `MSStudent`, `PhD_Student` and `UGStudent`.

Thus, all the master's students (OWL:Individual), whether taught or research-based, were direct instances of class `MSStudent`. In a subsequent version of ontology V_2, in order to distinguish between research-based and course-based students of a master's degree, the class `MSStudent` is split into two sibling classes (i.e. `MSByResearchStudent` and `MSTaughtStudent`). Based on the selected evolution strategy, the direct instances of the deleted class `MSStudent` are distributed among the newly added classes.

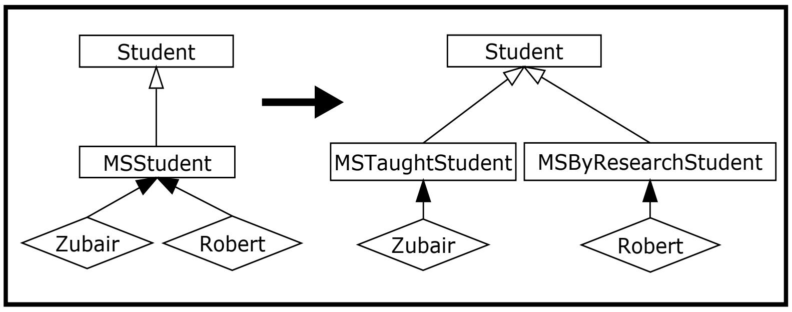


Figure 7.5: Identified composite change - `‘‘Split class’’`

The example given in Figure 7.6 represents an identified `Pull up property` change on class `PhD_Student`, where `MSByResearchStudent` and `PhD_Student` were direct subclasses of `Student`. In the previous version of the ontology V_1, `MSByResearchStudent` and `PhD_Student` were grouped under the class `ResearchStudent`. In this regard, the next step is to pull up the common properties of `PhD_Student` and `MSbyResearchStudent` to the common superclass `ResearchStudent` in the subsequent version V_2. Thus, common properties (such as, `ResearchTrack` (object property), `Affiliation` (object property), `isSupervisorOf` (object property), `ResearchTitle` (data property) etc.) are pulled up.

7.6 Summary

Activity log mining is not restricted to creating new formal process models, but can be extended to discover implicit semantic knowledge from the change log. For example,

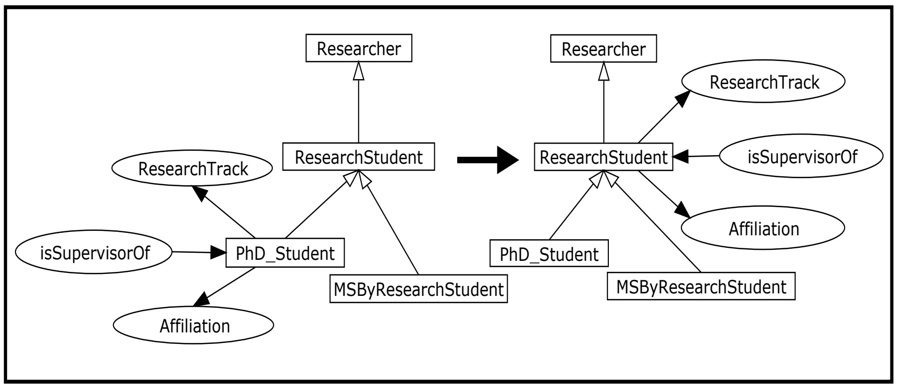


Figure 7.6: Identified composite change- `‘‘Pull up property’’`

such knowledge may give an ontology engineer clues about semantics/reasons behind any of the applied change, based on the actual current data of change activities.

In this chapter, we presented our research towards identification of the composite change patterns from an ontology change log in the form of pattern matching algorithms. We formalised the ontology change log data using a graph-based approach, where each graph node represents an atomic change operations. We adapted the double pushout (DPO) graph transformation, where an input ontology subgraph is transformed into a target ontology subgraph based on the applied change operations and conditions to be satisfied. We presented the change detection algorithms that capture the composite change patterns from the change log graph.

We analyzed the detected composite changes of different types. It has been realized that learning about semantics behind any of the applied change helps us in keeping the ontology consistent in a more appropriate manner. To do so, higher level evolutionary strategies are essential [Javed et al., 2012a]. Furthermore, a composite change can be applied in different ways, that leads to application of different change operations (from atomic or composite level change operations).

Chapter 8

Change patterns discovery algorithms

In this chapter, we discuss our approach towards discovery of domain-specific change patterns (level 3 change operators) from an Atomic Change Log (ACL). For the storage of ontology changes, we employ an RDF triple-based storage system in order to maintain a complete, fine granular ontology change representation and to identify the re-usable domain-specific change patterns which cannot be identified by simply navigating or querying the ontology changes. We formalize the change log using a graph-based approach and analyze the ontology change log graph in order to identify the frequent change sequences that occur during evolution as a combination of single atomic change operations. Such sequences are then applied as a reference in order to discover reusable usage-driven domain-specific change patterns. We describe the pattern discovery algorithms and measure their performance using experimental results.

8.1 Empirical analysis of atomic change log graph

We studied the atomic change log empirically. The atomic change log graph allows us to identify and classify frequent changes that occur in domain ontologies over a period of time. Initially, we analyzed the change log graph manually and observed the groups of atomic change operations that occur repeatedly during the evolution of domain ontologies. We identified these as frequent recurring change patterns that can be reused. While patterns are sometimes used in their exact form, often more flexibility is needed. Users often use different orderings of change operations to perform the same (semantically identical) change at different times. To capture semantically identical, but operationally different change patterns, we introduce a metric, called *node-distance* (c.f. Section 6.3.2). This help us to a more flexible notion of a pattern.

Definition 8.1: Node-Distance refers to the distance between two adjacent nodes of a sequence in a change log graph. In order to identify the recurrent change patterns, the value of a node distance is a user input and is denoted by an uppercase letter X.

8.1.1 Types of ontology change patterns

We organized the different types of patterns into two basic subdivisions, i.e. *Ordered Change Patterns (OP)* and *Unordered Change Patterns (UP).*

- *Ordered Change Patterns (OP)* comprise ordered change operations from the change log graph. Such (complete or partial) change sequences (c.f. Section 6.4.1) may have positive node distance value, starting from zero to a user given value (X).

  Type 1: Ordered Complete Change Patterns (OCP).

  Type 2: Ordered Partial Change Patterns (OPP).

- *Unordered Change Patterns* comprise unordered change operation from change log graph. These (complete or partial) change sequences may have node distance whose range is from (user-defined) negative node distance value ($-X$) to positive node distance value ($+X$).

Type 3: Unordered Complete Change Patterns (UCP).

Type 4: Unordered Partial Change Patterns (UPP).

8.2 Metrics for ontology change pattern discovery

We consider identifying recurring change operations from a change log as a problem of recognition of a frequent pattern in a graph. Identifying recurring sets of applied changes can provide an opportunity to define reusable domain-specific change patterns that can be implemented encapsulating existing knowledge-based systems [Javed et al., 2011a]. The motivation behind it is the reusability of recurrent domain-specific changes (patterns), in line with the basic idea of software reuse and to support pattern-based ontology evolution [Javed et al., 2009]. First, we describe some metrics by introducing the following definitions.

Pattern Support: The *support* of a change pattern p is the number of occurrences of such a pattern in the change log graph G [Agrawal et al., 1995, Zhao et al., 2003]. Pattern support is denoted by *supp(p)*. The minimum number of occurrences required for a sequence s in change log graph G to qualify as a change pattern p is the *minimum pattern support*, denoted by *min_supp(p)*.

Pattern Length: The *length* of a change pattern p is the number of atomic change operations in it, denoted by *len(p)* [Agrawal et al., 1994, Hirate et al., 2006]. The minimum length required for a sequence s in a change log graph G to qualify as a member of a candidate pattern set is the *minimum pattern length*, denoted by *min_len(p)*.

Candidate Change Pattern Sequence: For a given $ACL = < ac_1, ac_2, ac_3 \cdots ac_n >$, a candidate change pattern sequence cs is a sequence $< ac_{p1}, ac_{p2}, ac_{p3} \cdots ac_{pk} >$

with

- $ac_{pi} \in ACL$ for $i = 1, 2 \cdots k$ and

- if pos(ac_{pi}) < pos(ac_{pj}) in cs, then

  pos(ac_{pi}) < pos(ac_{pj}) in ACL ...

  for all $i = 1 \cdots k-1$ and $j = 2 \cdots k$.

Change Pattern Sequence: A candidate change pattern sequence cs is a discovered change pattern p if

- $len(cs) \geq min\_len(p)$.

  i.e. the length of the candidate change pattern sequence cs is equal to or greater than the threshold value set by the minimum pattern length.

- $supp(cs) \geq min\_supp(p)$.

  i.e. the support for the candidate change pattern sequence cs in a change log graph G is above the threshold value of the minimum pattern support.

Ordered Change Pattern: Let $p = \{s_1, s_2 \cdots s_d\}$ be a set consisting of a candidate change pattern sequence cs ($cs = s_1$) and the change pattern sequences $\{s_2, s_3 \cdots s_d\}$ that support cs in ACL. The candidate change pattern sequence cs is a discovered ordered change pattern (OP) with

- $s_i =< ac_{i1}, ac_{i2} \cdots ac_{in} > \in p$ for $i = 1 \cdots d$

- if pos(ac_{ix}) < pos(ac_{iy}) in s_i then

  pos(ac_{ix}) < pos(ac_{iy}) in ACL ...

  for all $x = 1 \cdots n-1$, $y = 2 \cdots n$.

Unordered Change Pattern: Unordered change patterns are those patterns where in comparison to a candidate change pattern sequence cs, the existing type-equivalent graph nodes in a discovered change sequence s are not in same sequential order (c.f. Section 6.4.1).

Let $p = \{s_1, s_2 \cdots s_d\}$ be a set consisting of a candidate change pattern sequence cs

Table 8.1: Input parameters for pattern discovery algorithms.

Input Parameters	**Type**
Graph representing Change log triples - G	Graph
Minimum Pattern support - *min_supp*	Integer
Minimum Pattern Length - *min_len*	Integer
Maximum n-distance - X	Integer

($cs = s_1$) and the change pattern sequences $\{s_2, s_3 \cdots s_d\}$ that support cs in ACL and u and v be the first and the last positions in a discovered change pattern sequence s_i (for $i = 2 \cdots d$), respectively. The candidate change pattern sequence cs is a discovered unordered change pattern (UP) with

- $s_i =< ac_1, ac_2 \cdots ac_n > \in p$ for $i = 2 \cdots d$

- if $u =$ pos(ac_1) in s_i and

$v =$ pos(ac_n) in s_i, then

$u \leq$ pos(ac_x) $\leq v$ in ACL, for all $x = 1 \cdots n$

8.3 Complete change pattern discovery algorithms

This section describes the algorithms used for the discovery of complete change patterns (CP). The section is divided into two parts, i.e. an algorithm for searching ordered complete change patterns (OCP) and an algorithm for searching unordered complete change patterns (UCP). The inputs to the pattern discovery algorithms are given in Table 8.1.

Before we give a description of each algorithm in detail, it is necessary to introduce some frequently used terms.

Definition 8.2- Target Entity, Primary and Auxiliary Context: The word *target entity* refers to the ontology entity to which the change has actually been applied, whereas the *primary and auxiliary context* refer to the ontology entities which will be affected by such change. For example, in change operation `Add rangeOfAxiom(hasSupervisor, Faculty)`, object property `hasSupervisor` is the target entity (to which change has been applied) and concept `Faculty` is the primary context. Similarly, in change operation `Add dataPropertyAssertionAxiom(Javed, studentId, '58106383')`, Individual `Javed` is the target entity, data property `studentId` is the primary context and the literal value `'58106383'` is the auxiliary context.

Definition 8.3- Candidate Node (cn): A candidate node *cn* is a node from the graph which will be selected at the start of the graph node's iteration process. Each node of the graph will act as a candidate node *cn* in one iteration each of the algorithm.

Definition 8.4- Candidate Sequence (cs): The candidate sequence *cs* is the context-aware set of graph nodes starting from particular candidate node *cn*.

Definition 8.5- Discovered Node (dn): The discovered node *dn* is a node which matches the candidate node *cn* (in a particular iteration) in terms of its operation, element and type of context. Capital letter *DN* refers to the set of discovered nodes.

Definition 8.6- Discovered Sequence (ds): The discovered sequence *ds* is the context-aware set of graph nodes starting from particular discovered node *dn* and matches candidate sequence *cs* (in a particular iteration). Capital letter *DS* refers to the set of discovered node sequences.

8.3.1 OCP discovery algorithm

The OCP discovery algorithm is similar to the *Brute-force* exact pattern matching algorithms due to the assumption that the *discovered sequence* is an ordered change sequence, where the (searched) graph nodes are one after the other (in comparison to the candidate sequence). However, the difference lies in i) identifying the locations from where a matching discovered sequence may start and ii) the *search space* of a graph node. This is described below.

After generating a candidate sequence, OCP algorithm first identifies the locations in the change log graph from where a discovered sequence may start (Breadth First Search (BFS)). Once these locations are identified, we iterate over each of them. Here, we opt for the Depth First Search (DFS) approach. In each iteration, we try to match the first node of the candidate sequence with the first node present at particular location. If the node is matched, we try to match the second node of the candidate sequence, and so on. If we hit a failure, rather sliding the candidate sequence over the next graph node and restart the whole pattern matching process (as in case of Brute-force), we search for the unmatched graph node further in the search space specified by the permissible node-distance value (c.f. Section 6.3.2). This approach has been adapted from the state of the art sequential pattern mining algorithms with gap-constrained [Zhu et al., 2007, Zhang et al., 2007, Li et al., 2008]. Thus, if the candidate node is matched to any graph node in the search space, it is added in the discovered sequence and we try to match the next node of the candidate sequence. On the other hand, if the candidate node is yet not matched to a graph node in the search space, we stop the search process and move to the next iteration.

To discover ordered complete change patterns (OCP), the identified sequences are of same length and contain change operations in the exact same sequential order. The pseudo-code of the *OCP* algorithm is described in algorithm lists (8.3.3 – 8.3.5) and explained below in detail in a form of steps and sub-steps.

8.3.1.1 Description of algorithm

Step A: The algorithm iterates over each node of the graph and selects it as a candidate node (cn_k), where k refers to the identification key of the node in graph G.

Step B: Once the candidate node and its target entity are captured, an iteration process of an extension of a candidate node cn_k to its adjacent nodes cn_{k++} starts and it continues until no more extension is possible (i.e. an adjacent node does not share the same target entity).

Step B.1: If the target entity of the adjacent node is matched with the target entity of candidate node, it will be taken as the next node of the candidate sequence *cs*. If the target entity does not match, an iterative process will start to find the next node whose target entity matches with the target entity of the candidate node. The iterations will continue based on the user-given value X, i.e. the allowed gap between two adjacent nodes of a pattern (*n-distance*).

Algorithm 8.3.3 Ordered Complete Change Pattern Discovery Algorithm

Input: Graph (G), Minimum Pattern Support (min_supp), Minimum Pattern Length (min_len), Maximum *n-distance* (X)

Output: Set of Domain-Specific Change Patterns (S)

```
1: for $i = 0$ to $N_G.size$ do
2:     $k = 0$
3:     $cs \leftarrow GenerateCandidateSequence(cn_k)$
4:     if $(cs.size < min\_len)$ then
5:         go back to step 1.
6:     end if
7:     $DN \leftarrow DiscoverMatchingNodes(cn_k)$
8:     $DS \leftarrow DN$
9:     if $(DS.size < min\_supp)$ then
10:        go back to step 1.
11:    end if
12:    while $(DS.size \geq min\_supp)$ do
13:        for each discovered sequence $ds$ in $DS$ do
14:            $t \leftarrow getTargetEntity(ds)$
15:            $Expand(dn_j, X)$
```

```
16:         $Match(dn_{j++}, cn_{k++}, t)$
17:         if (Expanded && Matched) then
18:            $ds \leftarrow dn_{j++}$
19:         else
20:            break while loop.
21:         end if
22:      end for
23:      if ($ds.size < min\_len$) then
24:         discard $ds$ from $DS$
25:      end if
26:   end while
27:   $max \leftarrow$ get Maximum Size of Sequences such that ($max \geq min\_supp$)
28:   for each sequence $ds$ in $DS$ do
29:      if ($ds.size < max$) then
30:         discard $ds$
31:      else
32:         trimSequence($ds$, $max$)
33:      end if
34:   end for
35:   $P_{domain\_specific} \leftarrow (ds + cs)$
36:   $S \leftarrow P_{domain\_specific}$
37: end for
```

Step C: Once the candidate sequence is constructed and is above the threshold value of minimum pattern length (min_len), next step is to search for the matching nodes (i.e. discovered nodes dn) of the same type as candidate node cn_k.

Step D: If the number of discovered nodes d_n is above the threshold value set for the minimum pattern support (*min_supp*), next step is to expand the discovered nodes and match them to parallel candidate nodes. Each discovered node is expanded one after another. Similar to the expansion of candidate nodes, the identification of next node of discovered sequence *ds* is an iterative process (depending on the input value of X).

- *Step D.1*: The expansion of a discovered node *dn* stops if either no further extension of that particular discovered node is possible or expansion has reached the size of candidate sequence (i.e. length of *ds* becomes equal to length of *cs*).

Algorithm 8.3.4 Method:GenerateCandidateSequence()

Input: Graph (G), Maximum *n-distance* (X), Graph Node (n)

Output: Candidate Sequence (cs)

```
1: $k = 0$
2: $cn_k \leftarrow n$
3: $cs \leftarrow cn_k$
4: context = $true$
5: while (context) do
6:   $Expand(cn_k, X)$
7:   if (Exanded) then
8:     $cs \leftarrow cn_{k++}$
9:   else
10:    context = $false$
11:  end if
12: end while
13: return $cs$
```

Step E: At the end of the expansion of a discovered sequence, if the length of an expanded discovered sequence is less than the threshold value of minimum pattern length, it is discarded from the set of discovered sequences.

Algorithm 8.3.5 Method:DiscoverMatchingNodes()

```
Input: Graph Node ($n$), Graph $G$
Output: Array of Candidate Nodes($CN$)
1: for each graph node $x$ of graph $G$ do
2:   if ($n.id \neq x.id$) then
3:     if (matchOperation($n$, $x$) && matchElement($n$, $x$)) then
4:       if (matchContext($n$, $x$)) then
5:         $DN \leftarrow x$
6:       end if
7:     end if
8:   end if
9: end for
```

Step F: Once the expansion process of discovered nodes is finished, the next step is to find the maximum length of the sequences (max) such that the value of max is greater than or equal to threshold value of minimum pattern length (min_len) and the number of identified sequences is greater than or equal to the threshold value of minimum pattern support (min_sup).

Step F.1: All discovered sequences, whose length is less than the value max, are discarded from the set of discovered sequences. Those discovered sequences whose length is greater than the value max, are truncated to the size max.

Step G: As a last step, a candidate sequence along with discovered sequences is saved as a *domain-specific change pattern* in result list S and the algorithm goes back to step 1 and selects next graph node as a candidate node.

8.3.2 UCP discovery algorithm

To perform a group of change operations in same order over time is very unlikely. In a real world scenario, users perform ontology changes by opting for different orders of change operations. However, the end result (i.e. impact) of the change operation sequences may be the same (i.e. semantically identical sequences). The main difference between *OCP* and *UCP* discovery algorithm is the definition of the search space for a specific graph node search. As the change operations in a sequence can be in an unordered form, the basic idea to discover the unordered complete patterns (Type 3 - c.f. Section 8.1.1) is to modify the node search space in each iteration containing the earlier nodes as well as subsequent nodes based on the permissible node distance value. The pseudo code of *UCP* algorithm is described in algorithm lists (8.3.6 – 8.3.8) and is explained in rest of the section.

8.3.2.1 Description of algorithm

Step A: Similar to the OCP algorithm, the UCP algorithm iterates over each node of the graph and selects it as a candidate node (cn_k), where k refers to the identification key of the node in graph G.

Step B: An iteration process is used to construct a candidate sequence cs by extending candidate node cn_k to its subsequent context-matching nodes cn_{k++}.

Step C: The next step is to identify the discovered nodes dn and add them as a first member to the discovered sequence set DS. There are two main differences in the extension of discovered sequences ds in the UCP and OCP algorithms, i.e.

i. The area of change log graph in which the mapping node will be searched (step D).

ii. Introduction of an unidentified nodes list ul, which will keep record of unidentified candidate nodes.

Algorithm 8.3.6 Unordered Complete Change Pattern Discovery Algorithm

Input: Graph (G), Minimum Pattern Support (min_supp), Minimum Pattern Length (min_len), Maximum *n-distance* (X)

Output: Set of Domain-Specific Change Patterns (S)

```
1:  for $i = 0$ to $N_G.size$ do
2:      $k = 0$
3:      $cs \leftarrow GenerateCandidateSequence(n(g_i))$
4:      if ($cs.size < min\_len$) then
5:          go back to step 1.
6:      end if
7:      $DN \leftarrow DiscoverMatchingNodes(cn_k)$
8:      $DS \leftarrow DN$
9:      if ($DS.size < min\_sup$) then
10:         go back to step 1.
11:     end if
12:     while ($DS.size \geq min\_supp$) do
13:         for each discovered sequence $ds$ in $DS$ do
14:             $t \leftarrow getTargetEntity(ds)$
15:             setSearchSpace($ds$)
16:             $a \leftarrow searchInSpace(ds, cn_{k++}, t)$
17:             if (found) then
18:                 $ds \leftarrow a$
19:                 ascendSequence($ds$)
20:                 setSearchSpace($ds$)
21:                 if ($!ul.isEmpty()$) then
22:                     nodeFound = true
23:                     while ($!ul.isEmpty()$ && $nodeFound$) do
```

```
24:               $nodeFound \leftarrow searchUnidentifiedNodes(ul, ds)$
25:               ascendSequence($ds$)
26:               setSearchSpace($ds$)
27:             **end while**
28:           **end if**
29:         **else**
30:           $ul \leftarrow cn_{k++}$
31:         **end if**
32:       **end for**
33:       **if** $(ds.size < min\_len)$ **then**
34:         discard $ds$ from $DS$
35:       **end if**
36:     **end while**
37:     **for** each discovered sequence $ds$ in $DS$ **do**
38:       **if** $(ds.size < cs.size)$ **then**
39:         discard $ds$ from $DS$
40:       **end if**
41:     **end for**
42:     $P_{domain\_specific} \leftarrow (ds + cs)$
43:     $S \leftarrow P_{domain\_specific}$
44: **end for**
```

Step D: Before the extension process on any discovered node starts, the search space (i.e. the range of graph nodes in which a particular node will be searched) has to be set. The search space is described using two integer variables, i.e. *start_range* (r_s) and *end_range* (r_e) where, r_s and r_e represent the node ids of the starting and ending graph nodes of the search space. The range of the search space can be calculated as;

$$r_s = min(id) - X - 1 \tag{8.1}$$

$$r_e = max(id) + X + 1 \tag{8.2}$$

Where, $min(id)$ and $max(id)$ are the minimum and maximum id values of the existing graph nodes in the discovered sequence ds at any particular iteration.

Step E: New values of r_s and r_e are calculated at the start of each iteration of the discovered node expansion process. For example, given the gap constraint (X) user input value as 1 and a discovered sequence ds contains two graph nodes, $ds = \{n_9, n_{11}\}$ at any particular iteration, then the space in which next candidate node will be searched will be the sequence of graph nodes $n_7 - n_{13}$. As the algorithm scans the whole graph only once (i.e. in step 7 to get the discovered node set) and narrows the search space later, the search space defining technique helps us in achieving a good performance of the algorithm.

Step F: The unidentified nodes list ul keeps record of all candidate nodes which were not matched in the ds expansion process. If a new node is added to the discovered sequence ds, the sequence will be converted into ascending form (based on their id values) and the search space is reset. If the match becomes false and ds is not expanded, the respective candidate node cn_{k++} is added to the unidentified nodes list.

Step G: Once the discovered sequence ds is expanded, an iteration process is applied to the ul to search for the unidentified nodes in the updated search space. If an unidentified candidate node is found and matched (to a discovered node) in the updated search space, the node will be added into the discovered sequence and removed from the unidentified node list. Based on the modified discovered sequence, the values of r_s and r_e are re-calculated.

Step H: At the end of the expansion of any particular discovered sequence, if the length of any expanded discovered sequence is less than the minimum threshold value of pattern length, it must be discarded from the set of discovered sequences.

Step I: In next step, all discovered sequences whose length is less than the length of the candidate sequence are discarded.

Step J: As a last step, a candidate sequence along with discovered sequences is saved as a domain-specific change pattern in the result list S and the algorithm goes back to step 1 and selects next graph node as a candidate node.

Algorithm 8.3.7 Method:setSearchSpace()

Input: Graph G, Discovered Sequence ds, Sequence Gap Contraint X

Output: Updated search space ($minId$ - $maxId$)

```
1: $n1 \leftarrow getFirstNodeOfSequence(ds)$
2: $n2 \leftarrow getLastNodeOfSequence(ds)$
3: $minId = n1.getNodeID() - X - 1$
4: if ($minId \leq 0$) then
5:     $minId = 1$
6: end if
7: $maxId = n2.getNodeID() + X + 1$
8: if ($maxId > G.size$) then
9:     $maxId = G.size$
10: end if
```

Algorithm 8.3.8 Method:Method:searchInSpace()

Input: Graph Node n, Discovered Sequence ds

Output: Updated Discovered Sequence ds

```
1: $t \leftarrow getTargetEntity(ds)$
```

```
2: for each node $w$ in range from $minId$ to $maxId$ do
3:   if $(ds.contains(w))$ then
4:     go back to step 2
5:   else
6:     if $(matchOperation(n, w)$ && $matchElement(n, w))$ then
7:       if $(matchContext(n, w, t))$ then
8:         $ds \leftarrow w$
9:         return $ds$
10:      end if
11:    end if
12:  end if
13: end for
```

8.4 Illustration of results and practical benefits

When ontologies are large and in a continuous process of change, our pattern discovery algorithms can automatically detect change patterns. Such patterns are based on operations that have been used frequently. This reduces the effort required in terms of time consumption and consistency management. Earlier, we presented pattern-based ontology change operators and motivated the benefits of pattern-based change management where patterns are usually domain-specific compositions of change operators. Our work here can be utilized to determine these patterns and make them available for reuse.

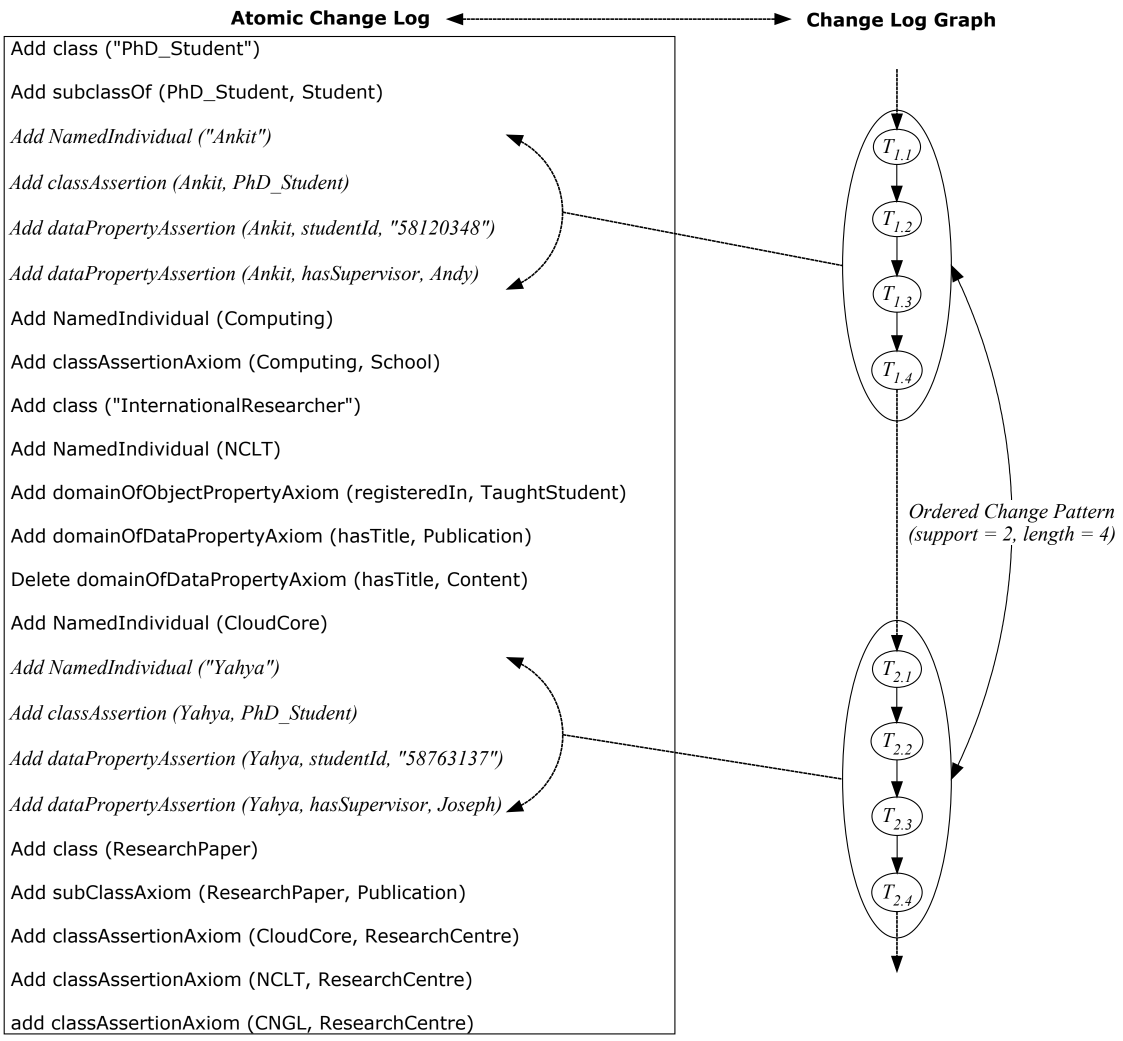
- The key concern is the identification of frequent change patterns from change logs. Generally, these are frequent operator combinations and can result in generic patterns. However, our observation is that many of these are domain-specific, as the example below will illustrate.

- This can be extended to identify semantically equivalent changes in the form of a change pattern. For instance, a reordering of semantically equivalent operations needs to be recognized by the algorithms (i.e. discovery of unordered change patterns).

8.4.1 Illustration of algorithm's results

Figure 8.1 presents a part of a change log session of the university ontology and the identification of a change pattern, represented as a sequence of graph nodes. The frequency of the usage of a change operation set specifies an opportunity for a potential reuse of the set. The change operation set can be extracted and specified as a change pattern and can be applied in the future whenever the same change has to be performed. Identification of change patterns from a change log session of a small size is relatively easy, but as the size of the atomic change log increases, an automated approach for change patterns discovery is a necessity. Furthermore, the change operations present in the change pattern support sequences can either be in the exact same order (i.e. ordered change patterns) or can be unordered (i.e. unordered change patterns). Unordering of change operations makes manual discovery of change patterns more complex.

Two examples from discovered change pattern sequences, one from each level, i.e. ABox-based change patterns and TBox-based change patterns, are given in Tables 8.2 and 8.3. The example in Table 8.2 is the ABox-based change pattern from the university ontology, representing the registration procedure of a new PhD student to the department. First, the student has been registered as a PhD student of a particular department. Then, a student Id, email Id and a supervisor (which is a faculty member of the university) is assigned to the student. At the end, the student is added as a member of a particular research group of the university. We captured such change patterns and stored them in the ontology evolution framework for their reuse. Hence, whenever a new PhD student has to be registered, a stored change pattern can be applied as a single transaction (ensuring cross-ontology integrity constraints to be met).



```
Add class ("PhD_Student")
Add subclassOf (PhD_Student, Student)
Add NamedIndividual ("Ankit")
Add classAssertion (Ankit, PhD_Student)
Add dataPropertyAssertion (Ankit, studentId, "58120348")
Add dataPropertyAssertion (Ankit, hasSupervisor, Andy)
Add NamedIndividual (Computing)
Add classAssertionAxiom (Computing, School)
Add class ("InternationalResearcher")
Add NamedIndividual (NCLT)
Add domainOfObjectPropertyAxiom (registeredIn, TaughtStudent)
Add domainOfDataPropertyAxiom (hasTitle, Publication)
Delete domainOfDataPropertyAxiom (hasTitle, Content)
Add NamedIndividual (CloudCore)
Add NamedIndividual ("Yahya")
Add classAssertion (Yahya, PhD_Student)
Add dataPropertyAssertion (Yahya, studentId, "58763137")
Add dataPropertyAssertion (Yahya, hasSupervisor, Joseph)
Add class (ResearchPaper)
Add subClassAxiom (ResearchPaper, Publication)
Add classAssertionAxiom (CloudCore, ResearchCentre)
Add classAssertionAxiom (NCLT, ResearchCentre)
add classAssertionAxiom (CNGL, ResearchCentre)
```

Figure 8.1: Change log session and associated (graph-based) discovery of change patterns

The example in Table 8.3 is a TBox-based change pattern from the software application ontology, representing the introduction of a new software activity. First, a new class (*TargetEntity_c1*) has been added as a subclass of *Software:Activity*. Later, to perform this activity, a new procedure has been added as a subclass of *Software:Procedure* in the help infrastructure section of the ontology. Finally, the activity and the procedure to perform such activity are linked to each other using an object property *Software:hasProcedure*.

Table 8.2: ABox-based change pattern (extracted from university ontology)

Change Operations
(<TargetEntity_i> <rdf:type> <owl:individual)>
(<TargetEntity_i> <rdf:type> <Univ:PhD_Student>)
(<TargetEntity_i> <Univ:isStudentOf> <Univ:Department_i>)
(<TargetEntity_i> <Univ:StudentID> <xsd:int>)
(<TargetEntity_i> <Univ:EmailID> <xsd:string>)
(<TargetEntity_i> <Univ:hasSupervisor> <Univ:Faculty_i>)
(<TargetEntity_i> <Univ:MemberOf> <Univ:ResearchGroup_i>)

Table 8.3: TBox-based change pattern (extracted from software ontology)

Change Operations
(<TargetEntity_c1> <rdf:type> <owl:class>)
(<TargetEntity_c1> <rdfs:subClassOf> <Software:Activity>)
(<TargetEntity_c2> <rdf:type> <owl:class>)
(<TargetEntity_c2> <rdfs:subClassOf> <Software:Procedure>)
(<Software:hasProcedure> <rdfs:domain> <TargetEntity_c1>)
(<Software:hasProcedure> <rdfs:range> <TargetEntity_c2>)

8.4.2 Practical benefits

Tool Support for Change Tracking: One of the key benefits of our change patterns discovery approach is its integration with an existing ontology change tracking toolkit (such as Protégé, Neon etc.). We incorporated the change capturing and pattern discovery algorithms (as a plugin) in OnE, our Ontology Editing framework. We executed the change pattern discovery algorithms on the recorded ontology changes and discovered change patterns. Users can choose a suitable change patterns from the discovered change pattern list and store them in their user profile. Later, whenever users load that particular ontology, they get the list of stored change patterns in their profile and can apply them in the form of transactions.

Change Request Recommendation: The identified change patterns can also be used for change request recommendations. For example, whenever a user adds a new PhD student to the university ontology, based on the identified *PhD Student Registra-*

tion change pattern, it can be recommended to the user to add *student id*, *email id* of the student and *assign a supervisor* to him/her (using object property *hasSupervisor*). Similarly, in the software application domain, whenever a user deletes certain activity from the domain, deletion of the relevant help files can also be recommended to the user.

8.5 Summary

In this chapter, we presented a graph-based approach for the discovery of the ontology change patterns. Graphs are suitable (to represent ontology change logs) as they can easily be analyzed and are efficiently processable. The ontology changes are formalized using Attributed Graphs (AG) which are typed over a generic Attributed Type Graph (ATG).

We studied the ontology change log graphs empirically and noticed that the users perform identical group of atomic change operations recurrently as a combination of ordered/unordered atomic change operations. Such group of atomic change operations can be presented at a higher level in a form of domain-specific change patterns. The major contribution of this chapter in this regard is the algorithms for the discovery of domain-specific change patterns to support pattern-based ontology evolution.

We identify two fundamental types of change patterns, i.e. *Ordered Change Patterns (OCP)* and *Unordered Change Patterns (UCP).* Ordered change patterns consist of ordered change operation sequences from the change log, such that the node-distance between two adjacent graph nodes of the sequence is a positive (integer) value only. In case of unordered change patterns, the node-distance between two adjacent graph nodes of a sequence, can be positive or a negative (integer) value.

The change pattern discovery algorithms provide support in a number of ways. They can easily integrate within an existing ontology change tracking tool, where the discovered change patterns can be stored in an user profile and can be instantiated whenever a similar group of change operations has to be performed by the user. Discovered change patterns benefited us in terms of identification of correlations and the causal dependencies (which can be integrity constraints) across the ontology taxonomy. Causal dependencies across the ontology subsumption hierarchy can easily be overlooked. Pattern discovery can identify successful changes based on these dependencies. These are association rules that capture non-obvious relationships.

Chapter 9

Experimental results and evaluation

9.1 Evaluation criteria and strategy

9.1.1 Aim

Previous chapters have presented a pattern-driven ontology evolution life cycle that includes change operationalisation (using a layered change operator and pattern framework), recording of ontology changes (using a layered change log framework) and mining of higher level (composite and domain-specific) change patterns. In this chapter, we evaluate the proposed layered ontology change operator and pattern framework, layered ontology change log model and our contribution towards mining of ontology change patterns. The generic international standard model of evaluation process ISO/IEC 14598 [1] (of a software product), supported by the quality measurements, defines a set of evaluation process characteristics [Suryn et al., 2003]. The standard assists in an evaluation where different actors need to understand, accept and trust the results of evaluation. ISO/IEC 14598 is used to apply the evaluation characteristics described in the standard

[1] `http://www.cse.dcu.ie/essiscope/sm4/14598-5.html`

ISO/IEC 9126-1. Considering the defined categorisation for quality characteristics in ISO/IEC 9126-1, we explored quality characteristics from the *usability*, *functional suitability* and *performance efficiency* categories. Standard ISO/IEC FCD 9126-1 defines these characteristics as follows:

- *Usability*: The capability of the solution to be understood, learned, used and liked by the user, when used under specified conditions.

- *Functional suitability*: The capability of the solution to provide functions which meet stated and implied needs when the solution is used under specified conditions. The functional suitability of a solution can be assessed in terms of validity, correctness and completeness.

  Validity: The capability of the solution to resolve the real-world problems faced by the users.

  Correctness: For each given input, the capability of the solution to produce a free of error, accurate output.

  Completeness: The capability of the solution to provide a result if one exists, and if not, to report that no result is available.

- *Performance efficiency*: The capability of the solution to provide the required performance (in terms of processing time), relative to the amount of resources used, under stated conditions.

The main concerns in evaluating the layered ontology change framework is its usability and functional suitability. That is, how useful the proposed solution is and how effectively it solves the problems faced by the users in the real world. The usability of the solution is to be understandable, attractive and useful to the ontology engineers and domain experts. The functional suitability of the solution is to be suitable for answering the real-world problems faced by the domain experts and ontology engineers. In terms of change pattern identification algorithms, correctness, completeness and performance are the key factors for evaluating the efficiency of the algorithms. The completeness of the algorithms is to identify all types of higher-level change patterns from the atomic change log. There must not be any unidentified change patterns left out. The correctness of the algorithms is that the result list of the algorithms should not contain any false identified change patterns and the performance of the pattern identification algorithms is measured based on their process execution time.

9.1.2 Evaluation strategies

Empirical case studies and lab-based experiments, in a controlled environment, can be used to evaluate any software system and to accept or reject the effectiveness of methods, techniques or tools [Easterbrook et al., 2004]. The overall evaluation strategies adopted in our work involve the following three methods:

- *Empirical case studies*: We performed case studies in two different domains in order to examine the evolution of domain ontologies and to evaluate the proposed layered ontology change framework. The framework is evaluated in terms of its usability and functional suitability. The objectives of the empirical case studies and the results are discussed in Sections 9.2 and 9.3.

- *Tool support*: To evaluate the proposed framework, we made use of our ontology editing tool (*OnE*) that we built using OWL API. The tool provides most of the ontology editing functionalities. It allows the declaration of new classes, object properties, data type properties and individuals in the loaded domain ontology. A user can create relations between two or more classes by using defined object and data type properties. The properties can be instantiated at the instance level using assertions. For storage and querying of domain ontologies and the change logs, the tool has been integrated with the RDF Sesame repository. Applied changes and the loaded domain ontologies are stored in the form of triples in a Sesame repository. The ontology change triples are later retrieved and reformulated into an attributed graph for the change pattern identification purposes.

- *Experiments*: We used an experiment-based approach for evaluating the functional suitability and performance efficiency of the change pattern mining algorithms. The results of the lab experiments have been evaluated from both quantitative and qualitative perspectives. In Section 9.4, the results and evaluation of our controlled lab experiments, conducted in order to identify predefined composite changes from the atomic change log, are given. In Section 9.5, the domain-specific change pattern discovery algorithms are evaluated.

9.2 Evaluation of the change operator framework

9.2.1 Objective

The objective is to evaluate the layered change operator framework based on the usability and functional suitability of the solution. In this regard, first we need to confirm that the proposed framework is valid and represents real-world changes. Second, we evaluate how useful the framework is for the ontology engineers and domain experts and how easily it could be understood, learnt and adapt by the users. We evaluate the layered change operator framework using empirical case studies in two different domains. Below, we discuss the experimental setup (in terms of domain selection and ontology development), empirical results and analysis of the empirical results.

9.2.2 Experimental setup

Domain selection: As case studies, the domains *university administration* and *database systems* were taken into consideration.

University ontology: The domain *university administration* was selected because it represents an organization involving people, organizational units and processes. Three main subcategories of the involved people are students, faculty members and the administrative staff. The student category can be subdivided into undergraduate, masters and PhD students. Similarly, faculty members can be categorized into lecturer, senior lecturer, associate professor and (full) professor. The ontology covers the key activities of the university as an organization. At the start of each year, students register for different courses where each course consists of a number of subjects. New students enrol in the university and new student Ids and email Ids are assigned. At the end of each academic year, final year students graduate and their data is removed (from the active version of the domain ontology). Every year, new faculty members join the university and are assigned specific teaching, supervisory and/or administrative duties.

We started building the university ontology in early 2009. In order to construct the first version, we made use of the Academic Institution Internal Structure Ontology (AIISO[2]) and the ontology constructed at Lehigh University[3] [Guo et al., 2005]. The core classes consist of terms such as `College`, `Course`, `Department`, `Module`, `Subject` and `Faculty`. The core properties include `code`, `name`, `teaches`, `description` etc. We kept updating the ontology by using Dublin City University (DCU[4]) as an example, but also included other realistic constructed items in order to broaden the experimental base. We conceptualized most of the activities and the processes in DCU for the construction of the ontology. We added new classes, object properties, data properties and, most importantly, populated the ontology by adding data at the instance level.

To perform the case study on a small scale, we loaded two types of instance level data (of our research centre *CNGL*[5]) into our university ontology, i.e. research group members (people) and published content (publications). Currently, the ontology consists

[2]`http://vocab.org/aiiso/schema#`
[3]`http://swat.cse.lehigh.edu/onto/univ-bench.owl`
[4]`www.dcu.ie`
[5]`www.cngl.ie`

of 61 classes, 22 object properties, 15 data properties and over 450 named individuals. The university domain ontology is reasonably stable at the schema level. We observed very few schema level changes (in comparison to instance level changes) in the domain. These changes include merging of multiple schools under a single department, addition of new research groups and addition of new faculty types. In contrast, changes occur at the instance level on a daily basis. Recording details about all the defined individuals into a database defines a large knowledge base. In this regard, the university domain ontology corresponds to OWL 2 QL profile[6]. The current version of the (schema level) university ontology is given in appendix D.

Database ontology: The *database systems* is a technical domain that can be looked at from different perspectives – for example being a topic in a course curriculum or a textbook on the subject.

The database textbook ontology was derived from the taxonomy arising from the table of content and the index of a database course book [Elmasri et al., 2006]. The classes are categorized based on the prescribed chapters in the course book and different broader/narrower relationships have been defined representing the pre-requisite topics and chapters. These broader/narrower relationships depict that the topic of a given chapter is necessary to understand the advanced topic available in another chapter. Such relationships supported an instructor in constructing an outline of the database course.

The technical database domain ontology was developed by the domain experts [Boyce et al., 2007]. The ontology consists of 109 classes, 31 object properties and in total more than 100 axioms. Classes in the database domain ontology include `database language`, `database model`, `algebra operations`, `update operations`, `relational calculus`, `logical operators` and `relational database`. The class `database language` is fur-

[6]OWL 2 QL profile is aimed for the applications that consist of very large size of instance level data.

ther categorized into `data definition language` (DDL), `data manipulation language` (DML), `storage definition language` (SDL) and `view definition language` (VDL). Similarly, the `database model` class is further categorized into `object-based logical models`, `physical data models` and `record-based logical models`. The class `update operations` is divided into `delete`, `insert` and `modify` subclasses. Object properties are used to define the relationships between them. The database system domain ontology corresponds to OWL 2 EL profile[7]. The current version of database ontology (technical) is given in appendix E.

9.2.3 Results

In this section, first we outline the empirical observations on the evolution in the university domain ontology and later discuss the user-based evaluation of empirical observations in general.

9.2.3.1 Empirical observations (university ontology)

The objective of the university domain ontology is to help administer the proper execution of the day-to-day activities of a university. In this sense, the university ontology is used to represent the university content as a large-scale information system. Changes at the instance level happen on a daily basis and less frequently and more irregularly at the schema level. A list of observed inclusion type changes from the university ontology is given in Table 9.1. The table is based on the actual changes carried out in DCU. The frequent changes involve joining or leaving of faculty, student or staff, introduction of new courses, organisation of events etc.

Empirical observations showed that at the start of the ontology development process, conceptual level (i.e. schema level) changes were much larger in number compared to instance level changes, but they gradually decreased (Figure 9.1). Creating the hierarchy

[7]OWL 2 EL profile is aimed for the applications that contain very large number of classes and properties.

Table 9.1: A list of observed changes in university domain ontology

Changes in the Domain Ontology (University)	Changes in the Domain
Add individual (X)	Add a new Faculty
Add classAssertionAxiom (X, Faculty)	
Delete classAssertionAxiom (X, Faculty)	Remove a Faculty
Delete individual (X)	
Add individual (X)	Enrol a new undergraduate Student
Add classAssertionAxiom (X, UGStudent)	
Add dataPropertyAssertionAxiom (X, emailId, XSD:String)	Assign an email Id
Add dataPropertyAssertionAxiom (X, studentId, XSD:Int)	Assign a student Id
Add individual (X)	Enrol a new PhD Student
Add classAssertionAxiom (X, PhD_Student)	
Add objectPropertyAssertionAxiom (X, hasSupervisor, type:Faculty)	Assign a supervisor to PhD Student
Add objectPropertyAssertionAxiom (X, isMemberOf, type:School)	Assign a School to the Student
Add class (X)	Merge two Departments into a single Derpartmet
Add subClassOfAxiom (X, type:Department)	
Transfer Roles (X, (X1, X2))	
Delete subClassOfAxiom (X1, type:Department)	
Delete class(X1)	
Delete subClassOfAxiom (X2, type:Department)	
Delete class(X2)	
Add individual (X)	Add a new Course
Add classAssertionAxiom (X, Course)	
Add dataPropertyAssertionAxiom (X, courseCode, XSD:String)	Assign a Course Code
Add individual (X)	Add a new Staff Member
Add classAssertionAxiom (X, StaffMember)	
Add objectPropertyAssertionAxiom (X, isMemberOf, type:Department)	Assign a department
Add objectPropertyAssertionAxiom (X, isManagerOf, type:Department)	Assign a managerial role
Add individual (X)	Create a new university event
Add classAssertionAxiom (X, Event)	
Add dataPropertyAssertionAxiom (X, time, XSD:DateTime)	Add details of the event
Add objectPropertyAssertionAxiom (X, venue, type:Building)	
Delete objectPropertyAssertionAxiom (X, isMemberOf, type:Department)	Change Department of a Staff Member
Add objectPropertyAssertionAxiom (X, isMemberOf, type:Department)	
Add individual (X)	Create a a new vacancy
Add classAssertionAxiom (X, Vacancy)	
Add dataPropertyAssertionAxiom (X, descrip, XSD:String)	Add description of a vacancy
Add class(X)	Add a new Faculty category
Add subClassOfAxiom(X, Faculty)	

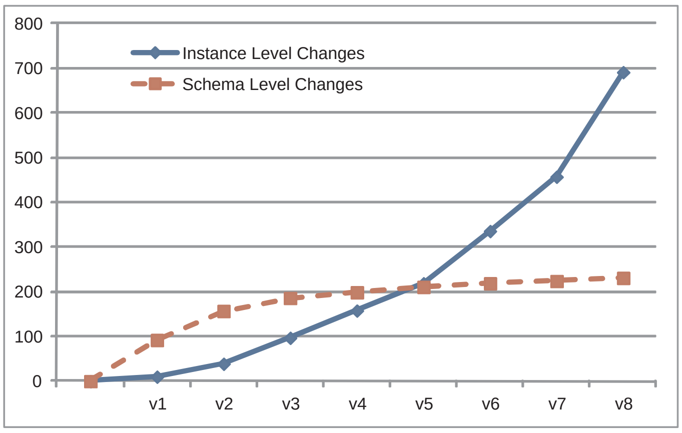


Figure 9.1: Schema level vs. instance level ontology changes (*university ontology)*

that represents the domain's core elements is a once-off event. Therefore, the cost is high only for the first few versions of the ontology.

In subsequent versions, addition of the students, faculty, courses, events, staff etc. (at the instance level) is the actual function of the ontology. This is quite realistic as at the start of the ontology development, ontology engineers add more content at the schema level and once the schema level is stable, the next step is for content managers to provide the data to populate the ontology at instance level. The cost of adding such information is part of the running cost of the ontology.

9.2.3.2 User-based evaluation of the framework

Different levels of change patterns emerge by clustering the empirically observed frequent changes in the domain ontology. These change patterns are useful for the ontology engineers to modify domain ontologies more easily and more correctly. We evaluated the usability and functional suitability of the framework in two steps.

> *Step one:* We involved five ontology engineers in the evaluation of the framework in terms of its change operational cost. The change operational cost has been evaluated in two ways, i.e. in terms of the number of steps to be performed and

the time required to perform the specified steps. To do so, we selected eight different ontology change operations, two from the atomic level (level one), four from the composite level (level two) and two from the domain-specific level (level three) - Table 9.2.

Table 9.2: List of change operations and their type

No.	Type	Change
1	Atomic	Add class (Lecturer), Add subClassOf (Lecturer, Faculty)
2	Atomic	Add individual (John), Add classAssertion (John, UGStudent)
3	Composite	Split class (ResearchStudent, (MSByResearchStudent, PhDResearchStudent)), *Strategy: Split the Roles*
4	Composite	Merge classes ((MSByResearchStudent, PhDResearchStudent), ResearchStudent), *Strategy: Aggregate all roles*
5	Composite	Copy class (ResearchStudent, ResearchIntern, Researcher)
6	Composite	Split class (ResearchStudent, (MSByResearchStudent, PhDResearchStudent)), *Strategy: Attach to both classes*
7	Domain-specific	PhD Student Registration (Tylor Kane, 58106382, tylor@computing.dcu.ie, Joe Morris, Computing, CNGL, Irish)
8	Domain-specific	Add New University Event (AICS 2012, ResearchEvent, 17 Sep 2012, 19 Sep 2012, 23rd Irish Conference on Artificial Intelligence and Cognitive Science, Ray Walshe, aisc2012@comuting.dcu.ie, +353-1 700 597)

Operational cost in terms of number of steps: First, we evaluated the framework on the basis of the number of steps required to perform the specific changes given in Table 9.2. To do so, we make use of our Ontology Editor (OnE) and widely used Protégé[8] framework. Results are given in Figure 9.2 in the form of a bar chart. It is evident that in the case of atomic level change operations, both frameworks require the same number of steps to be performed. However, usage of evolution strategies and pattern-driven data entry forms (for performing higher-level change operations) significantly reduces the evolution effort in terms of the number of required steps. For example, in the case of the composite change operation `Merge classes` (change 4), the ontology engineer needed to perform eight (8) steps (in OnE) in comparison to fifteen (15) steps in Protégé. The biggest difference was seen in the case of the `Split class` composite change operation

[8]`http://protege.stanford.edu/`

where the selected strategy was to join the roles to both the newly added classes (change 6). The result is fairly understandable as in the case of Protégé, ontology engineers needed to attach each role one after the other and hence increased the number of required steps. The more the roles (to be attached), the more the steps it requires. On the other hand, in the case of OnE, ontology engineers only needed to select the appropriate evolution strategy and all the roles were automatically attached to the newly added split classes. Hence, increase or decrease of roles (to be attached) does not have any effect on the number of required steps.

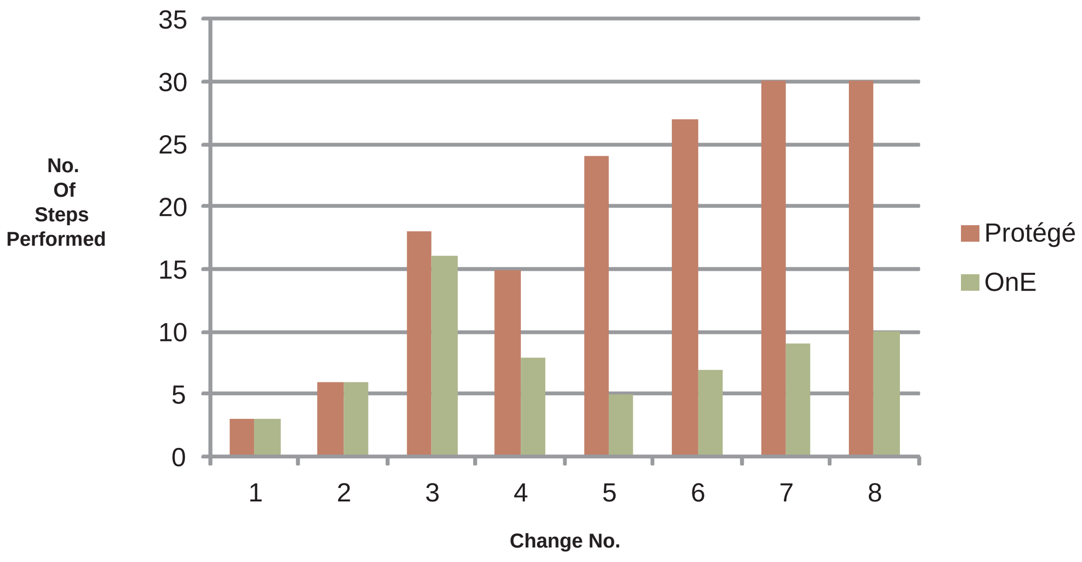


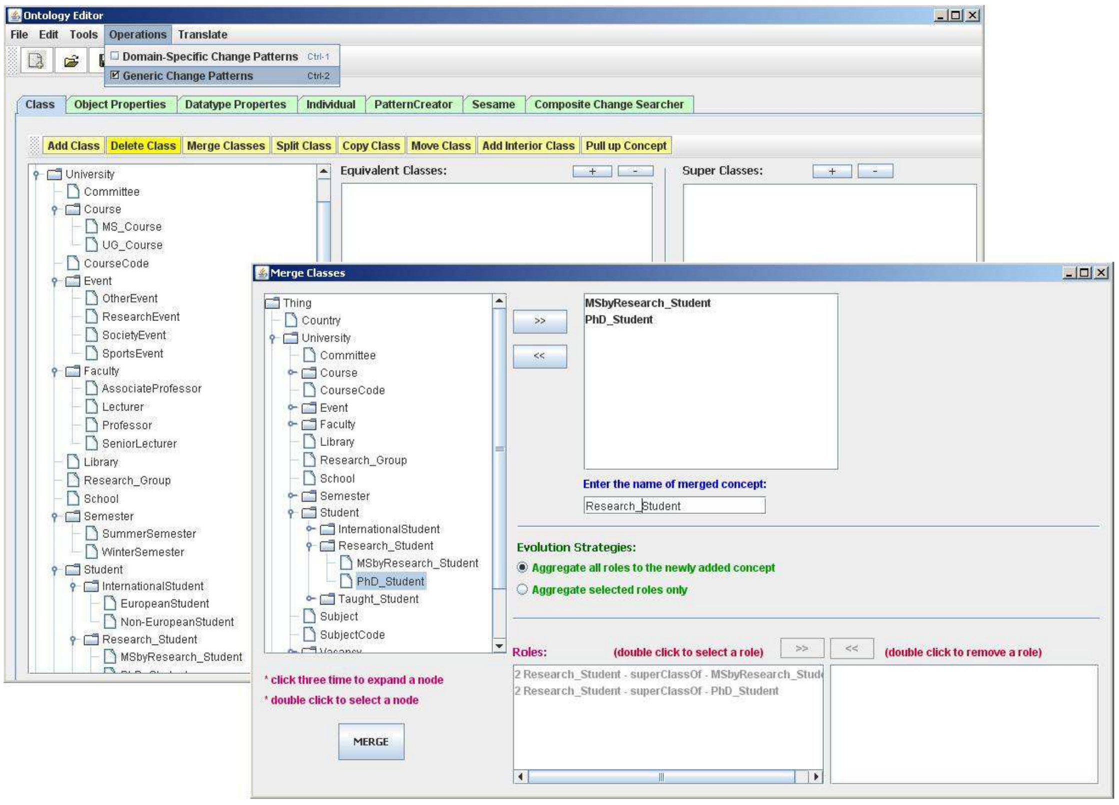
Figure 9.2: Protege Vs. OnE - Number of steps performed

Operational cost in terms of time: Second, we evaluated the framework based on the time required to perform the different level of change operations. We compared the time taken by the ontology engineers (minimum, maximum and average) for performing the changes in both ontology editing frameworks. The performance comparison is given in Table 9.3. Learning effects on the performance of different users have been considered and factored into our controlled experiment. We observed that on average the time occupied by the two ontology editing frameworks to perform an ontology change using atomic change operators, is in a similar range; however, the usage of higher level change operators and the evolution strategies had a reasonable impact on the required time (change nos. 3–6). For example, in the case of performing `Merge classes` (change 4) in Protégé, ontology engineers needed to attach each role one after the other. As we mentioned earlier, the more roles, the more time it is going to take. On the other hand, selecting evolution strategy "Aggregate all roles" (by one single operation - c.f. Figure 9.3) reduced the time required for attaching all the roles. Similarly, in the case of `Split class` (change 6), by selecting the evolution strategy "Attach to both classes" the user did not need to attach the roles to the split classes one after the other.

Table 9.3: Comparison between OnE and Protégé (min:sec)

Change No.	**Protégé**			**OnE**		
	Min.	Max.	Avg.	Min.	Max.	Avg.
1	0:03	0:12	0:06	0:04	0:10	0:06
2	0:11	0:34	0:21	0:07	0:24	0:17
3	0:55	3:21	1:53	0:22	2:08	0:57
4	0:35	1:18	1:05	0:11	0:39	0:20
5	0:37	1:55	1:09	0:07	0:21	0:12
6	1:03	1:42	1:26	0:09	0:39	0:19
7	0:51	2:34	1:40	0:17	1:40	0:57
8	1:26	2:59	2:00	0:31	1:52	1:08

Step two: While the selection of evolution strategies makes the evolution process fast and reduces the effort, the user-based evaluation of change patterns confirms that using change patterns as a defined data entry forms also makes the evolution process intuitive and simple. We requested ontology engineers to complete a questionnaire based on their experience of performing specified ontology changes by utilizing the change patterns. Users agree that the framework is easy to understand and reduces the complexity in terms of change operationalization. The questionnaire and results of step 2 are discussed in Section 9.3.3.



MSbyResearch_Student
PhD_Student

Enter the name of merged concept: Research_Student

Evolution Strategies:
[✓ Aggregate all roles to the newly added concept]
Aggregate selected roles only

2 Research_Student - superClassOf - MSbyResearch_Stud
2 Research_Student - superClassOf - PhD_Student

Figure 9.3: Framework window of “merge” composite change operator

9.2.4 Discussion

Based on our empirical observations of common changes in the domain ontologies and different perspectives of the users towards a domain, we studied the patterns that are common, resulting in a layered change operator framework, as discussed in Chapter 4.

The change operators and patterns are based on actual changes being carried out by ontology engineers and observed by us in both the university administration and database systems ontologies. This makes the proposed change operator framework valid from a practical perspective. We identified different levels of change operations for those who focus on the generic as well as the domain-specific changes. In this regard, the change operators at lower levels (i.e. level one and level two) are generic and can be applied in any domain ontology. The change patterns at level three and level four need to be customized. These change patterns can be used as data entry forms, in order to apply some common and frequent changes in the application domain. Performing such frequent changes, in a form of change patterns, makes the ontology change management faster and easier.

We observed during the user-based evaluation that ontology engineers perform semantically equivalent composite change operations using different orders of atomic level change operations (c.f. Section 6.3). Thus, the empirical results confirm that the lower-level (level one and two) change operators are useful to ontology engineers to suitably define their own change patterns, i.e. provide an adequate customization solution. Domain experts (who have less knowledge of the underlying ontology language and its specification) can use the level three domain-specific change patterns and customize them to meet their requirements. In such cases, domain-specific change patterns are at the right level of abstraction and consequently, more useful for the domain experts. This proves the functional suitability of the proposed change operator framework. Change patterns are useful for reducing maintenance workload, especially for ontologies that change very frequently and in a systematic way. It will be costly for instance that for a new addition one needs to restructure (using lower level change operators) the whole hierarchy, which can be expensive to develop, test and validate, and then redeploy. A systematic approach, such as the use of change patterns, is a necessity here.

We evaluated the usability of the framework from its implementation's point of view. To do this, we deployed a few higher level change patterns in our ontology editing framework (OnE) as an optional setting and involved five ontology engineers in order to test the usability of the framework. An ontology engineer can select an appropriate change pattern from the given list. Each defined change pattern consists of its own user interface. Figure 9.3 explains the use of an implemented *merge* composite change pattern from the given list. Once the merge change pattern is selected, the ontology engineer can select the entities to be merged. Interface support (such as tool tips etc.)

is implemented to support a transparent, easy and understandable evolution process. As we discussed above, different ontology engineers may have different perspectives on a domain ontology, different evolution strategies (c.f. Section 7.3) have been implemented. This makes the evolution process customizable to meet the needs of an ontology engineer and significantly reduces the manual effort. The feedback from the ontology engineers (c.f. Section 9.3.3) acknowledged that the system is easy to understand, learn and can easily be adopted. The results in Table 9.2 illustrate that the use of change patterns (along with the evolution strategies at the composite level) reduces the required effort in terms of time and consistency management and gives a free hand to evolve the ontology based on their own needs.

In summary, the effectiveness of an ontology change is considerably dependent on the granularity, how the change operators are combined and the extent of their impact on the domain ontology. We proposed a layered change operator framework (in Chapter 4) where change operators can be atomic, composite or domain-specific. In this section, we evaluated the layered change operator framework in terms of its validity, efficiency and usability. We performed the empirical case studies in two domains. The changes in the two domains had been empirically observed resulting in a layered operator framework. The results indicate that the framework is valid and adequate to efficiently handle ontology evolution. While ontology engineers who are more interested in step by step changes and defining their own change patterns, other users can focus on domain-specific changes at level three. Higher level change patterns are customizable and are based on the actual changes carried out in the domains, which makes them functionally suitable to meet the needs of a user. The proposed operator framework has been implemented in our ontology editing framework.

9.3 Evaluation of the layered change log model

9.3.1 Objective

The objective here is to evaluate the layered change log model based on its functional suitability. The changes must be represented in such a way that it is useful and understandable by the domain experts and ontology engineers. In this regard, the functional suitability of the layered change log model is, first, to maintain a comprehensive understanding of the evolution of domain ontologies and, second, to explicitly present the intent behind any applied change. We made use of user feedback as a metric in order to evaluate and answer the specified questions. Below, we discuss the experimental setup (in terms of change log construction) and the results.

9.3.2 Experimental setup

To validate the change log model, we made use of the existing empirical case study data from the university domain ontology. Our ontology editing framework provides functionality to record the ontology changes in a repository (Figure 9.4). We recorded the ontology changes in the form of triples and in order to construct the change triples, we made use of a metadata model using the OWL language (c.f. Section 5.2). We used an RDF-based triple store to record the changes applied to the university ontology. The change log framework works in line with the proposed layered change operator framework. If a user makes use of level one atomic change operators, changes are being recorded in the atomic change log (c.f. Table 5.1) and if a user makes use of a change pattern, the change is recorded in the pattern change log (c.f. Table 5.2).

9.3.3 Results

We utilized a user-based evaluation, as discussed in Section 9.2.3.2, in order to empirically evaluate the layered change logs. The ontology engineers performed the given

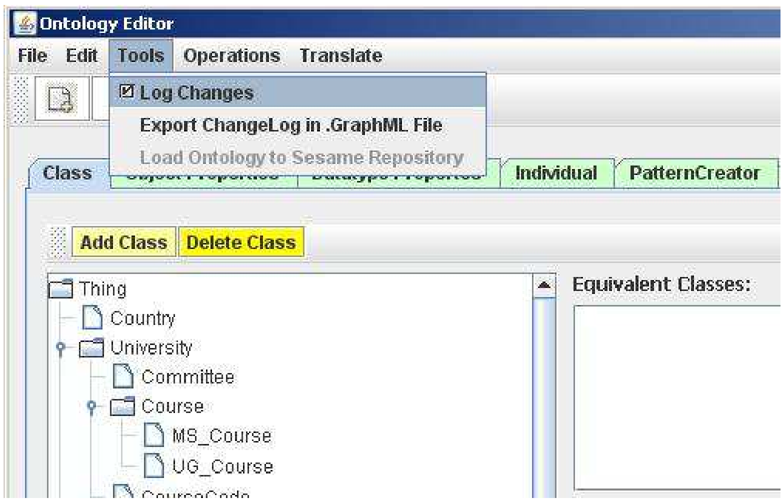


Figure 9.4: Framework window - ontology change logging

change operations using atomic change operators and higher-level change patterns. The applied changes had been logged into atomic and pattern change logs accordingly. We presented the two change logs to the users in order to manually analyze how the changes have been recorded and represented in layered logs. Do the layered logs maintain a fine-grained representation of ontology changes? Is representation of ontology change at a higher-level in the form of patterns more intuitive? To answer these questions, we involved five ontology engineers who have expertise in the area of software engineering and large databases. We requested the participants to give a rating to the claims we made about the functional suitability and usability of the layered change log framework. The claims are rated separately by each ontology engineer from 1 to 5 (where rating 5 represents “strongly agree”, 4 “agree”, 3 “neutral”, 2 “agree” and 1 “strongly agree”). The claims and the user-based ratings (average) are given in Table 9.4.

The feedback from the participants confirm that the solution is useful and functionally suitable for the ontology engineers and domain expert. The highest rating was given to claim 1 (i.e. ACL presents a complete fine-grained representation of ontology changes). Participants agree that the representation of the ontology changes at a higher

Table 9.4: Questionnaire-based evaluation of the layered framework

No.	Claim	User's feedback (1 to 5) - Average
1	Atomic Change Log (ACL) presents a complete and fine-grained representation of applied ontology changes (***Completeness***).	4.67 (93.33%)
2	Pattern Change Log (PCL) supports in understanding the intent behind an applied ontology change (***Validity***).	4.00 (80.00%)
3	The ontology changes recorded in ACL and PCL are easily understandable (***Validity***).	4.33 (86.67%)
4	Recording of domain ontology and change log in a single RDF repository allows the user to concurrently navigate through them (***Functional suitability***).	3.67 (73.33%)
5	The framework is easy to understand, learn and use (***Usability***).	4.33 (86.66%)
6	The customizable evolution strategies allow users to evolve the ontology based on their own needs (***Adequacy***).	4.33 (86.66%)

level helped them to understand the intent behind the applied changes - making the solution practically valid. The lowest rating was given to claim 4 (i.e. recording of ontology and change logs in a single RDF repository allows user to concurrently navigate). Participants agree that the framework does allow concurrent navigation. However, a graph-based illustration of associations between change log and domain ontologies would be more intuitive for the users.

9.3.4 Discussion

The layered change log maintains the structural and semantic representation of ontology changes at two separate levels without losing their interdependence. On the one hand, the atomic change log is used for a fine-granular representation of applied ontology changes. Each atomic change is recorded in a form of a triple set where each triple represent a single attribute of the applied change. Furthermore, as change patterns are not only recorded in the pattern change log, but also at the atomic level, the atomic change log depicts a complete representation of the applied ontology changes. On the other hand, the pattern change log presents a higher level picture of the ontology evolution. PCL represents the user's intent of applied ontology changes more explicitly. To do so, the attributes such as `PatternName`, `PatternPurpose` are being attached to each recorded change pattern (c.f. Section 5.3.2).

The RDF-triple format supports fine-grained representation of applied ontology changes. Such storage of ontology changes at a fine-grained level is adequate in terms of extracting specific knowledge from the change logs. Recording of metadata details allowed us to learn about the users as an additional advantage. One can generate user profiles and can understand in which section (or entities) of domain ontology the users are most interested in and thus can populate the domain ontology accordingly.

Recording the ontology changes at a higher level as change patterns, helps in knowing the intent and the impact of ontology changes more precisely. Thus, more accurate evolution strategies can be applied to keep the schema-level and instance-level data consistent and valid. In the case of the `Pull up property` composite change (Figure 9.5), a user can select an evolution strategy *"do nothing"* as the previous property instantiations are still valid or *"assert the instances explicitly as defined instances of earlier domain class"* to not lose any existing knowledge, rather than deleting all previous instantiations. In the case of the `Pull down property` composite change, a user can *"revalidate"* as some of the previous property instantiations will still be valid. Thus, rather than deleting all of the previous instantiations, it is better to revalidate manually and delete only those which are not valid anymore.

Our research is not only focused on determining the ontology differences between the versions, but also how it has changed from an operational perspective and to support an ontology engineer in executing the changes (through identified patterns). We conducted experiments (discussed in next two sections) on a number of atomic change log case scenarios empirically, in order to identify the frequent (composite and domain-specific) change patterns. The results acknowledged that the proposed layered change log model facilitates a structured ontology evolution process.

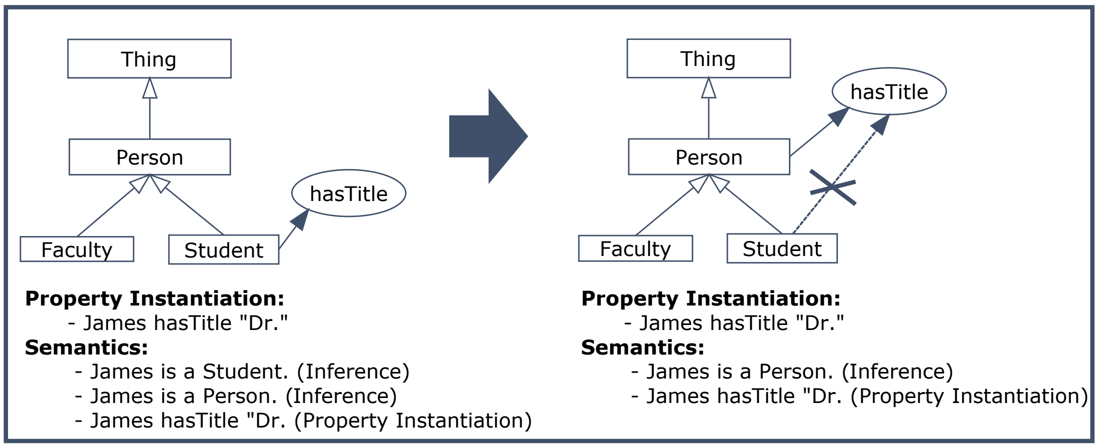

Thing
Person
Faculty
Student
hasTitle
Property Instantiation:
- James hasTitle "Dr."
Semantics:
- James is a Student. (Inference)
- James is a Person. (Inference)
- James hasTitle "Dr. (Property Instantiation)
Thing
Person
Faculty
Student
hasTitle
Property Instantiation:
- James hasTitle "Dr."
Semantics:
- James is a Person. (Inference)
- James hasTitle "Dr. (Property Instantiation)

Figure 9.5: Pull up property - composite change case scenario

9.4 Evaluation of composite change detection algorithms

9.4.1 Objective

The composite change detection algorithm given in Chapter 7 identifies the occurrences of pre-defined composite changes in an atomic change log. To do so, the atomic change log was transformed into a linear sequential graph and a graph-based matching algorithm was applied to identify the defined composite changes. The aim here is to evaluate the given composite change detection algorithm based on its performance. In this regard, we selected correctness and completeness as the two main evaluation criterion.

9.4.2 Experimental setup

We measured the completeness and correctness of our composite change pattern detection algorithms by comparing their results with the manual approach. In this regard, we gathered five ontology engineers and gave them a brief description of the domain (i.e. *university administration*), the *composite changes* and their definitions. Once the ontology engineers had a clear idea about the domain ontology and the composite changes, we performed the evaluation in three steps:

- Step 1: Amongst the ontology engineers, we distributed five sessions of the ontology change log and asked them to identify the discussed composite changes from these change log sessions. To perform the evaluation on a small scale, we selected only six types of composite change patterns (i.e. split class, add specialize class, group classes, add interior class, pull up property and pull down property) and selected a small subset of the atomic change log (i.e. 120 atomic ontology changes).

- Step 2: At the end of step 1, we gave the ontology engineers the results of our controlled experiments (i.e. results of the automated approach) and asked them to testify whether the detected composite changes are valid - (correctness).

- Step 3: In last step, we asked ontology engineers to verify their results against the results of the auto approach - (completeness).

9.4.3 Results

A complete list of identified composite change patterns is given in appendix H. Table 9.5 gives the details of the comparison between manual and automated detection of composite change patterns. Here in the table, the term *"candidate"* change pattern refers to the identified change patterns that as a whole or partially can be acknowledged as a composite change pattern. The candidate change patterns identified through the manual or automated approach need to be reviewed again by an expert ontology engineer, before confirming them as a correctly identified composite change pattern.

The comparison of manually identified change patterns with the automated approach confirms the completeness of the algorithm, i.e. there is no single change pattern omitted by the algorithm (row 6 - Table 9.5). Further, the feedback of ontology engineers in step two of the experimental setup, where we requested the ontology engineers to verify the identified change patterns, confirms that identified change patterns are correct and valid (row 7 - Table 9.5). The ontology engineers were able to identify ten composite changes

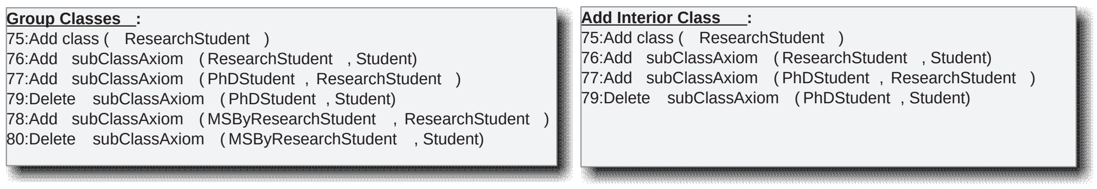
Table 9.5: Comparison between manual vs. automated composite change detection

No.	**Type**	**Manual**	**Automated**
1	Change Log size	120 atomic changes	
2	Total identified change patterns	10	15
3	sub change patterns	0	4
4	Candidate change patterns	1	1
5	Complete change patterns	9	10
6	Missed change patterns	1	0
7	False change patterns	0	0
8	Time taken	$>$ 55 min	$<$ 1 sec

in comparison to the automated approach where the number of detected composite changes was fifteen. The ontology engineers were able to identify almost all complete change patterns (row 5 - Table 9.5), but the main difference lies in three cases, i.e. i) the time taken to identify these changes (from a small change log), ii) the omitted composite change patterns having positive *n-distance* (c.f. Section 6.3.2) and omitted overlapped change patterns.

- The ontology engineers took almost an hour to go through a small subset of atomic ontology changes and to identify correct change patterns (row 8 - Table 9.5). This result shows that identifying composite change pattern manually is possible, but at a very high cost of time consumption (as the ontology engineers took almost thirty seconds to go through and relate a single atomic ontology change with other changes).

- All ontology engineers missed the identification of an *"add specialize class"* composite change pattern, due to the availability of a few extra change operations in between the change operations of the composite change (row 6 - Table 9.5). This shows that the manual identification of a composite change pattern where all the atomic change operations are in a sequence with zero n-distance between them, is relatively easier in comparison to the identification of a composite change pattern where atomic change operations have some positive node distance between them.



```
Group Classes :
75:Add class ( ResearchStudent )
76:Add subClassAxiom ( ResearchStudent , Student)
77:Add subClassAxiom ( PhDStudent , ResearchStudent )
79:Delete subClassAxiom ( PhDStudent , Student)
78:Add subClassAxiom ( MSByResearchStudent , ResearchStudent )
80:Delete subClassAxiom ( MSByResearchStudent , Student)
```

```
Add Interior Class :
75:Add class ( ResearchStudent )
76:Add subClassAxiom ( ResearchStudent , Student)
77:Add subClassAxiom ( PhDStudent , ResearchStudent )
79:Delete subClassAxiom ( PhDStudent , Student)
```

Figure 9.6: Overlapping between identified composite change patterns

- We observed four cases in the result list where an identified composite change was also detected completely or partially as a different category of identified composite change (row 3 - Table 9.5). In other words, a subset of a composite change fulfills the conditions (to be identified) of another category of composite change. This finding acknowledges our specification of a pattern change log, where a change pattern can overlap (completely or partially) with other identified change patterns. An example of such overlapped change patterns is given in Figure 9.6. The identification of such sub change patterns were missed by the manual approach.

9.4.4 Discussion

Identification of composite change patterns (discussed in Chapter 7) not only helps in understanding the evolution of domain ontologies, but also reduces the manual effort required in terms of time and consistency management. In this sense, the identified change patterns can be utilized as pre-defined change patterns to perform specific composite tasks in a specific way. Furthermore, based on the identified composite changes, more appropriate (composite level) strategies can be employed in order to keep the validity and consistency of the ontology and instances. In this section, we evaluated the composite change pattern detection algorithms in terms of their performance. To do so, we performed a user case study where we compared the results of the composite change detection algorithm with the manual approach in terms of its correctness and completeness. In comparison to manual approach, the automated approach in identifying the composite change patterns is beneficial in different ways. This is because the manual identification of change patterns from a large scale ontology change log is time consuming and practically infeasible. As the size of the change log increases, the time required to identify composite change pattern manually will increase intensively and using some automated approach is inevitable. Furthermore, the manual approach is error-prone. While the results of the algorithms were complete, ontology engineers failed to manually identify the overlapping change patterns and the change pattern with a sequence gap. The identified change patterns can be utilized in an ontology editing framework as a recommender system. The output of the algorithm was verified by taking ontology engineers feedback to confirm the correctness and the completeness of the algorithm.

9.5 Evaluation of change pattern discovery algorithms

9.5.1 Objective

In Chapter 8, we presented the domain-specific change pattern discovery algorithms for ordered complete (OCP) and unordered complete (UCP) change patterns. To do so, we transformed the atomic change log into a linear sequential graph (c.f. Section 6.2) and a graph-based pattern discovery approach was utilized. In this section, we evaluate the pattern discovery algorithms based on three criteria, i.e. effectiveness, efficiency and correctness. We measured the effectiveness of the algorithms in terms of the number of identified change patterns (quantitative). The efficiency of the algorithms has been measured based on the processing time (speed) and the correctness of the algorithms has been evaluated in terms of correctly identified change patterns (qualitative).

9.5.2 Experimental Setup

Change pattern discovery algorithms identify the change patterns from the ontology change log graph based on the input threshold values of minimum length of change pattern (*min_len*), minimum support of a change pattern (*min_supp*) and the permissible node-distance (c.f. Section 6.3.2) between two adjacent nodes of a change pattern sequence in a change log graph. The evaluation of the algorithms has been achieved in three separate steps.

- Step 1: The effectiveness of the two algorithms has been measured in terms of number of identified change patterns (quantitative). We analyzed the effect of varying the input parameter values on the overall results. To do so, for a fixed minimum pattern support value, we varied the threshold values for the minimum pattern length and the permissible node-distance and compared their results.

- Step 2: In order to evaluate the efficiency of the two algorithms, we kept the minimum pattern support (*min_supp*) and minimum pattern length (*min_len*) static and varied the node distance value and compared the algorithm's results. This allowed us to analyze how an increase in the permissible sequence gap affects the number of identified change patterns and the overall processing time.

- Step 3: In order to verify the correctness of the results of the two algorithms, we made use of ontology engineers feedback. The ontology engineers looked into the discovered change patterns manually and examined how many of them are sound and correct (qualitative).

All experiments with the change pattern discovery algorithms was conducted on a 3.0 GHz Intel Core 2 Duo CPU with 3.25 GB of RAM, running MS Windows XP. We used SPARQL queries in order to capture more than five hundred ontology changes from the university atomic change log (>5000 log triples) and converted them into a linear graph using a graph API. We utilized our algorithms to discover the domain-specific change patterns in ontology change log graphs.

9.5.3 Results

Quantitative evaluation: The results of the two algorithms, in terms of number of identified change patterns, are given in Figures 9.7 and 9.8. The results show that in both cases (i.e. ordered and unordered change pattern discovery approach) the number of identified change patterns (P) increases with the increase in the permissible node distance (N-distance) value and decreases with the increase in the threshold value of the pattern length (min_len) (Figure 9.7–9.8). Thus, we can say that P is directly proportional to N-distance (P $\propto$ N-distance) and inversely proportional to minimum change pattern length (P $\propto 1/min\_len$).

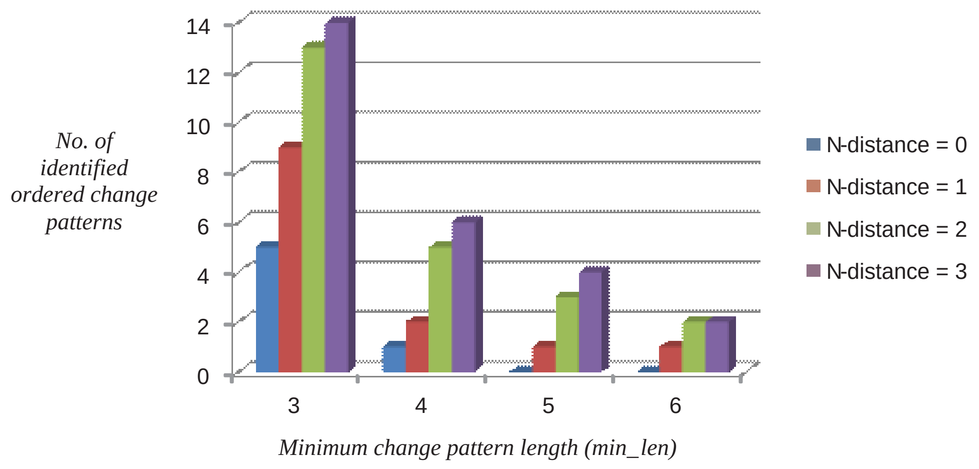


Figure 9.7: No. of identified ordered complete change patterns (Quantitative)

These results are quite realistic as the increase in the permissible node distance value increases the range where a change pattern can be identified. This also allows the capturing of those patterns that have a few extra ontology changes within them. Similarly, the increase in the minimum pattern length threshold makes sure that the discovered change patterns are of greater size and cover most of the identical change sequences, but in doing so the number of identified change patterns decreases.

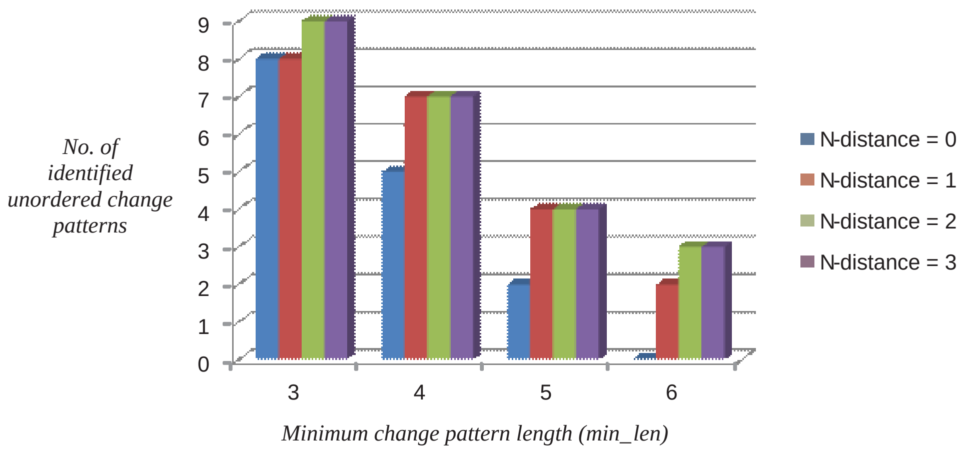


Figure 9.8: No. of identified unordered complete change patterns (Quantitative)

Efficiency-based evaluation: The comparison between the two algorithms in terms of time consumption is given in Table 9.6. Though the UCP algorithm takes more time to process the change log data, it is more effective in terms of numbers of discovered change patterns. It discovers more change patterns in comparison to OCP. Similarly, in terms of the size of maximal patterns and coverage of identical change sequences, UCP is superior to OCP.

Table 9.6: Comparison b/w OCP and UCP algorithms with minimum pattern support (min_supp) = 5 and minimum pattern length (min_len) = 5.

	a - **OCP Algorithm**			b - **UCP Algorithm**		
Node Dist.	Patterns Found	Time (ms)	Seq. in a Pattern	Patterns Found	Time (ms)	Seq. in a Pattern
0	0	469	0	4	1359	9
1	3	609	8	7	2282	13
2	5	875	16	6	3906	18
3	5	985	15	8	4968	21
4	5	1110	17	8	6078	21
5	5	1203	17	9	7141	21

Qualitative evaluation: The correctness of the algorithms has been verified by using a manual approach. The results of the qualitative evaluation (in comparison to the quantitative results given in Figure 9.7 and 9.8) are given in Figure 9.9 and 9.10. We observed that the number of (manually) identified change patterns reduced slightly. This is due to the multiple extraction of completely overlapped change patterns by the algorithms. Though the result list contained only a few completely overlapped change patterns, the overall results of the algorithms are correct. The identified change patterns were recorded in a pattern repository. A user can browse through, select and apply a change pattern from the list. We applied some of the stored change patterns in the domain ontology to confirm that they are valid and are useful to perform a frequent ontology change.

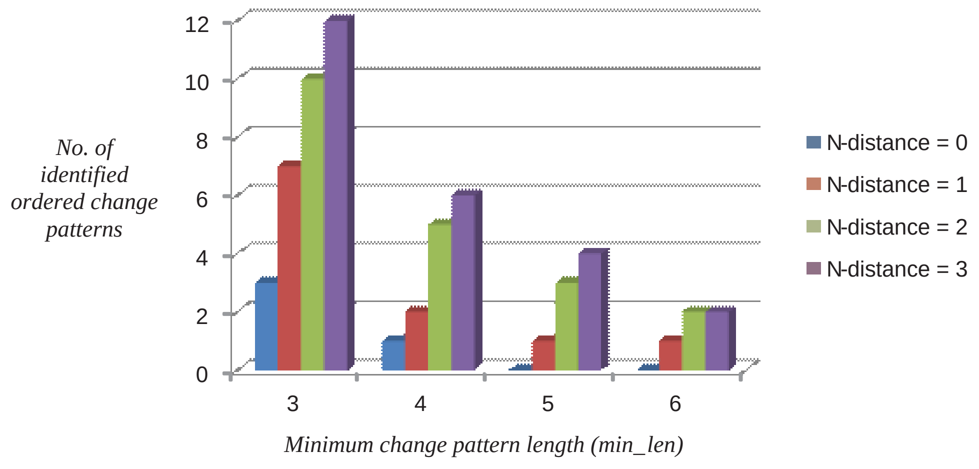


Figure 9.9: No. of identified ordered complete change patterns (Qualitative)

Limitations: The known limitation of the algorithms is that they cannot be applied to the change parameters which are represented as complex expressions. Our algorithms consider all parameters as atomic classes, properties or individuals. Secondly, our algorithms used an assumption, i.e. the target entity is always the first parameter of any

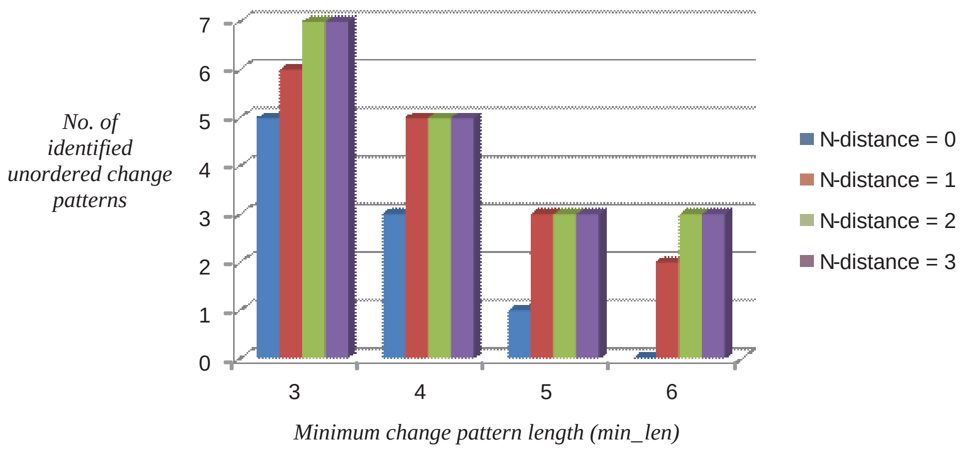


Figure 9.10: No. of identified unordered complete change patterns (Qualitative)

ontology change operations. This assumption does not suit, for example in the case of inverse properties, such as *hasSupervisor* and *isSupervisorOf*. In our future work, deep comparison of ontology change operations will be made in order to identify the target entity (context) in relation to identified change operations of a sequence.

9.5.4 Discussion

In this section, we evaluated the ordered complete (OCP) and unordered complete (UCP) change pattern discovery algorithms based on their effectiveness, efficiency and correctness. The UCP algorithm is effective in terms of the number of identified change patterns. This is due to the coverage of change sequences that are unordered (with respect to the reference change sequence) and are used to perform identical changes in the domain ontology. On the other hand, the OCP algorithm is efficient in terms of time consumption due to the permissibility of only positive node distances (x), i.e. the iteration process for the search of the next adjacent sequence node only operates in forward direction of the change log graph. However, in the case of UCP, for the search of the next adjacent sequence node, the algorithm also operates in a backward direction. This is due to the possibility of change operations in an unordered form compared to the referenced candidate change sequence. Another reason for the efficiency of OCP is the immediate termination of node search iterations once the next adjacent sequence node is not identified in the search space. However, in the case of UCP, if the next adjacent node is not identified, it is saved in the unidentified node list and the iteration moves forward to search for the next adjacent node until the whole change sequence ends. Unordered change operations make the UCP algorithm more complex in comparison to OCP as UCP needs to i) keep record of all change operations of the sequence (even if they are not identified), ii) recalculate the search space in each iteration, iii) search the next sequence node not only in the search space of the graph, but also in the unidentified list of change nodes and iv) convert a sequence to ascending form in each iteration.

9.6 Summary

In this chapter, we presented the results of the case study and lab experiments and evaluated our contribution. The main concern in evaluating the layered ontology change framework is its usefulness. Is it useful for different actors involved in ontology-driven content based systems and how effectively can the layered framework be used in a real world scenario? With regard to the higher level change pattern identification algorithms, the main concern is their effectiveness. We selected usability, functional suitability and performance efficiency as three key evaluation criteria. We performed empirical case studies in two domains in order to evaluate the practical suitability and usability of the proposed layered ontology change framework. Lab experiments were used to evaluate the effectiveness and performance of the ontology change pattern identification algorithms.

We selected university administration and database systems as domains for our case study. The university ontology represents an organization consisting of classes such as students, faculty, departments, research centers, courses etc. The database ontology represents a technical domain that can be looked at from different perspectives, such as concepts being covered in a course outline or a text book on the subject etc. The empirical results in both domains confirm that the layered framework is adequate and valid from a practical perspective. The higher level change operators, representing the ontology changes in the form of a domain-specific change pattern, are useful for the domain experts and the low level change operators are suitable for ontology engineers. Furthermore, logging the ontology changes at two different levels of granularity helps in two ways, i.e. first, recording the *fine-grained representation* of each ontology change in a lower level atomic change log and, second, *recording the intent* behind applied change operations in a higher level pattern change log. Capturing the intent of ontology changes at a higher level of abstraction leads to using more accurate evolution strategies for applied ontology change operations.

We compared the results of our composite change detection algorithms with the results of a manual detection approach. The comparison shows that an ontology engineer can detect the change patterns manually from a small scale ontology change log. However, in a real world scenario, the manual detection approach from a large scale ontology changes logs is not feasible and error-prone. An automated approach is a necessity here. On the one hand, the detected composite change patterns can be used to capture the intent behind the applied changes, on the other hand, the discovered domain-specific change patterns can be used as once-off change pattern specifications that can be instantiated in the future, whenever similar changes are to be applied.

Chapter 10

Conclusions

In development of tools and methods for ontology evolution, researchers initially focused on the fine-grained representation of ontology changes and capturing the differences between different versions of the domain ontologies. However, an explicit and semantic representation of an applied ontology change requires ontology changes to be captured at a higher level of abstraction. In this thesis, we focused on operationalisation and representation of ontology changes not only at the lower level, but also at a higher level (in the form of change patterns). We presented the layered change operator framework that allows users to perform ontology changes based on different levels of abstraction. We proposed a layered change log model that works in line with the change operator framework. Finally, change pattern identification algorithms are given that support the semantic representation of applied ontology changes by recording them in a higher level change log.

In Section 10.1, we describe a summary of the contribution based on our objectives, given solutions and the implementation. In Section 10.2, we discuss our approach. At the end, some future directions including an extension of our current work is given in Section 10.3.

10.1 Summary of contribution

Ontologies can support a variety of purposes, ranging from capturing the conceptual knowledge to the organisation of digital content and information. However, information systems are always subject to change and ontology change management can pose challenges. This thesis contributes a pattern-based ontology evolution framework focusing on ontology change operationalisation and representation phases of the ontology evolution life cycle. The contribution of this thesis can be summarized as,

- A layered ontology change operator framework based on the granularity, domain-specificity and abstraction of changes.

- A layered ontology change log model that captures the objective of ontology changes at a higher level of granularity and abstraction and supports a comprehensive understanding of ontology evolution.

- Graph-based algorithms for the detection of defined composite changes from the lower level change log that supports the identification of the intent behind any of the applied changes.

- Graph-based algorithms for discovery of recurring domain-specific change patterns that support in defining new usage-driven change patterns.

10.2 Discussion

Ontology-based content models help researchers to take a step forward from traditional digital content management systems to conceptual knowledge modelling to meet the requirements of the semantically aware content-based systems. In this regard, domain ontologies become essential for knowledge sharing activities, especially in areas such as bio-informatics, educational technology systems, indexing and retrieval, etc. We have been working with non-public domain ontologies used to annotate the content in large-scale information systems. As information systems will evolve with time, the underlying domain ontologies need to be synchronized.

The change in the domain ontology reflects the general changes in the information systems, flaws in an earlier conceptualization of information system, addition of new classes in the domain etc. The changes in the domain ontologies may include changes in the class hierarchy; some classes may get removed, modified, pulled up/down in the hierarchy etc. More description (in a form of object/data properties) can be added to the existing classes. In this thesis, we presented an ontology change management system, organized as a four-phase ontology evolution life cycle. The ontology change management system focuses on the ontology change operationalisation and on the representation and identification of ontology change patterns from ontology change logs.

To the best of our knowledge, currently there exist no ontology editing tool that provides ontology change operators based on different levels of granularity and abstraction. Ontology editing frameworks, such as Protégé, NeON, OBO-Edit etc., perform ontology changes at an atomic level. This restricts the usage of domain ontologies to specialized ontology engineers. We presented a layered change framework consisting of a layered change log model that works in line with the given layered change operator framework. While ontology engineers typically deal with generic changes at lower levels, other users can focus on domain-specific changes at higher levels. Such a layered change operator framework enables us to deal with structural and semantic changes at two separate levels without losing their interdependence. Additionally, it enables us to define a set of domain-specific changes which can be stored in a pattern catalogue, using a pattern template, as a consistent specification of domain-specific change patterns. The empirical study indicates that the solution is valid and efficiently adequate to handle ontology evolution. We found that a significant portion of ontology change and evolution is represented in our framework.

Identification of higher level change operations gives an ontology engineer clues about semantics / reasons behind any of the applied changes, based on the actual change activity data from a change log. We operationalized the identification of higher level changes using graph-based matching and pattern discovery approaches. Learning about semantics behind any of the applied change helped us in keeping the ontology consistent in a more appropriate manner. To do so, higher level evolutionary strategies are essential.

Constructing and storing the domain knowledge using a frame-based approach was introduced in the Protégé-Frames editor. It allows users to construct customizable domain-specific data entry forms and enter the instance-level data. As the class hierarchy as well as the description about any class will evolve over time, such data-entry forms will get obsolete unless customized through time. Discovery of the domain-specific change patterns from the change log can assist in this regard. It not only allows defining new "usage-driven" change patterns, but can also aid in customizing and editing of already existing "user-defined" data entry forms. As good patterns always arise from practical experience [36], such change patterns, created in a collaborative environment, provide guidelines to ontology change management and can be used in any change recommendation system.

We evaluated our contribution based on the empirical case studies and the experiments in a controlled environment. The change operators and patterns we found were based on actual changes being carried out by the users. The empirical results confirmed that the layered change framework is useful and suitable for ontology engineers and domain experts.

10.3 Future work

In this section, we discuss a few directions for the future work:

- Enhanced reusability of domain-specific change patterns through domain transfer

- Change pattern specification

- Identification of pattern-level causal dependencies

10.3.1 Enhanced reusability through domain transfer.

The benefit of a change pattern reuse is not only a saving in time, cost, and effort, but also an increase in "reliability" [Hemmann et al., 1993]. A highly reusable domain specific change pattern indicates that it is generally accepted within the domain. The reusability of the discovered domain-specific change patterns can be enhanced through domain transfer. During our empirical study, we observed similarities of patterns across domains which are similar to each other. For example, in the university domain, one can identify classes such as students, faculties and employees; a production company may have employees, customers, owners or shareholders. The change patterns provided at higher level can be applied to any subject domain ontology that is composed of a similar conceptual structure. The domain specific change patterns may require a small customization to meet the domain's own requirements. Similarity between two domain ontologies can be acknowledged by analyzing conceptual and syntactical structures within the domain ontologies. A number of algorithms are already developed to capture the similarities between classes of domain ontologies [Castano et al., 2003, Andrejko et al., 1990]. Some text engineering algorithms, e.g. Levenshtein's edit distance, may be used for textual comparisons of the named entities. An algorithm will be developed that distinguishes the similarities between two domains and validates how feasible it is to transfer it to other domain. During the refinement process, change operations can be added, deleted or modified. The sequence or parameters of change operations can also be altered to meet user needs.

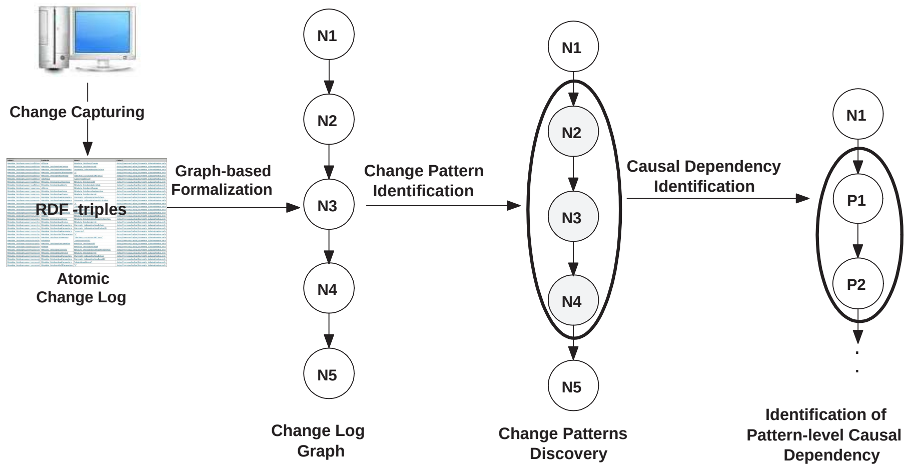
10.3.2 Change pattern specification.

Good documentation is vital for effective reuse of any framework. In this regard, our future work includes a specification of the (user-defined/usage-driven) domain-specific change patterns to support the notion of pattern-based ontology evolution. More specifically, we are interested in the once-off specification of the domain-specific change patterns that assist the ontology engineer to choose the appropriate change pattern in a given ontology evolution context. This can be achieved by utilizing a pattern template that enables a consistent change pattern specification for change patterns comprising of descriptive data (including pattern's name, its purpose, related change patterns etc.) and change data information (including definition along with pre/post conditions etc of involved change operations). In addition, change patterns available in the catalogue may also be classified based on the categorisation of available change operations in a domain-specific change pattern.

10.3.3 Pattern-level causal dependencies.

A causal dependency is related to the identification of ontological entities which frequently (if not always) evolve together. That means, change in one part of the domain ontology has a direct impact on another section of the ontology. In the future, we are interested in detecting pattern-level causal dependencies where one change pattern leads to the application of another change pattern. For example, whenever a new course is introduced in a university department, new subjects (and subject codes) are introduced simultaneously, course-related books have to be purchased and added to the library, new vacancies have been advertised in order to provide the expertise etc.

These causal dependencies are actually the association rules [Agrawal et al., 1995] that represent the relationships among the different discovered change patterns, defining a pattern language. These association rules can be used to capture causally dependent ontological entities. Such causal dependencies can be deployed in existing ontology



Change Capturing
RDF -triples
Atomic Change Log
Graph-based Formalization
N1
N2
N3
N4
N5
Change Log Graph
Change Pattern Identification
N1
N2
N3
N4
N5
Change Patterns Discovery
Causal Dependency Identification
N1
P1
P2
Identification of Pattern-level Causal Dependency

Figure 10.1: Identification of pattern-level causal dependency

change management systems in order to discover new trends within the domain, change request recommendations etc. This, in turn, can also serve as basis for process improvement actions, e.g. it may trigger patterns redesign or better control mechanisms [Guenther et al., 2006].

Bibliography

– Abgaz, Y.M., Javed, M., Pahl, C.: *Analyzing Impacts of Change Operations in Evolving Ontologies.* ISWC Workshops: Joint Workshop on Knowledge Evolution and Ontology Dynamics (EvoDyn), Boston, USA, (2012).

– Abgaz, Y.M., Javed, M., Pahl, C.: *Dependency Analysis in Ontology-driven Content-based Systems.* In: 12th International Conference on Artificial Intelligence and Soft Computing (ICAISC), Zakopane, Poland, (2012).

– Abgaz, Y.M., Javed, M., Pahl, C.: *A Framework for Change Impact Analysis of Ontology-driven Content-based Systems.* In: On the Move to Meaningful Internet Systems. OTM Workshops: 7th International IFIP Workshop on Semantic Web and Web Semantics (SWWS), Crete, Greece, (2011).

– Abgaz, Y.M., Javed, M., Pahl, C.: *Empirical Analysis of Impacts of Instance-driven Changes in Ontologies.* In: Proceedings of On the Move to Meaningful Internet Systems (OTM) Workshops: Volume 6428 of Lecture Notes in Computer Science, Springer-Berlin/Heidelberg, pages 368–377, (2010).

– Agrawal, R., Srikant, R.: *Fast Algorithms for Mining Association Rules.* In: International Conference of Very Large Data Bases (VLDB94), pages 487-499, Santiago, Chile, (1994).

– Agrawal, R., Srikant, R.: *Mining Sequential Patterns.* In: Proceedings of the 11th

International Conference on Data Engineering (ICDE). Philip S. Yu and Arbee L. P. Chen (Edition). IEEE Computer Society, Washington DC, USA, pages 3–14, (1995).

– Agosti, M., Di Nunzio, G.M.: *Web Log Mining: A Study of User Sessions.* In: 10th DELOS Thematic Workshop on Personalized Access, Profile Management, and Context Awareness in Digital Libraries (PersDL), Corfu, Greece, pages 70–74, (2007).

– Altschul, S., Gish, W., Miller, W., Myers, E., Lipman, D.: *Basic Local Alignment Search Tool.* In Journal of Molecular Biology, Volume 215(3), pages 403–410, (1990).

– Andrejko A., Bielikova M.: *Comparing Instances of the Ontological Concepts.* In: Proceedings of Second Workshop on Tools for Acquisition, Organisation and Presenting Info. and Knowledge, pages 26–35, (2007).

– Ashburner, M., Ball, C.A., Blake, J.A., Botstein, D., Butler, H., Cherry, J.M., Davis, A.P., Dolinski, K., Dwight, S.S., Eppig, J.T., Harris, M.A., Hill, D.P., Issel-Tarver, L., Kasarskis, A., Lewis, S., Matese, J.C., Richardson, J.E., Ringwald, M., Rubin, G.M., Sherlock, G.: *Gene ontology: Tool for the Unification of Biology.* Nature Genet. Volume 25, pages 25-29, (2000).

– Auer, S., Herre, H.: *A Versioning and Evolution Framework for RDF Knowledge Bases* In: Proceedings of the 6th International Andrei Ershov Memorial Conference on Perspectives of Systems Informatics, pages 55-69, Springer-Verlag, (2007).

– Banerjee, J., Kim, W., Kim, H.J., Korth, H.F.: *Semantics and Implementation of Schema Evolution in Object-Oriented Databases.* In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 311-322, (1987).

– Bandara, K.Y., Wang, M., Pahl, C.: *Context Modeling and Constraints Binding in Web Service Business Processes.* In: Proceedings of the First International Workshop on Context-Aware Software Technology and Applications (CASTA), Amsterdam, The Netherlands, (2009).

– Baumgartner, P., Furbach, U., Niemelá, I.: *Hyper Tableaux.* In: Orlowska, E., Alferes, J.J., Moniz Pereira, L. (eds.) JELIA 1996. Volume 1126 of Lecture Notes in Computer Science, Springer-Heidelberg, pages 1-17, (1996).

– Baggenstos, D.: *Implementation and Evaluation of Graph Isomorphism Algorithms for RDF-Graphs.* Diploma Thesis, University of Zurich, (2006).

– Bloehdorn, S., Petridis, K., Saathoff, C., Simou, N., Tzouaras, V., Avrithis, Y., Handschuh, S., Kompatsiaris, Y., Staab, S., Strintzis, M.G.: *Semantic Annotation of Images and Videos for Multimedia Analysis.* In: Proceedings of the 2nd European Semantic Web Conference (ESWC), Heraklion, Greece, (2005).

– Bechhofer, S., Yesilada, Y., Horan, B., Goble, C.: *Knowledge-driven Hyperlinks: Linking in the Wild.* In: 4th International Conference on Adaptive Hypermedia and Adaptive Web-Based Systems (AH), Volume 4018 of Lecture Notes in Computer Science, Springer, pages 1–10, (2006).

– Burleson, C.: *Introduction to the Semantic Web Vision and Technologies - Part 3 - The Resource Description Framework.* A Semantic focus blog: Available at `http://www.semanticfocus.com/blog`

– Boyce, S., Pahl, C.: *The Development of Subject Domain Ontologies for Educational Technology Systems.* In Journal of Educational Tech. & Society, Volume 10(3), pages 275–288, (2007).

– Boyer, R.S., Moore, J.S.: *A Fast String Searching Algorithm.* Communications of the ACM, Volume 20(10), pages 762–772, (1977).

– Castano, S., Ferrara, A. Montanelli, S.: *H-MATCH: An Algorithm for Dynamically Matching Ontologies in Peer-based System.* In: Proceedings of the 1st International Workshop on Semantic Web and Databases (SWDB), (2003).

– Carr, L., Bechhofer, S., Goble, C., Hall, W.: *Conceptual Linking: Ontology-based Open Hypermedia.* In: 10th International World Wide Web Conference, Hong Kong, pages 334–342, (2001).

– Cook, J.E., Wolf, A.L.: *Discovering Models of Software Processes from Event-based Data.* In: ACM Transactions on Software Engineering and Methodology. Volume 5(3), pages 215–249, (1998).

– Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., Loew, M.: *Algebraic Approaches to Graph Transformation, Part-I: Basic Concepts and Double Pushout Approach.* Technical Report Tr-96-17, Universita di Pisa, Dipartimento di Informatica, (1996).

– Daconta, M.C., Obrst, L.J., Smith, K.T.: *The Semantic Web: A guide to the future of XML, Web Services and Knowledge Management.* Wiley Computer Publishing, ISBN 0-471-43257-1, (2003).

– De Leenheer, P., Mens, T.: *Using Graph Transformation to Support Collaborative Ontology Evolution* In: A. Schurr, M. Nagl, A. Zundorf (Eds.), Proceedings of Agtive (Kassel, Germany), Volume 5088 of Lecture Notes in Computer Science, Springer, pages 44-58, (2007).

– Dill, S., Eiron, N., Gibson, D., Gruhl, D., Guha, R., Jhingran, A., Kanungo, T., Rajagopalan, S., Tomkins, A., Tomlin, J.A., Zien, J.Y.: *SemTag and Seeker: Bootstrapping the Semantic Web via Automated Semantic Annotation.* In: Proceedings of the 12th International Conference on World Wide Web (WWW'03), Budapest, Hungary, ACM Press, pages 178–186, (2003).

– Djedidi, R., Aufaure, M-A.: *Ontology Evolution: State of the Art and Future Directions.* In Book: Ontology Theory, Management and Design: Advanced Tools and

Models, F. Gargouri and W. Jaziri (Eds.), Section III. Chapter 8, Information Science Reference Publisher, (2010).

– D'Aquin, M., Doran, P., Motta, E., Tamma, V.A.M.: *Towards a Parametric Ontology Modularization Framework based on Graph Transformation.* In: B.C. Grau, V. Honavar, A. Schlicht, F. Wolter (Eds.), WoMO, CEUR Workshop Proceedings, Volume 315, (2007).

– Drewes, F., Hoffmann, B., Plump, D.: *Hierarchical Graph Transformation.* In: Journal of Computer System Science, Volume 64(2), pages 249-283, (2002).

– Easterbrook, S., Singer, J., Storey, M.A., Damian, D.: *Selecting Empirical Methods for Software Engineering Research.* Guide to Advanced Empirical Software Engineering, pages 285–311, (2008).

– Ehrig, H., Prange, U., Taentzer, G.: *Fundamental Theory for Typed Attributed Graph Transformation.* In: Proceedings of the International Conference on Graph Transformation, pages 161–177, (2004).

– Ehrig, H., Pfender, M., Schneider, H.J.: *Graph Grammars: An Algebraic Approach.* In: 14th Annual IEEE Symposium on Switching and Automata Theory, pages 167-180, (1973).

– Elmasri, R., Navathe, S.M.: *Fundamentals of Database Systems.* Fifth Edition - Addison Wesley, (2007).

– Espinoza, M., Gómez-Pérez, A., Mena, E.: *LabelTranslator A Tool to Automatically Localize an Ontology..* In: Bechhofer, S., Hauswirth, M., Hoffmann, J., Koubarakis, M. (eds.) ESWC 2008. Volume 5021 of Lecture Notes in Computer Science, Springer-Heidelberg, pages 792-796, (2008).

– Falconer, S., Tudorache, T., Noy, N.F.: *An Analysis of Collaborative Patterns in Large-scale Ontology Development Projects.* In: Proceedings of the 6th International Conference on Knowledge Capture (K-CAP '11), pages 25–32, (2011).

– Filipowska, A., Kaczmarek, M., Markovic, I.: *Organisational Ontology Framework for Semantic Business Process Management.* In: Proceedings of the 12th International Business Information Systems Conference (BIS 2009). Volume 21 of Lecture Notes on Business Information Processing (LNBIP), Springer, pages 1–12, (2009).

– Flouris, G., Plexousakis, D., Antoniou, G.: *A Classification of Ontology Change.* In SWAP: Poster Proceedings of the 3rd Italian Semantic Web Workshop, Semantic Web Applications and Perspectives, (2006).

– Flury, T., Privat, G., Ramparany, F.: *OWL-based Location Ontology for Context-aware Services.* In: Artificial Intelligence in Mobile Systems, Nottingham, UK, (2004).

– Gacitua-Decar, V., Pahl, C.: *Ontology-based Patterns for the Integration of Business Processes and Enterprise Application Architectures.* In: G. Mentzas et al. (Eds), Semantic Enterprise Application Integration for Business Processes: Service-Oriented Frameworks, IGI Pub. (2009).

– Gómez-Pérez, A., Fernandez-López, M., Corcho, O.: *Ontological Engineering: With Examples from the Areas of Knowledge Management, E-commerce and the Semantic Web.* Springer - Business and Economics, (2006).

– Groner, G., Staab, S.: *Categorization and Recognition of Ontology Refactoring Pattern.* Arbeitsberichte aus dem Fachbereich Informatik, Number 09/2010, Institut WeST, University of Koblenz-Landau, (2010).

– Gruber, T.R.: *A Translation Approach to Portable Ontology Specifications.* Knowledge Acquisition, Volume 5(2), pages 199-220, (1993).

– Gruhn, V., Pahl, C., Wever, M.: *Data Model Evolution as Basis of Business Process Management.* In: Proceedings of the 14th International Conference on Object-Oriented and Entity-Relationship Modelling (OOER '95), Springer, pages 270–281, (1995).

– Guenther, C., Rinderle, S., Reichert, M., Van der Aalst, W.: *Change Mining in Adaptive Process Management Systems.* In: Proceedings of the 14th International Conference on Cooperative Information Systems, Volume 4275 of Lecture Notes in Computer Science, Springer, pages 309–326, (2006).

– Guo, Y., Pan, Z., Heflin, J.: *LUBM: A benchmark for OWL knowledge base systems.* In: Journal of Web Semantics, Volume 3, Issue 2-3, pages 158-182, (2005).

– Haarslev, V., Hidde, K., Móller, R., Wessel, M.: *The RacerPro Knowledge Representation and Reasoning System.* In Semantic Web Journal, Volume 2, pages 1–11, (2011).

– Haase, P., Lewen, H., Studer, R., Erdmann, M., d'Aquin, M., Motta, E.: *The NeOn Ontology Engineering Toolkit.* In: Demo Session of Developers Track at WWW' 2008, Beijing, China, (2008).

– Haase, P., Sure, Y.: *Usage Tracking for Ontology Evolution.* EU IST Project SEKT Deliverable D3.2.1, Work Package 3.2, (2003).

– Handschuh, S., Staab, S., Studer, R.: *Leveraging Metadata Creation for the Semantic Web with CREAM.* In: KI 2003–Advances in Artificial Intelligence. Volume 2821 of Lecture Notes in Computer Science, Springer Berlin/Heidelberg, pages 19–33, (2003).

– Hemmann, T., Voss, H.: *A Reusable and Specializable Interpretation Model for Model-Based Diagnosis.* In: 3rd KADS Meeting Siemens AG. Munich, pages 189–205, (1993).

– He, D., Goker, A.: *Detecting Session Boundaries from Web User Logs.* In: Proceedings of the 22nd Annual Colloquium on Information Retrieval Research, Cambridge, UK. British Computer Society, pages 57–66, (2000).

– Heckel, R., Kster, J.M., Taentzer, G.: *Confluence of Typed Attributed Graph Transformation Systems.* In: Proceedings of the 1st International Conference on Graph Transformation (ICGT), Volume 2505 of Lecture Notes in Computer Science, Springer, pages 161–176, (2002).

– Hesse, W.: *Engineers Discovering the "Real World" From Model-Driven to Ontology-Based Software Engineering.* In: Proceedings of the 2nd International United Information Systems Conference (UNISCON), Volume 5 of Lecture Notes in Business Information Processing. Springer, pages 136–147, (2008).

– Hirate, Yu., Yamana, H.: *Sequential Pattern Mining with Time Intervals.* In: Volume 3918 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, pages 775–779, (2006).

– Holohan, E., McMullen, D., Melia, M., Pahl, C.: *Adaptive E-Learning Content Generation based on Semantic Web Technologies.* In: Proceeding of the International Workshop on Applications of Semantic Web Technologies for E-Learning (SW-EL'2005) at the 12th International Conference on Artificial Intelligence in Education (AIED), IOS Press, (2005).

– Horridge, M.: *OWL Syntaxes.* (2010), `http://ontogenesis.knowledgeblog.org/88`.

– Huan, J.: *Graph Based Pattern Discovery in Protein Structures.* PhD Thesis, Department of Computer Science, University of North Carolina, (2006).

– Inokuchi, A., Washio, T., Motoda, H.: *An Apriori-based Algorithm for Mining Frequent Substructures from Graph Data.* In: Proceedings of the 4th European Conference on Principles of Data Mining and Knowledge Discovery, pages 13–23, (2000).

– Ivancsy, I., Vajk, I.: *Frequent Pattern Mining in Web Log Data.* Acta Polytechnica Hungarica. Journal of Applied Sciences, Volume 3(1), pages 77–90, (2006).

– Javed, M., Abgaz, Y.M., Pahl, C.: *Composite Ontology Change Operators and their Customizable Evolution Strategies.* ISWC Workshops: Joint Workshop on Knowledge Evolution and Ontology Dynamics (EvoDyn), Boston, USA, (2012).

– Javed, M., Abgaz, Y.M., Pahl, C.: *Towards Implicit Knowledge Discovery from Ontology Change Log Data.* In: 5th International Conference on Knowledge Science, Engineering and Management (KSEM), Volume 7091 of Lecture Notes in Computer Science, Springer-Verlag, pages 136–147, (2011).

– Javed, M., Abgaz, Y.M., Pahl, C.: *Graph-based Discovery of Ontology Change Patterns.* ISWC Workshops: Joint Workshop on Knowledge Evolution and Ontology Dynamics (EvoDyn), Bonn, Germany, (2011).

– Javed, M., Abgaz, Y.M., Pahl, C.: *A Layered Framework for Pattern-based Ontology Evolution.* In: 3rd International Workshop on Ontology-Driven Information System Engineering (ODISE), London, UK, (2011).

– Javed, M., Abgaz, Y.M., Pahl, C.: *Ontology-based Domain Modelling for Consistent Content Change Management.* In: International Conference on Ontological and Semantic Engineering, Venice, Italy, (2010).

– Javed, M., Abgaz, Y.M., Pahl, C.: *A Pattern-based Framework of Change Operators for Ontology Evolution.* In: On the Move to Meaningful Internet Systems: OTM Workshops. Volume 5872 of Lecture Notes in Computer Science, Springer, pages 544-553, Algarve, Portugal, (2009).

– Jiang, D., Pei, J., Li, H.: *Web Search/Browse Log Mining: Challenges, Methods, and Applications.* In: Proceedings of the 19th International Conference on World Wide Web, pages 1351–1352, (2010).

– Kiryakov, A., Popov, B., Terziev, I., Manov, D., Ognyanoff, D.: *Semantic Annotation, Indexing, and Retrieval.* In: Journal of Semantic Web, Volume 2, pages 49–79, (2004).

– Kahan, J., Koivunen, M.J., Prud'Hommeaux, E., Swick, R.R.: *Annotea: An open RDF Infrastructure for Shared Web Annotations.* In: Proceedings of the 10th International World Wide Web Conference (WWW 2001), Hong Kong, (2001).

– Klein, M.: *Change Management for Distributed Ontologies.* PhD Thesis, Vrije University Amsterdam, (2004).

– Klein, M. and Noy, N.F.: *A Component-Based Framework for Ontology Evolution.* In: Proceedings of the IJCAI-03 Workshop on Ontologies and and Distribution Systems, Volume 71, (2003).

– Kosala, R., Blockeel, H.: *Web Mining Research: A Survey*: Newsletter of the Special Interest Group (SIG) on Knowledge Discovery and Data Mining, ACM, Volume 2(1), pages 1–15, (2000).

– Kuramochi, M., Karypis, G.: *Frequent Subgraph Discovery.* In: 1st IEEE Conference on Data Mining (ICDM '01), pages 313–320, (2001).

– Li, C., Wang, J.: *Efficiently Mining Closed Subsequences with Gap Constraints.* In: Proceedings of the SIAM International Conference on Data Mining (SDM), pages 13–322, (2008).

– Liang, Y., Alani, H., Shadbolt, N.: *Ontology Change Management in Protégé.* In: Proceedings of AKT DTA Colloquium, Milton Keynes, UK, (2005).

– Maedche, A., Motik, B., Stojanovic, L., Studer, R., Volz, R.: *Managing Multiple Ontologies and Ontology Evolution in Ontologging.* In: Proceedings of the Intelligent Information Processing, Montreal, Canada, pages 51–68, (2002).

– Montgomery, A.L., Faloutsos, C.: *Identifying Web Browsing Trends and Patterns.* In: Proceedings of the IEEE Journal Computer, Volume 34, Issue 7, pages 94-95, (2001).

– Mitra, P., Wiederhold, G., Kersten, M.: *A Graph-oriented Model for Articulation of Ontology Interdependencies.* In: Proceedings of the Conference on Extending Database Technology (EDBT' 2000), Konstanz, Germany, (2000).

– Noy, N.F., Klein, M.: *Ontology evolution: Not the Same as Schema Evolution.* In: Knowledge and Information Systems, Volume 6(4), pages 428–440, July, (2004).

– Oliver, D. E., Shahar, Y., Shortliffe, E. H., Musen, M. A.: *Representation of Change in Controlled Medical Terminologies.* In: Journal of Artificial Intelligence in Medicine, Volume 15(1), pages 53–76, (1999).

– Pabarskaite, Z., Raudys, A.: *A Process of Knowledge Discovery from Web Log Data: Systematization and Critical Review.* In: Journal of Intelligent Information Systems, Volume 28(1), pages 79–104, (2007).

– Pahl, C., Javed, M., Abgaz, Y.M.: *Utilising Ontology-based Modelling for Learning Content Management.* In: Proceedings of ED-MEDIA 2010-World Conference on Educational Multimedia, Hypermedia & Telecommunications, Toronto, Canada, (2010).

– Palma, R., Haase, P., Corcho, O., Gomez-Perez, A.: *Change Representation For OWL 2 Ontologies.* In: Proceedings of the 6th International Workshop on OWL: Experiences and Directions (OWLED), (2009).

– Papavassiliou, V. ,Flouris, G.,Fundulaki, I., Kotzinos, D., Christophides, V.: *On Detecting High-Level Changes in RDF/S KBs.* In: Proceedings of the 8th International Semantic Web Conference, Volume 5823 of Lecture Notes in Computer Science, Springer, pages 473–488, (2009).

– Patil, A.A., Oundhakar, S., Sheth, A, Verma, K.: *Meteor-S: Web Service Annotation Framework.* In: WWW '04 - Proceedings of the 13th International Conference on World Wide Web, pages 553-562, ACM Press, (2004).

– Pedrinaci, C., Domingue, J.: *Towards an Ontology for Process Monitoring and Mining.* In: Proceedings of the Workshop on Semantic Business Process and Product Life Cycle Management, (2007).

– Peng, W., Li, T., Ma, S.: *Mining Logs Files for Data-driven System Management.* In: Journal of SIGKDD Explorations, Volume 7(1), pages 44–51, (2005).

– Petridis, K., Bloehdorn, S., Saathoff, C., Simou, N., Dasiopoulou, S., Tzouvaras, V., Handschuh, S., Avrithis, Y., Kompatsiaris, I., Staab, S.: *Knowledge Representation and Semantic Annotation of Multimedia Content.* In: IEEE Proceedings Vision, Image and Signal Processing, Special issue on Knowledge-Based Digital Media Processing. Volume 153(3), pages 255–262, (2006).

– Pinto, H., Han, J., Pei, J., Wang, K., Chen, Q., Dayal, U.: *Multi-dimensional Sequential Pattern Mining.* In: ACM International Conference on Information and Knowledge Management (CIKM '01), pages 81–88, (2001).

– Pitkow, J., Margaret, R.: *Integrating Bottom-up and Top-down Analysis for Intelligent Hypertext.* In: Intelligent Knowledge Management, Intelligent Hypertext Workshop, National Institute of Standard Technology, (1994).

– Plantevit, M., Laurent, A., Laurent, D., Teisseire, M., Choong, Y.W.: *Mining Multi-dimensional and Multilevel Sequential Patterns.* In: Proceedings of the ACM Transactions on Knowledge Discovery from Data, Article 4, Volume 4(1), (2010).

– Plessers, P., De Troyer, O., Casteleyn, S.: *Understanding Ontology Evolution: A Change Detection Approach.* In Web Semantics: Science, Services and Agents on the World Wide Web, Volume 5(1), pages 39–49, (2007).

– Plessers, P., De Troyer, O.: *Ontology Change Detection Using a Version Log.* In: Proceedings of the 4th International Semantic Web Conference, Springer, pages 578–592, (2005).

– Qin, L., Atluri, V.: *Evaluating the Validity of Data Instances Against Ontology Evolution Over the Semantic Web.* In: Journal Information and Software Technology. Volume 51(1), pages 83–97, (2009).

– Quint, V., Vatton, I.: *An Introduction to Amaya.* W3C NOTE, (1997).

– Reeve, L., Han, H.: *Semantic Annotation for Semantic Social Networks Using Community Resources.* In: Journal AIS SIGSEMIS Bulletin. Volume 2, No. 3-4, pages 52–56, (2005).

– Rieß, C., Heino, N., Tramp, S., Auer, S.: *EvoPat - Pattern-based Evolution and Refactoring of RDF knowledge Bases.* In: Proceedings of the 9th International Semantic Web Conference on The semantic Web - Volume Part I, ISWC10, pages 647-662. Springer-Verlag, (2010).

– Ritcher, J.D.: *The OBO Flat File Format Specification - Version 1.2.* `http://www.geneontology.org/GO.format.obo-1\_2.shtml`, (2006).

– Rudolf, M.: *Utilizing Constraint Satisfaction Techniques for Efficient Graph Pattern Matching.* In: Proceeding of 6th International Workshop on Theory and Application of Graph Transformation, Volume 1764 of Lecture Notes in Computer Science, Springer-Verlag, pages 381–394, (2000).

– Schmidt-Schauss, M., Smolka, G.: *Attributive Concept Descriptions with Complements.*. In Artificial Intelligence, Volume 48(1), pages 1-26, (1991).

– Schmidt, D., Fayad, M., Johnson, R.: *Software Patterns.* In: Communications of the ACM, Special Issue on Patterns and Pattern Lang. Volume 39(10), pages 37-39, (1996).

– Schroeter, R., Hunter, J., Kosovic, D.: *Vannotea–A Collaborative Video Indexing, Annotation and Discussion System for Broadband Networks.* In: Proceedings of the K-CAP 2003 Workshop on Knowledge Markup and Semantic Annotation, Florida, USA, (2003).

– Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: *Pellet: A Practical OWL-DL Reasoner.* In: Journal of Web Semantics, Volume 5(2), pages 51-53, (2007).

– Shaban-Nejad, A., Haarslev, V.: *An Enhanced Graph-oriented Approach for Change Management in Distributed Biomedical Ontologies and Linked Data.* In: Proceedings of the IEEE International Conference on Bioinformatics and Biomedicine Workshops (BIBMW), pages 615–622, (2011).

– Shearer, R., Motik, B., Horrocks, I.: *HermiT: A Highly Efficient OWL Reasoner.* In: OWLED 2008, Volume 432 of CEUR Workshop Proceedings, (2008).

– Spiliopoulos, V., Vouros, G.A., Karkaletsis, V.: On the Discovery of Subsumption Relations for the Alignment of Ontologies. In: Journal of Web Semantics, Volume 8(1), pages 69–88, (2010).

– Srikant, R., Agrawal, R.: *Mining Sequential Patterns: Generalizations and Performance Improvements.* In: Proceedings of the International Conference on Extending Data Base Technology. Volume 1057 of Lecture Notes in Computer Science, Springer Verlag, pages 3–17, (1996).

– Stefanowski, J.: *Algorithms for Context Based Sequential Pattern Mining.* In: Fundamenta Informaticae, Volume 76(4), pages 495–510, (2007).

– Stojanovic, L.: *Methods and Tools for Ontology Evolution.* PhD Thesis, University of Karlsruhe, (2004).

– Stojanovic, L., Maedche, A., Stojanovic, N., Studer, R.: *Ontology Evolution as Reconfiguration-design Problem Solving.* In: Proceedings of the 2nd International Conference on Knowledge Capture (KCAP), (2003).

– Stojanovic, L., Maedche, A., Motik, B., Stojanovic, N.: *User-driven Ontology Evolution Management.* In: Proceedings of the 13th International Conference on Knowledge Engineering and Knowledge Management (EKAW). Volume 2473 of Lecture Notes in Computer Science, Springer, pages 285–300, (2002).

– Suryn, W., Abran, A.: *ISO/IEC SQuaRE. The Second Generation of Standards for Software Product Quality.* In: 12th International Conference on Software Engineering and Applications (SEA), Marina del Rey, California, USA (2003).

– Tao, F., Murtagh, F.: *Towards Knowledge Discovery From WWW Log Data.* In: IEEE International Conference on Information Technology: Coding and Computing, pages 302–307, (2000).

– Tallis, M., Gil, Y.: *Designing Scripts to Guide Users in Modifying Knowledge-based Systems.* In: Proceedings of the 16th National Conference on Artificial Intelligence and 11th Conference on Innovative Applications of Artificial Intelligence, pages 242–249, Florida, USA, (1999).

– Trinkunas, J., Vasilecas, O.: *A Graph Oriented Model for Ontology Transformation into Conceptual Data Model.* In: Journal of Information Technology and Control, Volume 36, pages 126–132, (2007).

– Tsarkov, D., Horrocks, I.: *FaCT++ Description Logic Reasoner: System Description.* In: Proceedings of the International Joint Conference on Automated Reasoning (IJCAR), Springer, pages 292–297, (2006).

– Tury, M., Bielikova, M.: *An Approach to Detect Ontology Changes.* In: Proceedings of the 6th International Conference on Web Engineering (ICWE), Palo Alto, California, ACM Press, (2006).

– Uren, V., Cimiano, P., Iria, J., Handschuh, S., Vargas-Vera, M., Motta, E., Ciravegna, F.: *Semantic Annotation for Knowledge Management: Requirements and a Survey of the State of the Art.* In: Journal of Web Semantics. Volume 4, Issue 1, pages 14–28, (2006).

– Van der Aalst, W.M.P.: *Matching Observed Behaviour and Modelled Behaviour. An Approach Based on Petri Nets and Integer Programming.* In: Journal on Decision Support Systems, Volume 42(3), pages 1843-1859, (2006).

– Vargas-Vera, M., Motta, E., Domingue, J., Lanzoni, M., Stutt, A., Ciravegna, F.: *MnM: Ontology Driven Semi-Automatic and Automatic Support for Semantic Markup.* In: 13th International Conference on Knowledge Engineering and Management (EKAW 2002), ed Gomez-Perez, A., Volume 2473 of Lecture Notes on Artificial Intelligence, Springer Berlin/Heidelberg, pages 379–391, (2002).

– Valiente, G., Martínez, C.: *An Algorithm for Graph Pattern-Matching.* In: Proceedings of the 4th South American Workshop on String Processing. Volume 8 of Int. Informatics Series, pages 180–197, (1997).

– Wach, E. P.: *Automated Ontology Evolution for an E-Commerce Recommender.* In: Proceedings of the 14th International Conference on Business Information Systems (BIS), Volume 97 of Lecture Notes in Business Information Processing, Springer Berlin/Heidelberg, pages 166–177, (2011).

– Wen, L., Wang, J., Van der Aalst, W.M.P., Huang, B., Sun, J.: *Mining Process Models with Prime Invisible Tasks.* In: Journal of Data Knowledge Engineering, Volume 69, No. 10, pages 999–1021, (2010).

– W3C 2004: *OWL Web Ontology Language - Overview.* W3C Recommendation, World Wide Web Consortium, (2004), `http://www.w3.org/TR/owl-features/`.

– W3C 2009: *OWL 2 Web Ontology Language - Structural Specification and Functional-Style Syntax.* W3C Recommendation, World Wide Web Consortium, (2009), `http://www.w3.org/TR/owl2-syntax/`.

– W3C 2004: *OWL Web Ontology Language - Reference.* W3C Recommendation, World Wide Web Consortium,(2004), `http://www.w3.org/TR/owl-ref/\#Sublanguages`.

– Yan, X., Han, J.: *gSpan: Graph-based Substructure Pattern Mining.* In: IEEE International Conference on Data Mining, pages 721–724, (2002).

– Yu, L.: *Mining Change Logs and Release Notes to Understand Software Maintenance and Evolution.* In: CLEI Electron Journal, Volume 12, No. 2, pages 1–10, (2009).

– Zablith, F.: *Dynamic Ontology Evolution.* In: International Semantic Web Conference (ISWC) Doctoral Consortium, Karlsruhe, Germany, (2008).

– Zhang, M., Kao, B., Cheung, D.W., Yip, K.Y.: *Mining Periodic Patterns with Gap Requirement from Sequences.* In: Proceedings of the ACM Transactions on Knowledge Discovery from Data (TKDD), Article 2, Volume 1(2), (2007).

– Zhu, X., Wu, X.: *Mining Complex Patterns Across Sequences with Gap Requirements.* In: Proceedings of the 20th International Joint Conference on Artificial Intelligence, pages 2934–2940, (2007).

– Zhao, Q., Bhowmick, S.S.: *Sequential Pattern Mining: A Survey.* In: Technical Report, CAIS, Nanyang Technological University, Singapore, No. 2003118, (2003).

Appendix A

Metadata Ontology

In this appendix, we give the metadata ontology change model (discussed in Section 5.2) that has been implemented in form of an ontology using OWL language. The entities of the metadata ontology are being used to construct the RDF triples - representing the ontology changes at different level of granularity.

```
<?xml version="1.0"?>

<!DOCTYPE rdf:RDF [
    <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
    <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
    <!ENTITY MO "http://www.cngl.ie/ontology/MO.owl#" >
    <!ENTITY owl2xml "http://www.w3.org/2006/12/owl2-xml#" >
    <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
    <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
]>

<rdf:RDF xmlns="http://www.cngl.ie/ontology/MO.owl#"
     xml:base="http://www.cngl.ie/ontology/MO.owl"
     xmlns:owl2xml="http://www.w3.org/2006/12/owl2-xml#"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:MO="http://www.cngl.ie/ontology/MO.owl#"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:owl="http://www.w3.org/2002/07/owl#">
    <owl:Ontology rdf:about=""/>

    <!--
    ///////////////////////////////////////////////////////////////
    //
    // Object Properties
    //
    ///////////////////////////////////////////////////////////////
     -->

    <owl:ObjectProperty rdf:about="#docRef">
        <rdfs:domain rdf:resource="#Trace"/>
        <rdfs:subPropertyOf rdf:resource="&owl;topObjectProperty"/>
    </owl:ObjectProperty>

    <owl:ObjectProperty rdf:about="#hasAuxParam1">
        <rdfs:domain rdf:resource="#Change"/>
        <rdfs:range rdf:resource="#Entity"/>
        <rdfs:subPropertyOf rdf:resource="&owl;topObjectProperty"/>
    </owl:ObjectProperty>
```

```
<owl:ObjectProperty rdf:about="#hasAuxParam2">
    <rdfs:domain rdf:resource="#Change"/>
    <rdfs:subPropertyOf rdf:resource="&owl;topObjectProperty"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasAxiom">
    <rdfs:range rdf:resource="#Axiom"/>
    <rdfs:subPropertyOf rdf:resource="&owl;topObjectProperty"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasClassAxiom">
    <rdfs:subPropertyOf rdf:resource="&owl;topObjectProperty"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasCreator">
    <rdfs:domain rdf:resource="#Change"/>
    <rdfs:range rdf:resource="#User"/>
    <rdfs:subPropertyOf rdf:resource="&owl;topObjectProperty"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasDataPropertyAxiom">
    <rdfs:subPropertyOf rdf:resource="&owl;topObjectProperty"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasEntity">
    <rdfs:domain rdf:resource="#Change"/>
    <rdfs:range rdf:resource="#Entity"/>
    <rdfs:subPropertyOf rdf:resource="&owl;topObjectProperty"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasIndividualAxiom">
    <rdfs:subPropertyOf rdf:resource="&owl;topObjectProperty"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasObjectPropertyAxiom">
    <rdfs:subPropertyOf rdf:resource="&owl;topObjectProperty"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasOperation">
    <rdfs:subPropertyOf rdf:resource="&owl;topObjectProperty"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasRestriction">
    <rdfs:range rdf:resource="#Restriction"/>
    <rdfs:subPropertyOf rdf:resource="&owl;topObjectProperty"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasTargetParam">
    <rdf:type rdf:resource="&owl;InverseFunctionalProperty"/>
    <rdfs:domain rdf:resource="#Change"/>
    <rdfs:range rdf:resource="#Entity"/>
    <rdfs:subPropertyOf rdf:resource="&owl;topObjectProperty"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#ontRef">
    <rdfs:domain rdf:resource="#Trace"/>
    <rdfs:subPropertyOf rdf:resource="&owl;topObjectProperty"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#patternParticipants">
    <rdfs:range rdf:resource="#Entity"/>
    <rdfs:domain rdf:resource="#PatternChange"/>
    <rdfs:subPropertyOf rdf:resource="&owl;topObjectProperty"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#relatedPattern">
    <rdfs:domain rdf:resource="#PatternChange"/>
    <rdfs:range rdf:resource="#PatternChange"/>
    <rdfs:subPropertyOf rdf:resource="&owl;topObjectProperty"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="&owl;topObjectProperty"/>

<!--
///////////////////////////////////////////////////////////////
//
// Data properties
//
///////////////////////////////////////////////////////////////
 -->

<!-- http://www.cngl.ie/ontology/MO.owl#changeId -->
```

```
<owl:DatatypeProperty rdf:about="#changeId">
    <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="#hasDescription">
    <rdfs:domain rdf:resource="#Change"/>
    <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="#noOfOperations">
    <rdfs:domain rdf:resource="#ChangeOperation"/>
    <rdfs:range rdf:resource="&xsd;integer"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="#noOfParameters">
    <rdfs:domain rdf:resource="#Change"/>
    <rdfs:range rdf:resource="&xsd;integer"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="#patternLabel">
    <rdfs:domain rdf:resource="#PatternChange"/>
    <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="#patternPurpose">
    <rdfs:domain rdf:resource="#PatternChange"/>
    <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="#sessionId">
    <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="#timestamp">
    <rdfs:domain rdf:resource="#Change"/>
    <rdfs:range rdf:resource="&xsd;dateTime"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="#traceId"/>

<!--
///////////////////////////////////////////////////////////////////
//
// Classes
//
///////////////////////////////////////////////////////////////////
 -->

<!-- http://www.cngl.ie/ontology/MO.owl#AddConceptGeneralization -->

<owl:Class rdf:about="#AddConceptGeneralization">
    <rdfs:subClassOf rdf:resource="#CompositeChange"/>
</owl:Class>

<owl:Class rdf:about="#AddConceptSpecialization">
    <rdfs:subClassOf rdf:resource="#CompositeChange"/>
</owl:Class>

<owl:Class rdf:about="#AddInteriorConcept">
    <rdfs:subClassOf rdf:resource="#CompositeChange"/>
</owl:Class>

<owl:Class rdf:about="#Administrator">
    <rdfs:subClassOf rdf:resource="#User"/>
</owl:Class>

<owl:Class rdf:about="#AtomicChange">
    <rdfs:subClassOf rdf:resource="#ChangeOperation"/>
</owl:Class>

<owl:Class rdf:about="#AuxParam1">
    <rdfs:subClassOf rdf:resource="#Parameter"/>
</owl:Class>

<owl:Class rdf:about="#AuxParam2">
    <rdfs:subClassOf rdf:resource="#Parameter"/>
</owl:Class>

<owl:Class rdf:about="#Axiom">
    <rdfs:subClassOf rdf:resource="#OntologyElements"/>
</owl:Class>

<owl:Class rdf:about="#Cardinality">
```

```
    <rdfs:subClassOf rdf:resource="#Restriction"/>
</owl:Class>

<owl:Class rdf:about="#Change"/>

<owl:Class rdf:about="#ChangeCondition">
    <rdfs:subClassOf rdf:resource="&owl;Thing"/>
</owl:Class>

<owl:Class rdf:about="#ChangeOperation"/>

<owl:Class rdf:about="#ClassAxioms">
    <rdfs:subClassOf rdf:resource="#Axiom"/>
</owl:Class>

<owl:Class rdf:about="#CompositeChange">
    <rdfs:subClassOf rdf:resource="#ChangeOperation"/>
</owl:Class>

<owl:Class rdf:about="#DataPropertyAxioms">
    <rdfs:subClassOf rdf:resource="#Axiom"/>
</owl:Class>

<owl:Class rdf:about="#Document">
    <rdfs:subClassOf rdf:resource="&owl;Thing"/>
</owl:Class>

<owl:Class rdf:about="#Entity">
    <rdfs:subClassOf rdf:resource="#OntologyElements"/>
</owl:Class>

<owl:Class rdf:about="#GroupConcept">
    <rdfs:subClassOf rdf:resource="#CompositeChange"/>
</owl:Class>

<owl:Class rdf:about="#Guest">
    <rdfs:subClassOf rdf:resource="#User"/>
</owl:Class>

<owl:Class rdf:about="#IndividualAxioms">
    <rdfs:subClassOf rdf:resource="#Axiom"/>
</owl:Class>

<owl:Class rdf:about="#MergeConcept">
    <rdfs:subClassOf rdf:resource="#CompositeChange"/>
</owl:Class>

<owl:Class rdf:about="#MoveDownConcept">
    <rdfs:subClassOf rdf:resource="#CompositeChange"/>
</owl:Class>

<owl:Class rdf:about="#MoveUpConcept">
    <rdfs:subClassOf rdf:resource="#CompositeChange"/>
</owl:Class>

<owl:Class rdf:about="#ObjectPropertyAxioms">
    <rdfs:subClassOf rdf:resource="#Axiom"/>
</owl:Class>

<owl:Class rdf:about="#Ontology">
    <rdfs:subClassOf rdf:resource="&owl;Thing"/>
</owl:Class>

<owl:Class rdf:about="#OntologyElements"/>

<owl:Class rdf:about="#Parameter">
    <rdfs:subClassOf rdf:resource="&owl;Thing"/>
</owl:Class>

<owl:Class rdf:about="#PatternChange">
    <rdfs:subClassOf rdf:resource="#ChangeOperation"/>
</owl:Class>

<owl:Class rdf:about="#PhDStudentRegistration">
    <rdfs:subClassOf rdf:resource="#PatternChange"/>
</owl:Class>

<owl:Class rdf:about="#PullDownProperty">
    <rdfs:subClassOf rdf:resource="#CompositeChange"/>
</owl:Class>

<owl:Class rdf:about="#PullUpProperty">
    <rdfs:subClassOf rdf:resource="#CompositeChange"/>
```

```
</owl:Class>

<owl:Class rdf:about="#RenameEntity">
    <rdfs:subClassOf rdf:resource="#CompositeChange"/>
</owl:Class>

<owl:Class rdf:about="#Restriction">
    <rdfs:subClassOf rdf:resource="#OntologyElements"/>
</owl:Class>

<owl:Class rdf:about="#SplitConcept">
    <rdfs:subClassOf rdf:resource="#CompositeChange"/>
</owl:Class>

<owl:Class rdf:about="#TargetParam">
    <rdfs:subClassOf rdf:resource="#Parameter"/>
</owl:Class>

<owl:Class rdf:about="#Trace">
    <rdfs:subClassOf rdf:resource="&owl;Thing"/>
</owl:Class>

<owl:Class rdf:about="#User"/>

<owl:Class rdf:about="#add">
    <rdfs:subClassOf rdf:resource="#AtomicChange"/>
</owl:Class>

<owl:Class rdf:about="#classAssertionAxiom">
    <rdfs:subClassOf rdf:resource="#IndividualAxioms"/>
</owl:Class>

<owl:Class rdf:about="#concept">
    <rdfs:subClassOf rdf:resource="#Entity"/>
</owl:Class>

<owl:Class rdf:about="#dataProperty">
    <rdfs:subClassOf rdf:resource="#Entity"/>
</owl:Class>

<owl:Class rdf:about="#dataPropertyAssertionAxiom">
    <rdfs:subClassOf rdf:resource="#IndividualAxioms"/>
</owl:Class>

<owl:Class rdf:about="#delete">
    <rdfs:subClassOf rdf:resource="#AtomicChange"/>
</owl:Class>

<owl:Class rdf:about="#differentFromAxiom">
    <rdfs:subClassOf rdf:resource="#IndividualAxioms"/>
</owl:Class>

<owl:Class rdf:about="#disjointClassAxiom">
    <rdfs:subClassOf rdf:resource="#ClassAxioms"/>
</owl:Class>

<owl:Class rdf:about="#disjointDataPropertyAxiom">
    <rdfs:subClassOf rdf:resource="#DataPropertyAxioms"/>
</owl:Class>

<owl:Class rdf:about="#disjointObjectPropertyAxiom">
    <rdfs:subClassOf rdf:resource="#ObjectPropertyAxioms"/>
</owl:Class>

<owl:Class rdf:about="#domainOfDataPropertyAxiom">
    <rdfs:subClassOf rdf:resource="#DataPropertyAxioms"/>
</owl:Class>

<owl:Class rdf:about="#domainOfObjectPropertyAxiom">
    <rdfs:subClassOf rdf:resource="#ObjectPropertyAxioms"/>
</owl:Class>

<owl:Class rdf:about="#equivalentClassAxiom">
    <rdfs:subClassOf rdf:resource="#ClassAxioms"/>
</owl:Class>

<owl:Class rdf:about="#equivalentDataPropertyAxiom">
    <rdfs:subClassOf rdf:resource="#DataPropertyAxioms"/>
</owl:Class>

<owl:Class rdf:about="#equivalentObjectPropertyAxiom">
    <rdfs:subClassOf rdf:resource="#ObjectPropertyAxioms"/>
</owl:Class>
```

```
<owl:Class rdf:about="#functionalDataPropertyAxiom">
    <rdfs:subClassOf rdf:resource="#DataPropertyAxioms"/>
</owl:Class>

<owl:Class rdf:about="#functionalObjectPropertyAxiom">
    <rdfs:subClassOf rdf:resource="#ObjectPropertyAxioms"/>
</owl:Class>

<owl:Class rdf:about="#invFunctionalObjectPropertyAxiom">
    <rdfs:subClassOf rdf:resource="#ObjectPropertyAxioms"/>
</owl:Class>

<owl:Class rdf:about="#inverseObjectPropertyAxiom">
    <rdfs:subClassOf rdf:resource="#ObjectPropertyAxioms"/>
</owl:Class>

<owl:Class rdf:about="#maxCardinality">
    <rdfs:subClassOf rdf:resource="#Cardinality"/>
</owl:Class>

<owl:Class rdf:about="#minCardinality">
    <rdfs:subClassOf rdf:resource="#Cardinality"/>
</owl:Class>

<owl:Class rdf:about="#namedIndividual">
    <rdfs:subClassOf rdf:resource="#Entity"/>
</owl:Class>

<owl:Class rdf:about="#negativeDataPropertyAssertionAxiom">
    <rdfs:subClassOf rdf:resource="#IndividualAxioms"/>
</owl:Class>

<owl:Class rdf:about="#negativeObjectPropertyAssertionAxiom">
    <rdfs:subClassOf rdf:resource="#IndividualAxioms"/>
</owl:Class>

<owl:Class rdf:about="#objectProperty">
    <rdfs:subClassOf rdf:resource="#Entity"/>
</owl:Class>

<owl:Class rdf:about="#objectPropertyAssertionAxiom">
    <rdfs:subClassOf rdf:resource="#IndividualAxioms"/>
</owl:Class>

<owl:Class rdf:about="#postCondition">
    <rdfs:subClassOf rdf:resource="#ChangeCondition"/>
</owl:Class>

<owl:Class rdf:about="#preCondition">
    <rdfs:subClassOf rdf:resource="#ChangeCondition"/>
</owl:Class>

<owl:Class rdf:about="#rangeOfDataPropertyAxiom">
    <rdfs:subClassOf rdf:resource="#DataPropertyAxioms"/>
</owl:Class>

<owl:Class rdf:about="#rangeOfObjectPropertyAxiom">
    <rdfs:subClassOf rdf:resource="#ObjectPropertyAxioms"/>
</owl:Class>

<owl:Class rdf:about="#sameAsAxiom">
    <rdfs:subClassOf rdf:resource="#IndividualAxioms"/>
</owl:Class>

<owl:Class rdf:about="#subDataPropertyAxiom">
    <rdfs:subClassOf rdf:resource="#DataPropertyAxioms"/>
</owl:Class>

<owl:Class rdf:about="#subObjectPropertyAxiom">
    <rdfs:subClassOf rdf:resource="#ObjectPropertyAxioms"/>
</owl:Class>

<owl:Class rdf:about="#subclassOfAxiom">
    <rdfs:subClassOf rdf:resource="#ClassAxioms"/>
</owl:Class>

<owl:Class rdf:about="#symmetricObjectPropertyAxiom">
    <rdfs:subClassOf rdf:resource="#ObjectPropertyAxioms"/>
</owl:Class>

<owl:Class rdf:about="#transitiveObjectPropertyAxiom">
    <rdfs:subClassOf rdf:resource="#ObjectPropertyAxioms"/>
    </owl:Class>

    <owl:Class rdf:about="#valueRestriction">
        <rdfs:subClassOf rdf:resource="#Restriction"/>
    </owl:Class>

    <owl:Class rdf:about="&owl;Thing"/>

    <!--
    ///////////////////////////////////////////////////////////////
    //
    // Individuals
    //
   ///////////////////////////////////////////////////////////////
     -->

    <Administrator rdf:about="#Javed"/>

    <Administrator rdf:about="#Yalemisew"/>

    <!--
    ///////////////////////////////////////////////////////////////
    //
    // General axioms
    //
   ///////////////////////////////////////////////////////////////
     -->

    <rdf:Description>
        <rdf:type rdf:resource="&owl;AllDisjointClasses"/>
        <owl:members rdf:parseType="Collection">
            <rdf:Description rdf:about="#Change"/>
            <rdf:Description rdf:about="#ChangeCondition"/>
            <rdf:Description rdf:about="#ChangeOperation"/>
            <rdf:Description rdf:about="#OntologyElements"/>
            <rdf:Description rdf:about="#Parameter"/>
            <rdf:Description rdf:about="#User"/>
        </owl:members>
    </rdf:Description>
</rdf:RDF>

<!-- Generated by the OWL API (version 2.2.1.842) http://owlapi.sourceforge.net -->
```

Appendix B

Java Code: OrderedChangePatternFinder()

In this appendix, we give the Java implementation of discovering the ordered complete (OC) domain-specific change patterns (c.f. Section 8.2). The input to the Java method is minimum pattern length (`min_l`), minimum pattern support (`min_s`), and the permissible sequence gap between two adjacent nodes of a sequence (`gap`).

```
Input:
- Minimum Pattern Support (min_s),
- Minimum Pattern Length (min_l),
- Permissible node distance (gap)
Output:
- List of Ordered Change Patterns


public static void OrderedChangePatternFinder(int min_s, int min_l, int gap){
        min_sup = min_s;
        min_len = min_l;
        gap_const = gap;
        resultList = new ArrayList();
        resultList.clear();

        Set s = GraphCreator.graph.vertexSet();
        ArrayList Cand_Sequence = null;
        ArrayList Supp_sequence = null;
        ArrayList Result = null;

 //Iterator to select a node from the graph set as a candidate nodes
```

```
iteration: for(Iterator i = s.iterator(); i.hasNext();) {
           Cand_Sequence = new ArrayList();
           Cand_Sequence.clear();
           Supp_sequence = new ArrayList();
           Supp_sequence.clear();
           Result = new ArrayList();
           Result.clear();
           Cand_Context = new ArrayList();
           Cand_Context.clear();
           cContext = new ArrayList();
           cContext.clear();

           GraphNode Cand_Node = (GraphNode)i.next();
           int Cand_Node_Id = Cand_Node.getNodeID();

           Cand_Context.add(Cand_Node.getParam1());
           Cand_Sequence.add(Cand_Node);

           boolean ctx = true;
           while(ctx == true)
           {
            GraphNode Nxt_Cand_Node =
             PatternSearcher.findNode(++Cand_Node_Id);
               if(Nxt_Cand_Node != null)
                    {
                      Nxt_Cand_Node =
                        GraphNodeMatcher.contextXMatch(Nxt_Cand_Node, gap_const);
                    } //end of if
               if (Nxt_Cand_Node == null)
                    {
                         ctx = false;
                    }  //end of if
               else
                    {
                         Cand_Sequence.add(Nxt_Cand_Node);
                         Cand_Node_Id = Nxt_Cand_Node.getNodeID();
                    } //end of else
           }// end of while loop

           if(Cand_Sequence.size() < min_len)
               continue iteration;

           Supp_sequence =
            CandidateNodeSearcher.searchCandidateNodes(Cand_Node);

           if(Supp_sequence.size()+1 < min_sup)
               continue iteration;

           GraphNode nxtNode = null;
           GraphNode nxtCNode = null;

           int c_id;
           int counter1 = 0;
```

```
    a: while(counter1<Supp_sequence.size()&&Supp_sequence.size()+1>=min_sup){
                ArrayList a =  (ArrayList) Supp_sequence.get(counter1);
                nxtCNode = (GraphNode)a.get(0);
                c_id = nxtCNode.getNodeID();
                String target = nxtCNode.getParam1();

                GraphNode match2 = null;
                int count = 1;

            b:  while (count < Cand_Sequence.size()) {
                    nxtNode = (GraphNode) Cand_Sequence.get(count++);
                    GraphNode cn2 = PatternSearcher.findNode(++c_id);
                    if(nxtNode != null && cn2 != null) {
                     match2 =
                     GraphNodeMatcher.contextMatch(nxtNode,cn2,gap_const,target);
                     if(match2 != null) {
                           a.add(match2);
                           c_id = match2.getNodeID();
                     }// end of if
                     else
                          break b;
                     }// end of if
                 }//end of b: while loop

                if(a.size() < min_len ) {
                    Supp_sequence.remove(counter1);
                    counter1--;
                }// end of if
                counter1++;
            }// end of a: while


    int counting  = 0 ;
    int max = Cand_Sequence.size();
    int min = 0;

x:  while (max >= min_len){
        counting  = 0 ;
        for (int u = 0; u < Supp_sequence.size(); u++){
        ArrayList cp = (ArrayList) Supp_sequence.get(u);
           if(cp.size() >=  max)
                counting++;
        } // end of for loop
        if (counting >= min_sup) {
            min = max;
            break x;
        }// end of if
        if(counting < min_sup) {
            max--;
        }// end of if
    }// end of x:while loop
    for(int u = 0; u<Supp_sequence.size();u++)
    {
         ArrayList cp = (ArrayList) Supp_sequence.get(u);
         if(cp.size() < min){
             Supp_sequence.remove(u);
             u--;
         }// end of if

    }// end of for loop

        if( min >= min_len ){
            int r1 = Cand_Sequence.size()-1;
            while(r1 >= min)
            { Cand_Sequence.remove(r1--); }
            for(int u = 0; u<Supp_sequence.size();u++){
                    ArrayList cp = (ArrayList) Supp_sequence.get(u);
                    int r2 = cp.size()-1;
                    while(r2 >= min)
                    { cp.remove(r2--); }
                }// end of for loop
        }
    if(Supp_sequence.size()+1 >= min_sup) {
            Result.add(0, Cand_Sequence);
            for(int d = 0; d<Supp_sequence.size(); d++){
                       ArrayList a  = (ArrayList) Supp_sequence.get(d);
                       if( (a.size()>= min_len)){
                           Result.add(a);
                       }// end of if
                }// end of for
        }// end of if

      if(!subset && Result.size()>1)
          resultList = removeSubsets(Result, resultList);
      else if (Result.size()>1)
          resultList.add(Result);

   }// end of Iterations

printResult(resultList);

}
```

Appendix C

Java Code: UnorderedChangePatternFinder()

In this appendix, we give the Java implementation of discovering the unordered complete (UC) domain-specific change patterns (c.f. Section 8.2). The input to the Java method is minimum pattern length (`min_l`), minimum pattern support (`min_s`), and the permissible sequence gap between two adjacent nodes of a sequence (`gap`).

```
Input:
 - Minimum Pattern Support (min_s),
 - Minimum Pattern Length (min_l),
 - Permissible node distance (gap)
Output:
 - List of Unordered Change Patterns

public static void UnorderedChangePatternFinder(int min_s, int min_l, int gap)
  {
         min_sup = min_s;
         min_len = min_l;
         gap_const = gap;
         resultList = new ArrayList();
         resultList.clear();

          s = GraphCreator.graph.vertexSet();
        ArrayList Cand_Sequence = null;
        ArrayList Supp_Sequence = null;
         ArrayList StandeByList = null;
         ArrayList Result = null;
```

```
//Iteration to get all the nodes of graph and passing them
iteration:for(Iterator i = s.iterator(); i.hasNext();) {
         Cand_Sequence = new ArrayList(); Cand_Sequence.clear();
         Supp_Sequence = new ArrayList();  Supp_Sequence.clear();
         StandeByList = new ArrayList();  StandeByList.clear();
         Result = new ArrayList();  Result.clear();
         Cand_Context = new ArrayList();  Cand_Context.clear();
         cContext = new ArrayList(); cContext.clear();

          GraphNode node = (GraphNode)i.next();
          int Cand_Node_Id = node.getNodeID();
          Cand_Context.add(node.getParam1());
          Cand_Sequence.add(node);
          boolean ctx = true;
          while(ctx == true)
            {
               GraphNode Nxt_Cand_Node =  PatternSearcher.findNode(++Cand_Node_Id);
               if(Nxt_Cand_Node != null){
                          Nxt_Cand_Node =
                            GraphNodeMatcher.contextQMatch(Nxt_Cand_Node, gap_const);
                     }// end of if
               if (Nxt_Cand_Node == null) {
                          ctx = false;
                     }// end of if
                    else {
                         Cand_Sequence.add(Nxt_Cand_Node);
                         Cand_Node_Id = Nxt_Cand_Node.getNodeID();
                     }// end of else
            }// end of while loop

          if(Cand_Sequence.size() < min_len)
                continue iteration;

          Supp_Sequence = CandidateNodeSearcher.searchCandidateNodes(node);

          if(Supp_Sequence.size()+1 < min_sup)
                continue iteration;

          for(int h=0; h<Supp_Sequence.size(); h++){
                ArrayList s = new ArrayList();
                s.clear();
                StandeByList.add(s);
            }// end of for loop

          GraphNode nxtNode = null;
          int counter = 0;

    a: while( counter<Supp_Sequence.size() && Supp_Sequence.size() +1 >= min_sup){
                    ArrayList a =  (ArrayList) Supp_Sequence.get(counter);
                    ArrayList sb = (ArrayList) StandeByList.get(counter);
                    int count = 1;
          b:  while( count < Cand_Sequence.size() ){
                  nxtNode = (GraphNode) Cand_Sequence.get(count++);
            setRange(a);
            ArrayList na = searchInRange(a, nxtNode);
            if(na != null) {
                 a = ascendingForm(na);
                 setRange(a);
                 if(!sb.isEmpty()) {
                          int size_bsb, size_asb;
                          if(sb.size()>0 && a.size()>0){
                               do{
                                      size_bsb = sb.size();
                                      ArrayList next = searchinStandBy(sb, a);
                                      sb =     (ArrayList) next.get(0);
                                      a =      (ArrayList) next.get(1);
                                      a = ascendingForm(a);
                                      setRange(a);
                                      size_asb = sb.size();
                                      }while( sb.size()>0 && size_bsb > size_asb);
                          }//end of if

                } //end of if
             }//end of if
             else {
                sb.add(nxtNode);
             }

          }// end of b: while

    if(a.size() < min_len ) {
                  Supp_Sequence.remove(counter);
                  StandeByList.remove(counter);
                  counter--;
              }// end of if
    counter++;
    }// end of a: while

for(int d = 0; d < Supp_Sequence.size(); d++){
       ArrayList a  = (ArrayList) Supp_Sequence.get(d);
       if(a.size() < Cand_Sequence.size()){
          Supp_Sequence.remove(d);
          StandeByList.remove(d);
          d--;
      }

    }// end of for loop

    if(Supp_Sequence.size()+1 >= min_sup && Cand_Sequence.size() >= min_len) {
      Result.add(0, Cand_Sequence);
      for(int d = 0; d<Supp_Sequence.size(); d++) {
                 ArrayList a  = (ArrayList) Supp_Sequence.get(d);
                 if( (a.size()>= min_len)){
                     Result.add(a);
                 }// end of if
          }// end of for loop
          }//end of if

        if(!subset && Result.size()>1)
          resultList = removeSubsets(Result, resultList);
         else if (Result.size()>1)
          resultList.add(Result);

   }// end of all iterations

printResult(resultList);

}
```

Appendix D

University Administration Ontology

In this appendix, we give the current version of university case study ontology. The ontology covers the main constituents of the university domain. The entities such as faculty, student, departments courses, subjects, staff members have been defined and relate to each other.

```
<?xml version="1.0"?>
<rdf:RDF xmlns="http://www.cngl.ie/ontology/University.owl#"
     xml:base="http://www.cngl.ie/ontology/University.owl"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:University_Administration="http://www.cngl.ie/ontology/University.owl#"
     xmlns:owl="http://www.w3.org/2002/07/owl#"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <owl:Ontology rdf:about="http://www.cngl.ie/ontology/University.owl"/>

    <!--
    ///////////////////////////////////////////////////////////////////////
    //
    // Object Properties
    //
    ///////////////////////////////////////////////////////////////////////
    -->

    <owl:ObjectProperty rdf:about="http://www.cngl.ie/ontology/University.owl#CourseContact">
        <rdfs:domain rdf:resource="http://www.cngl.ie/ontology/University.owl#Course"/>
        <rdfs:range rdf:resource="http://www.cngl.ie/ontology/University.owl#Faculty"/>
    </owl:ObjectProperty>

    <owl:ObjectProperty rdf:about="http://www.cngl.ie/ontology/University.owl#EventVenue">
        <rdfs:domain rdf:resource="http://www.cngl.ie/ontology/University.owl#Event"/>
        <rdfs:range rdf:resource="http://www.cngl.ie/ontology/University.owl#School"/>
    </owl:ObjectProperty>

    <owl:ObjectProperty rdf:about="http://www.cngl.ie/ontology/University.owl#OfferedBySchool">
        <rdfs:domain rdf:resource="http://www.cngl.ie/ontology/University.owl#Course"/>
        <rdfs:range rdf:resource="http://www.cngl.ie/ontology/University.owl#School"/>
    </owl:ObjectProperty>
```

```
<owl:ObjectProperty rdf:about="http://www.cngl.ie/ontology/University.owl#RegisteredIn">
    <rdfs:range rdf:resource="http://www.cngl.ie/ontology/University.owl#Course"/>
    <rdfs:domain rdf:resource="http://www.cngl.ie/ontology/University.owl#Taught_Student"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="http://www.cngl.ie/ontology/University.owl#ResearchMemberOf">
    <rdfs:range rdf:resource="http://www.cngl.ie/ontology/University.owl#Research_Group"/>
    <rdfs:domain rdf:resource="http://www.cngl.ie/ontology/University.owl#Research_Student"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="http://www.cngl.ie/ontology/University.owl#VacancyContact">
    <rdfs:range rdf:resource="http://www.cngl.ie/ontology/University.owl#Faculty"/>
    <rdfs:domain rdf:resource="http://www.cngl.ie/ontology/University.owl#Vacancy"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="http://www.cngl.ie/ontology/University.owl#VacancyOfferedIn">
    <rdfs:range rdf:resource="http://www.cngl.ie/ontology/University.owl#School"/>
    <rdfs:domain rdf:resource="http://www.cngl.ie/ontology/University.owl#Vacancy"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="http://www.cngl.ie/ontology/University.owl#hasCountryOfOrigin">
    <rdfs:range rdf:resource="http://www.cngl.ie/ontology/University.owl#Country"/>
    <rdfs:domain rdf:resource="http://www.cngl.ie/ontology/University.owl#Student"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="http://www.cngl.ie/ontology/University.owl#hasCourseCode">
    <rdfs:domain rdf:resource="http://www.cngl.ie/ontology/University.owl#Course"/>
    <rdfs:range rdf:resource="http://www.cngl.ie/ontology/University.owl#CourseCode"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="http://www.cngl.ie/ontology/University.owl#hasEligibilityReq">
    <rdfs:domain rdf:resource="http://www.cngl.ie/ontology/University.owl#Course"/>
    <rdfs:range rdf:resource="http://www.cngl.ie/ontology/University.owl#DegreeLevel"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="http://www.cngl.ie/ontology/University.owl#hasPreRequisite">
    <rdfs:range rdf:resource="http://www.cngl.ie/ontology/University.owl#Subject"/>
    <rdfs:domain rdf:resource="http://www.cngl.ie/ontology/University.owl#Subject"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="http://www.cngl.ie/ontology/University.owl#hasSubject">
    <rdfs:domain rdf:resource="http://www.cngl.ie/ontology/University.owl#Course"/>
    <rdfs:range rdf:resource="http://www.cngl.ie/ontology/University.owl#Subject"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="http://www.cngl.ie/ontology/University.owl#hasSubjectCode">
    <rdfs:domain rdf:resource="http://www.cngl.ie/ontology/University.owl#Subject"/>
    <rdfs:range rdf:resource="http://www.cngl.ie/ontology/University.owl#SubjectCode"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="http://www.cngl.ie/ontology/University.owl#hasSupervisor">
    <rdfs:range rdf:resource="http://www.cngl.ie/ontology/University.owl#Faculty"/>
    <rdfs:domain rdf:resource="http://www.cngl.ie/ontology/University.owl#Research_Student"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="http://www.cngl.ie/ontology/University.owl#isCommitteeMemberOf">
    <rdfs:range rdf:resource="http://www.cngl.ie/ontology/University.owl#Committee"/>
    <rdfs:domain rdf:resource="http://www.cngl.ie/ontology/University.owl#Faculty"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="http://www.cngl.ie/ontology/University.owl#isFacultyOf">
    <rdfs:domain rdf:resource="http://www.cngl.ie/ontology/University.owl#Faculty"/>
    <rdfs:range rdf:resource="http://www.cngl.ie/ontology/University.owl#School"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="http://www.cngl.ie/ontology/University.owl#isOfferedInCourse">
    <rdfs:range rdf:resource="http://www.cngl.ie/ontology/University.owl#Course"/>
    <rdfs:domain rdf:resource="http://www.cngl.ie/ontology/University.owl#Subject"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="http://www.cngl.ie/ontology/University.owl#isStudentOf">
    <rdfs:range rdf:resource="http://www.cngl.ie/ontology/University.owl#School"/>
    <rdfs:domain rdf:resource="http://www.cngl.ie/ontology/University.owl#Student"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="http://www.cngl.ie/ontology/University.owl#isStudying">
    <rdfs:range rdf:resource="http://www.cngl.ie/ontology/University.owl#Course"/>
    <rdfs:domain rdf:resource="http://www.cngl.ie/ontology/University.owl#Taught_Student"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="http://www.cngl.ie/ontology/University.owl#isSupervisorOf">
    <rdfs:domain rdf:resource="http://www.cngl.ie/ontology/University.owl#Faculty"/>
    <rdfs:range rdf:resource="http://www.cngl.ie/ontology/University.owl#Research_Student"/>
```

```
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="http://www.cngl.ie/ontology/University.owl#isTakingSubject">
    <rdfs:range rdf:resource="http://www.cngl.ie/ontology/University.owl#Subject"/>
    <rdfs:domain rdf:resource="http://www.cngl.ie/ontology/University.owl#Taught_Student"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="http://www.cngl.ie/ontology/University.owl#isTeaching">
    <rdfs:domain rdf:resource="http://www.cngl.ie/ontology/University.owl#Faculty"/>
    <rdfs:range rdf:resource="http://www.cngl.ie/ontology/University.owl#Subject"/>
</owl:ObjectProperty>

<!--
///////////////////////////////////////////////////////////////
//
// Data properties
//
///////////////////////////////////////////////////////////////
 -->

<owl:DatatypeProperty rdf:about="http://www.cngl.ie/ontology/University.owl#CourseTitle">
    <rdfs:domain rdf:resource="http://www.cngl.ie/ontology/University.owl#Course"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="http://www.cngl.ie/ontology/University.owl#CreditHours">
    <rdfs:domain rdf:resource="http://www.cngl.ie/ontology/University.owl#Subject"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="http://www.cngl.ie/ontology/University.owl#DurationOfCourse">
    <rdfs:domain rdf:resource="http://www.cngl.ie/ontology/University.owl#Course"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#long"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="http://www.cngl.ie/ontology/University.owl#EmailID">
    <rdfs:domain rdf:resource="http://www.cngl.ie/ontology/University.owl#Faculty"/>
    <rdfs:domain rdf:resource="http://www.cngl.ie/ontology/University.owl#Student"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="http://www.cngl.ie/ontology/University.owl#EventDate">
    <rdfs:domain rdf:resource="http://www.cngl.ie/ontology/University.owl#Event"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="http://www.cngl.ie/ontology/University.owl#EventDescription">
    <rdfs:domain rdf:resource="http://www.cngl.ie/ontology/University.owl#Event"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="http://www.cngl.ie/ontology/University.owl#EventLink">
    <rdfs:domain rdf:resource="http://www.cngl.ie/ontology/University.owl#Event"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="http://www.cngl.ie/ontology/University.owl#Ref.Material">
    <rdfs:domain rdf:resource="http://www.cngl.ie/ontology/University.owl#Subject"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="http://www.cngl.ie/ontology/University.owl#StudentID">
    <rdfs:domain rdf:resource="http://www.cngl.ie/ontology/University.owl#Student"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#long"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="http://www.cngl.ie/ontology/University.owl#SubjectTitle">
    <rdfs:domain rdf:resource="http://www.cngl.ie/ontology/University.owl#Subject"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="http://www.cngl.ie/ontology/University.owl#VacancyClosingDate">
    <rdfs:domain rdf:resource="http://www.cngl.ie/ontology/University.owl#Vacancy"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="http://www.cngl.ie/ontology/University.owl#VacancyDescription">
    <rdfs:domain rdf:resource="http://www.cngl.ie/ontology/University.owl#Vacancy"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="http://www.cngl.ie/ontology/University.owl#VacancyPostedOn">
    <rdfs:domain rdf:resource="http://www.cngl.ie/ontology/University.owl#Vacancy"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="http://www.cngl.ie/ontology/University.owl#VacancySalary">
    <rdfs:domain rdf:resource="http://www.cngl.ie/ontology/University.owl#Vacancy"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#double"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="http://www.cngl.ie/ontology/University.owl#VacancyTitle">
    <rdfs:domain rdf:resource="http://www.cngl.ie/ontology/University.owl#Vacancy"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<!--
///////////////////////////////////////////////////////////////
//
// Classes
//
///////////////////////////////////////////////////////////////
 -->

<owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#AcademicVacancy">
    <rdfs:subClassOf rdf:resource="http://www.cngl.ie/ontology/University.owl#Vacancy"/>
</owl:Class>

<owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#AdministrativeVacancy">
    <rdfs:subClassOf rdf:resource="http://www.cngl.ie/ontology/University.owl#Vacancy"/>
</owl:Class>

<owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#AssociateProfessor">
    <rdfs:subClassOf rdf:resource="http://www.cngl.ie/ontology/University.owl#Faculty"/>
</owl:Class>

<owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#Committee">
    <rdfs:subClassOf rdf:resource="http://www.cngl.ie/ontology/University.owl#University"/>
</owl:Class>

<owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#Country"/>

<owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#Course">
    <rdfs:subClassOf rdf:resource="http://www.cngl.ie/ontology/University.owl#University"/>
</owl:Class>

<owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#CourseCode">
    <rdfs:subClassOf rdf:resource="http://www.cngl.ie/ontology/University.owl#University"/>
</owl:Class>

<owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#DegreeLevel"/>

<owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#EuropeanStudent">
    <rdfs:subClassOf rdf:resource="http://www.cngl.ie/ontology/University.owl#InternationalStudent"/>
</owl:Class>

<owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#Event">
    <rdfs:subClassOf rdf:resource="http://www.cngl.ie/ontology/University.owl#University"/>
</owl:Class>

<owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#Faculty">
    <rdfs:subClassOf rdf:resource="http://www.cngl.ie/ontology/University.owl#University"/>
</owl:Class>

<owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#InternationalStudent">
    <rdfs:subClassOf rdf:resource="http://www.cngl.ie/ontology/University.owl#Student"/>
</owl:Class>

<owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#Lecturer">
    <rdfs:subClassOf rdf:resource="http://www.cngl.ie/ontology/University.owl#Faculty"/>
</owl:Class>

<owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#Library">
    <rdfs:subClassOf rdf:resource="http://www.cngl.ie/ontology/University.owl#University"/>
</owl:Class>

<owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#MS_Course">
    <rdfs:subClassOf rdf:resource="http://www.cngl.ie/ontology/University.owl#Course"/>
</owl:Class>

<owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#MS_Student">
    <rdfs:subClassOf rdf:resource="http://www.cngl.ie/ontology/University.owl#Taught_Student"/>
</owl:Class>

<owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#MSbyResearch_Student">
    <rdfs:subClassOf rdf:resource="http://www.cngl.ie/ontology/University.owl#Research_Student"/>
```

```
</owl:Class>

<owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#Non-EuropeanStudent">
    <rdfs:subClassOf rdf:resource="http://www.cngl.ie/ontology/University.owl#InternationalStudent"/>
</owl:Class>

<owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#OtherEvent">
    <rdfs:subClassOf rdf:resource="http://www.cngl.ie/ontology/University.owl#Event"/>
</owl:Class>

<owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#PhD_Student">
    <rdfs:subClassOf rdf:resource="http://www.cngl.ie/ontology/University.owl#Research_Student"/>
</owl:Class>

<owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#Professor">
    <rdfs:subClassOf rdf:resource="http://www.cngl.ie/ontology/University.owl#Faculty"/>
</owl:Class>

<owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#ResearchEvent">
    <rdfs:subClassOf rdf:resource="http://www.cngl.ie/ontology/University.owl#Event"/>
</owl:Class>

<owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#ResearchVacancy">
    <rdfs:subClassOf rdf:resource="http://www.cngl.ie/ontology/University.owl#Vacancy"/>
</owl:Class>

<owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#Research_Group">
    <rdfs:subClassOf rdf:resource="http://www.cngl.ie/ontology/University.owl#University"/>
</owl:Class>

<owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#Research_Student">
    <rdfs:subClassOf rdf:resource="http://www.cngl.ie/ontology/University.owl#Student"/>
</owl:Class>

<owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#School">
    <rdfs:subClassOf rdf:resource="http://www.cngl.ie/ontology/University.owl#University"/>
</owl:Class>

<owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#Semester">
    <rdfs:subClassOf rdf:resource="http://www.cngl.ie/ontology/University.owl#University"/>
</owl:Class>

<owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#SeniorLecturer">
    <rdfs:subClassOf rdf:resource="http://www.cngl.ie/ontology/University.owl#Faculty"/>
</owl:Class>

<owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#SocietyEvent">
    <rdfs:subClassOf rdf:resource="http://www.cngl.ie/ontology/University.owl#Event"/>
</owl:Class>

<owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#SportsEvent">
    <rdfs:subClassOf rdf:resource="http://www.cngl.ie/ontology/University.owl#Event"/>
</owl:Class>

<owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#Student">
    <rdfs:subClassOf rdf:resource="http://www.cngl.ie/ontology/University.owl#University"/>
</owl:Class>

<owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#Subject">
    <rdfs:subClassOf rdf:resource="http://www.cngl.ie/ontology/University.owl#University"/>
</owl:Class>

<owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#SubjectCode">
    <rdfs:subClassOf rdf:resource="http://www.cngl.ie/ontology/University.owl#University"/>
</owl:Class>

<owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#SummerSemester">
    <rdfs:subClassOf rdf:resource="http://www.cngl.ie/ontology/University.owl#Semester"/>
</owl:Class>

<owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#Taught_Student">
    <rdfs:subClassOf rdf:resource="http://www.cngl.ie/ontology/University.owl#Student"/>
</owl:Class>

<owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#UG_Course">
    <rdfs:subClassOf rdf:resource="http://www.cngl.ie/ontology/University.owl#Course"/>
</owl:Class>

<owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#UG_Student">
    <rdfs:subClassOf rdf:resource="http://www.cngl.ie/ontology/University.owl#Taught_Student"/>
</owl:Class>

<owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#University"/>
```

```
    <owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#Vacancy">
        <rdfs:subClassOf rdf:resource="http://www.cngl.ie/ontology/University.owl#University"/>
    </owl:Class>

    <owl:Class rdf:about="http://www.cngl.ie/ontology/University.owl#WinterSemester">
        <rdfs:subClassOf rdf:resource="http://www.cngl.ie/ontology/University.owl#Semester"/>
    </owl:Class>

    <!--
    /////////////////////////////////////////////////////////////////
    //
    // Individuals
    //
    /////////////////////////////////////////////////////////////////
     -->

    <owl:NamedIndividual rdf:about="http://www.cngl.ie/ontology/University.owl#CNGL">
        <rdf:type rdf:resource="http://www.cngl.ie/ontology/University.owl#Research_Group"/>
    </owl:NamedIndividual>

    <owl:NamedIndividual rdf:about="http://www.cngl.ie/ontology/University.owl#ClausPahl">
        <rdf:type rdf:resource="http://www.cngl.ie/ontology/University.owl#Research_Group"/>
    </owl:NamedIndividual>

    <owl:NamedIndividual rdf:about="http://www.cngl.ie/ontology/University.owl#Javed">
        <rdf:type rdf:resource="http://www.cngl.ie/ontology/University.owl#PhD_Student"/>
        <StudentID rdf:datatype="http://www.w3.org/2001/XMLSchema#float">5.8106384E7</StudentID>
        <EmailID rdf:datatype="http://www.w3.org/2001/XMLSchema#string">mjaved@computing.dcu.ie</EmailID>
        <hasSupervisor rdf:resource="http://www.cngl.ie/ontology/University.owl#ClausPahl"/>
        <isStudentOf rdf:resource="http://www.cngl.ie/ontology/University.owl#SchoolOfComputing"/>
    </owl:NamedIndividual>

    <owl:NamedIndividual rdf:about="http://www.cngl.ie/ontology/University.owl#SchoolOfComputing">
        <rdf:type rdf:resource="http://www.cngl.ie/ontology/University.owl#School"/>
    </owl:NamedIndividual>

    <owl:NamedIndividual rdf:about="http://www.cngl.ie/ontology/University.owl#Yalemisew">
        <rdf:type rdf:resource="http://www.cngl.ie/ontology/University.owl#PhD_Student"/>
    </owl:NamedIndividual>
</rdf:RDF>
```

Appendix E

Database System Ontology

In this appendix, we give the current version of database (technical) case study ontology (c.f. Section 4.1.1).

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [
    <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
    <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
    <!ENTITY owl2xml "http://www.w3.org/2006/12/owl2-xml#" >
    <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
    <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
    <!ENTITY OntologyDatabase "http://www.semanticweb.org/ontologies/2008/10/OntologyDatabase.owl#" >
    <!ENTITY Modify "http://www.semanticweb.org/ontologies/2008/10/OntologyDatabase.owl#Modify/" >
    <!ENTITY Values "http://www.semanticweb.org/ontologies/2008/10/OntologyDatabase.owl#Values&amp;" >
    <!ENTITY Formula "http://www.semanticweb.org/ontologies/2008/10/OntologyDatabase.owl#Formula/" >
    <!ENTITY Inner_Join "http://www.semanticweb.org/ontologies/2008/10/OntologyDatabase.owl#Inner_Join/" >
]>
<rdf:RDF xmlns="http://www.semanticweb.org/ontologies/2008/10/OntologyDatabase.owl#"
     xml:base="http://www.semanticweb.org/ontologies/2008/10/OntologyDatabase.owl"
     xmlns:owl2xml="http://www.w3.org/2006/12/owl2-xml#"
     xmlns:Values="&OntologyDatabase;Values&amp;"
     xmlns:OntologyDatabase="http://www.semanticweb.org/ontologies/2008/10/OntologyDatabase.owl#"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:Inner_Join="&OntologyDatabase;Inner_Join/"
     xmlns:Modify="&OntologyDatabase;Modify/"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:Formula="&OntologyDatabase;Formula/"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:owl="http://www.w3.org/2002/07/owl#">
    <owl:Ontology rdf:about=""/>
    <!--
    ///////////////////////////////////////////////////////////////////////////////////////
    //
    // Object Properties
    //
    ///////////////////////////////////////////////////////////////////////////////////////
     -->
    <owl:ObjectProperty rdf:about="#Minus">
        <rdfs:range rdf:resource="#Rows"/>
        <rdfs:domain rdf:resource="#SET_DIFFERENCE"/>
    </owl:ObjectProperty>

    <owl:ObjectProperty rdf:about="#Renames">
        <rdfs:range rdf:resource="#Columns"/>
        <rdfs:domain rdf:resource="#Rename"/>
    </owl:ObjectProperty>

    <owl:ObjectProperty rdf:about="#changesValuesOf">
        <rdfs:range rdf:resource="#Table"/>
        <rdfs:domain rdf:resource="#UpdateOperations"/>
```

```
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasColumn">
    <rdfs:range rdf:resource="#Columns"/>
    <rdfs:domain rdf:resource="#Table"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasConstraints">
    <rdfs:range rdf:resource="#Constraints"/>
    <rdfs:domain rdf:resource="#Value"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasDatatype">
    <rdfs:range rdf:resource="#Data_Type"/>
    <rdfs:domain rdf:resource="#Domain"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasDomain">
    <rdfs:domain rdf:resource="#Column"/>
    <rdfs:range rdf:resource="#Domain"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasDomainConstraint">
    <rdfs:domain rdf:resource="#Domain"/>
    <rdfs:range rdf:resource="#Domain_Constraint"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasEntityIntegrityConstraint">
    <rdfs:range rdf:resource="#Entity_Integrity_Constraint"/>
    <rdfs:domain rdf:resource="#Primary_Key"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasKey">
    <rdfs:range rdf:resource="#Key"/>
    <rdfs:domain rdf:resource="#Table"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasKeyConstraint">
    <rdfs:domain rdf:resource="#Key"/>
    <rdfs:range rdf:resource="#Key_Constraint"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasNULLConstraint">
    <rdfs:range rdf:resource="#Constraint_On_NULL"/>
    <rdfs:domain rdf:resource="#NULL_Value"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasQueryLanguage">
    <rdfs:domain rdf:resource="#Relational_Database"/>
    <rdfs:range rdf:resource="#Structure_Query_Language"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasRetrievalOperations">
    <rdfs:range rdf:resource="#Relational_Algebra_Operations"/>
    <rdfs:domain rdf:resource="#Relational_Database"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasRow">
    <rdfs:range rdf:resource="#Rows"/>
    <rdfs:domain rdf:resource="#Table"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasSchema">
    <rdfs:domain rdf:resource="#Relational_Database"/>
    <rdfs:range rdf:resource="#Relational_Database_Schema"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasTable">
    <rdfs:domain rdf:resource="#Relational_Database"/>
    <rdfs:range rdf:resource="#Table"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasUpdateOperations">
    <rdfs:domain rdf:resource="#Relational_Database"/>
    <rdfs:range rdf:resource="#UpdateOperations"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasValue">
    <rdfs:domain rdf:resource="#Data_Type"/>
    <rdfs:range rdf:resource="#Value"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#inserts">
```

```
    <rdfs:domain rdf:resource="#Insert"/>
    <rdfs:range rdf:resource="#Rows"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#intersects">
    <rdfs:domain rdf:resource="#INTERSECTION"/>
    <rdfs:range rdf:resource="#Rows"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#joinRelatedRowsFrom">
    <rdfs:domain rdf:resource="#Join"/>
    <rdfs:range rdf:resource="#Table"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#mayContain">
    <rdfs:domain rdf:resource="#Formula/Condition"/>
    <rdfs:range rdf:resource="#Logical_Operators"/>
    <rdfs:range rdf:resource="#Quantifiers"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#mayViolate"/>

<owl:ObjectProperty rdf:about="#modifies">
    <rdfs:domain rdf:resource="#Modify/Update"/>
    <rdfs:range rdf:resource="#Rows"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#removes">
    <rdfs:domain rdf:resource="#Delete"/>
    <rdfs:range rdf:resource="#Rows"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#retrievesValuesFrom">
    <rdfs:domain rdf:resource="#Relational_Algebra_Operations"/>
    <rdfs:range rdf:resource="#Table"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#selectC">
    <rdfs:range rdf:resource="#Columns"/>
    <rdfs:domain rdf:resource="#Project"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#selectR">
    <rdfs:range rdf:resource="#Rows"/>
    <rdfs:domain rdf:resource="#Select"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#unite">
    <rdfs:range rdf:resource="#Rows"/>
    <rdfs:domain rdf:resource="#UNION"/>
</owl:ObjectProperty>

<!--
///////////////////////////////////////////////////////////////////////////////////////
//
// Classes
//
///////////////////////////////////////////////////////////////////////////////////////
 -->

<owl:Class rdf:about="#AND">
    <rdfs:subClassOf rdf:resource="#Logical_Operators"/>
</owl:Class>

<owl:Class rdf:about="#Additional_Relational_Operations">
    <rdfs:subClassOf rdf:resource="#Relational_Algebra_Operations"/>
</owl:Class>

<owl:Class rdf:about="#Aggregate_Function">
    <rdfs:subClassOf rdf:resource="#Additional_Relational_Operations"/>
</owl:Class>

<owl:Class rdf:about="#Alternate_Key">
    <rdfs:subClassOf rdf:resource="#Key"/>
</owl:Class>

<owl:Class rdf:about="#Binary_Model">
    <rdfs:subClassOf rdf:resource="#Object_Based_Logical_Model"/>
</owl:Class>

<owl:Class rdf:about="#Bit_String">
    <rdfs:subClassOf rdf:resource="#Data_Type"/>
</owl:Class>
```

```
<owl:Class rdf:about="#Boolean">
    <rdfs:subClassOf rdf:resource="#Data_Type"/>
</owl:Class>

<owl:Class rdf:about="#CARTESIAN_PRODUCT">
    <rdfs:subClassOf rdf:resource="#Set_Operations"/>
</owl:Class>

<owl:Class rdf:about="#Character_String">
    <rdfs:subClassOf rdf:resource="#Data_Type"/>
</owl:Class>

<owl:Class rdf:about="#Column">
    <rdfs:subClassOf rdf:resource="#Data_Type"/>
</owl:Class>

<owl:Class rdf:about="#Columns">
    <rdfs:subClassOf rdf:resource="#Relational_Database"/>
</owl:Class>

<owl:Class rdf:about="#Constraint_On_NULL">
    <rdfs:subClassOf rdf:resource="#Explicit_Constraints"/>
</owl:Class>

<owl:Class rdf:about="#Constraints">
    <rdfs:subClassOf rdf:resource="#Relational_Database"/>
</owl:Class>

<owl:Class rdf:about="#DB_Languages">
    <rdfs:subClassOf rdf:resource="#Databe_Ontology"/>
</owl:Class>

<owl:Class rdf:about="#DB_Operations">
    <rdfs:subClassOf rdf:resource="#Relational_Algebra_Operations"/>
</owl:Class>

<owl:Class rdf:about="#Data_Definition_Language">
    <rdfs:subClassOf rdf:resource="#DB_Languages"/>
</owl:Class>

<owl:Class rdf:about="#Data_Manipulation_Language">
    <rdfs:subClassOf rdf:resource="#DB_Languages"/>
</owl:Class>

<owl:Class rdf:about="#Data_Type">
    <rdfs:subClassOf rdf:resource="#Relational_Database"/>
</owl:Class>

<owl:Class rdf:about="#Database_Models">
    <rdfs:subClassOf rdf:resource="#Databe_Ontology"/>
</owl:Class>

<owl:Class rdf:about="#Databe_Ontology">
    <rdfs:subClassOf rdf:resource="&owl;Thing"/>
</owl:Class>

<owl:Class rdf:about="#Date">
    <rdfs:subClassOf rdf:resource="#Data_Type"/>
</owl:Class>

<owl:Class rdf:about="#Degree_Of_Relation">
    <rdfs:subClassOf rdf:resource="#Relational_Database"/>
</owl:Class>

<owl:Class rdf:about="#Delete">
    <rdfs:subClassOf rdf:resource="#UpdateOperations"/>
</owl:Class>

<owl:Class rdf:about="#Division">
    <rdfs:subClassOf rdf:resource="#DB_Operations"/>
</owl:Class>

<owl:Class rdf:about="#Domain">
    <rdfs:subClassOf rdf:resource="#Relational_Database"/>
</owl:Class>

<owl:Class rdf:about="#Domain_Constraint">
    <rdfs:subClassOf rdf:resource="#Explicit_Constraints"/>
</owl:Class>

<owl:Class rdf:about="#Domain_Relational_Calculus">
    <rdfs:subClassOf rdf:resource="#Relational_Calculus"/>
```

```
</owl:Class>

<owl:Class rdf:about="#ER_Model">
    <rdfs:subClassOf rdf:resource="#Object_Based_Logical_Model"/>
</owl:Class>

<owl:Class rdf:about="#Entity_Integrity_Constraint">
    <rdfs:subClassOf rdf:resource="#Integrity_Constraints"/>
</owl:Class>

<owl:Class rdf:about="#Existential_Quantifier">
    <rdfs:subClassOf rdf:resource="#Quantifiers"/>
</owl:Class>

<owl:Class rdf:about="#Explicit_Constraints">
    <rdfs:subClassOf rdf:resource="#Constraints"/>
</owl:Class>

<owl:Class rdf:about="#Formula/Condition">
    <rdfs:subClassOf rdf:resource="#Relational_Database"/>
</owl:Class>

<owl:Class rdf:about="#Frame_Memory">
    <rdfs:subClassOf rdf:resource="#Physical_Data_Model"/>
</owl:Class>

<owl:Class rdf:about="#Full_Outer_Join">
    <rdfs:subClassOf rdf:resource="#Outer_Join"/>
</owl:Class>

<owl:Class rdf:about="#Generalized_Projection">
    <rdfs:subClassOf rdf:resource="#Additional_Relational_Operations"/>
</owl:Class>

<owl:Class rdf:about="#Grouping">
    <rdfs:subClassOf rdf:resource="#Additional_Relational_Operations"/>
</owl:Class>

<owl:Class rdf:about="#Hierarchical_Model">
    <rdfs:subClassOf rdf:resource="#Record_Based_Logical_Model"/>
</owl:Class>

<owl:Class rdf:about="#Higher_Level_DML">
    <rdfs:subClassOf rdf:resource="#Data_Manipulation_Language"/>
</owl:Class>

<owl:Class rdf:about="#INTERSECTION">
    <rdfs:subClassOf rdf:resource="#Set_Operations"/>
</owl:Class>

<owl:Class rdf:about="#Implicit_Constraints">
    <rdfs:subClassOf rdf:resource="#Constraints"/>
</owl:Class>

<owl:Class rdf:about="#Inner_Join/Equi_Join">
    <rdfs:subClassOf rdf:resource="#Theta_Join"/>
</owl:Class>

<owl:Class rdf:about="#Insert">
    <rdfs:subClassOf rdf:resource="#UpdateOperations"/>
</owl:Class>

<owl:Class rdf:about="#Integrity_Constraints">
    <rdfs:subClassOf rdf:resource="#Explicit_Constraints"/>
</owl:Class>

<owl:Class rdf:about="#Interpretation_Of_Relation">
    <rdfs:subClassOf rdf:resource="#Implicit_Constraints"/>
</owl:Class>

<owl:Class rdf:about="#Join">
    <rdfs:subClassOf rdf:resource="#DB_Operations"/>
</owl:Class>

<owl:Class rdf:about="#Key">
    <rdfs:subClassOf rdf:resource="#Relational_Database"/>
</owl:Class>

<owl:Class rdf:about="#Key_Constraint">
    <rdfs:subClassOf rdf:resource="#Explicit_Constraints"/>
</owl:Class>

<owl:Class rdf:about="#Left_Outer_Join">
```

```
    <rdfs:subClassOf rdf:resource="#Outer_Join"/>
</owl:Class>

<owl:Class rdf:about="#Logical_Operators">
    <rdfs:subClassOf rdf:resource="#Relational_Database"/>
</owl:Class>

<owl:Class rdf:about="#Lower_Level_DML">
    <rdfs:subClassOf rdf:resource="#Data_Manipulation_Language"/>
</owl:Class>

<owl:Class rdf:about="#Modify/Update">
    <rdfs:subClassOf rdf:resource="#UpdateOperations"/>
</owl:Class>

<owl:Class rdf:about="#NOT">
    <rdfs:subClassOf rdf:resource="#Logical_Operators"/>
</owl:Class>

<owl:Class rdf:about="#NULL_Value">
    <rdfs:subClassOf rdf:resource="#Value"/>
</owl:Class>

<owl:Class rdf:about="#Natural_Join">
    <rdfs:subClassOf rdf:resource="#Inner_Join/Equi_Join"/>
</owl:Class>

<owl:Class rdf:about="#Network_Model">
    <rdfs:subClassOf rdf:resource="#Record_Based_Logical_Model"/>
</owl:Class>

<owl:Class rdf:about="#Numeric">
    <rdfs:subClassOf rdf:resource="#Data_Type"/>
</owl:Class>

<owl:Class rdf:about="#OR">
    <rdfs:subClassOf rdf:resource="#Logical_Operators"/>
</owl:Class>

<owl:Class rdf:about="#Object_Based_Logical_Model">
    <rdfs:subClassOf rdf:resource="#Database_Models"/>
</owl:Class>

<owl:Class rdf:about="#Object_Oriented_Model">
    <rdfs:subClassOf rdf:resource="#Object_Based_Logical_Model"/>
</owl:Class>

<owl:Class rdf:about="#Ordering_Of_Tuples">
    <rdfs:subClassOf rdf:resource="#Implicit_Constraints"/>
</owl:Class>

<owl:Class rdf:about="#Ordering_Of_Values">
    <rdfs:subClassOf rdf:resource="#Implicit_Constraints"/>
</owl:Class>

<owl:Class rdf:about="#Outer_Join">
    <rdfs:subClassOf rdf:resource="#Theta_Join"/>
</owl:Class>

<owl:Class rdf:about="#Physical_Data_Model">
    <rdfs:subClassOf rdf:resource="#Database_Models"/>
</owl:Class>

<owl:Class rdf:about="#Primary_Key">
    <rdfs:subClassOf rdf:resource="#Key"/>
</owl:Class>

<owl:Class rdf:about="#Project">
    <rdfs:subClassOf rdf:resource="#DB_Operations"/>
</owl:Class>

<owl:Class rdf:about="#Quantifiers">
    <rdfs:subClassOf rdf:resource="#Relational_Database"/>
</owl:Class>

<owl:Class rdf:about="#Range">
    <rdfs:subClassOf rdf:resource="#Relational_Database"/>
</owl:Class>

<owl:Class rdf:about="#Record_Based_Logical_Model">
    <rdfs:subClassOf rdf:resource="#Database_Models"/>
</owl:Class>
```

```
<owl:Class rdf:about="#Recursive_Closure">
    <rdfs:subClassOf rdf:resource="#Additional_Relational_Operations"/>
</owl:Class>

<owl:Class rdf:about="#Referential_Integrity_Constraint">
    <rdfs:subClassOf rdf:resource="#Integrity_Constraints"/>
</owl:Class>

<owl:Class rdf:about="#Relation_Schema">
    <rdfs:subClassOf rdf:resource="#Schema"/>
</owl:Class>

<owl:Class rdf:about="#Relational_Algebra_Operations">
    <rdfs:subClassOf rdf:resource="#Relational_Database"/>
</owl:Class>

<owl:Class rdf:about="#Relational_Calculus">
    <rdfs:subClassOf rdf:resource="#Relational_Database"/>
</owl:Class>

<owl:Class rdf:about="#Relational_Database">
    <rdfs:subClassOf rdf:resource="#Datebe_Ontology"/>
</owl:Class>

<owl:Class rdf:about="#Relational_Database_Schema">
    <rdfs:subClassOf rdf:resource="#Schema"/>
</owl:Class>

<owl:Class rdf:about="#Relational_Model">
    <rdfs:subClassOf rdf:resource="#Record_Based_Logical_Model"/>
</owl:Class>

<owl:Class rdf:about="#Rename">
    <rdfs:subClassOf rdf:resource="#DB_Operations"/>
</owl:Class>

<owl:Class rdf:about="#Right_Outer_Join">
    <rdfs:subClassOf rdf:resource="#Outer_Join"/>
</owl:Class>

<owl:Class rdf:about="#Rows">
    <rdfs:subClassOf rdf:resource="#Relational_Database"/>
</owl:Class>

<owl:Class rdf:about="#SET_DIFFERENCE">
    <rdfs:subClassOf rdf:resource="#Set_Operations"/>
</owl:Class>

<owl:Class rdf:about="#SQL_Alter">
    <rdfs:subClassOf rdf:resource="#SQL_DDL_Commands"/>
</owl:Class>

<owl:Class rdf:about="#SQL_Commands">
    <rdfs:subClassOf rdf:resource="#Structure_Query_Language"/>
</owl:Class>

<owl:Class rdf:about="#SQL_Create">
    <rdfs:subClassOf rdf:resource="#SQL_DDL_Commands"/>
</owl:Class>

<owl:Class rdf:about="#SQL_DDL_Commands">
    <rdfs:subClassOf rdf:resource="#SQL_Commands"/>
</owl:Class>

<owl:Class rdf:about="#SQL_DML_Commands">
    <rdfs:subClassOf rdf:resource="#SQL_Commands"/>
</owl:Class>

<owl:Class rdf:about="#SQL_Delete">
    <rdfs:subClassOf rdf:resource="#SQL_DML_Commands"/>
</owl:Class>

<owl:Class rdf:about="#SQL_Drop">
    <rdfs:subClassOf rdf:resource="#SQL_DDL_Commands"/>
</owl:Class>

<owl:Class rdf:about="#SQL_Insert">
    <rdfs:subClassOf rdf:resource="#SQL_DML_Commands"/>
</owl:Class>

<owl:Class rdf:about="#SQL_Select">
    <rdfs:subClassOf rdf:resource="#SQL_DML_Commands"/>
</owl:Class>
```

```
    <owl:Class rdf:about="#SQL_Update">
        <rdfs:subClassOf rdf:resource="#SQL_DML_Commands"/>
    </owl:Class>

    <owl:Class rdf:about="#Schema">
        <rdfs:subClassOf rdf:resource="#Relational_Database"/>
    </owl:Class>

    <owl:Class rdf:about="#Select">
        <rdfs:subClassOf rdf:resource="#DB_Operations"/>
    </owl:Class>

    <owl:Class rdf:about="#Semantic_Constraints">
        <rdfs:subClassOf rdf:resource="#Constraints"/>
    </owl:Class>

    <owl:Class rdf:about="#Semantic_Data_Model">
        <rdfs:subClassOf rdf:resource="#Object_Based_Logical_Model"/>
    </owl:Class>

    <owl:Class rdf:about="#Semantic_Integrity_Constraints">
        <rdfs:subClassOf rdf:resource="#Integrity_Constraints"/>
    </owl:Class>

    <owl:Class rdf:about="#Set_Operations">
        <rdfs:subClassOf rdf:resource="#Relational_Algebra_Operations"/>
    </owl:Class>

    <owl:Class rdf:about="#Storage_Definition_Language">
        <rdfs:subClassOf rdf:resource="#DB_Languages"/>
    </owl:Class>

    <owl:Class rdf:about="#Structure_Query_Language">
        <rdfs:subClassOf rdf:resource="#Relational_Database"/>
    </owl:Class>

    <owl:Class rdf:about="#Table">
        <rdfs:subClassOf rdf:resource="#Relational_Database"/>
    </owl:Class>

    <owl:Class rdf:about="#Theta_Join">
        <rdfs:subClassOf rdf:resource="#Join"/>
    </owl:Class>

    <owl:Class rdf:about="#Time">
        <rdfs:subClassOf rdf:resource="#Data_Type"/>
    </owl:Class>

    <owl:Class rdf:about="#Tuple_Relational_Calculus">
        <rdfs:subClassOf rdf:resource="#Relational_Calculus"/>
    </owl:Class>

    <owl:Class rdf:about="#UNION">
        <rdfs:subClassOf rdf:resource="#Set_Operations"/>
    </owl:Class>

    <owl:Class rdf:about="#Unifying_Model">
        <rdfs:subClassOf rdf:resource="#Physical_Data_Model"/>
    </owl:Class>

    <owl:Class rdf:about="#Universal_Quantifier">
        <rdfs:subClassOf rdf:resource="#Quantifiers"/>
    </owl:Class>

    <owl:Class rdf:about="#UpdateOperations">
        <rdfs:subClassOf rdf:resource="#Relational_Database"/>
    </owl:Class>

    <owl:Class rdf:about="#Value">
        <rdfs:subClassOf rdf:resource="#Relational_Database"/>
    </owl:Class>

    <owl:Class rdf:about="#Values&amp;NullValue_In_Tuple">
        <rdfs:subClassOf rdf:resource="#Implicit_Constraints"/>
    </owl:Class>

    <owl:Class rdf:about="#View_Definition_Language">
        <rdfs:subClassOf rdf:resource="#DB_Languages"/>
    </owl:Class>

    <owl:Class rdf:about="&owl;Thing"/>
</rdf:RDF>
```

Appendix F

Software Application Ontology

In this appendix, we give the current version of software application case study ontology (c.f. Section 4.1.1).

```
<?xml version="1.0"?>
<!DOCTYPE Ontology [
    <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
    <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
    <!ENTITY owl2xml "http://www.w3.org/2006/12/owl2-xml#" >
    <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
    <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
    <!ENTITY Ontology1242290150984 "http://www.semanticweb.org/ontologies/2009/4/14/Ontology1242290150984.owl#" >
]>
<Ontology xmlns="http://www.w3.org/2006/12/owl2-xml#"
     xml:base="http://www.w3.org/2006/12/owl2-xml#"
     xmlns:owl2xml="http://www.w3.org/2006/12/owl2-xml#"
xmlns:Ontology1242290150984="http://www.semanticweb.org/ontologies/2009/4/14/Ontology1242290150984.owl#"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:owl="http://www.w3.org/2002/07/owl#"
     URI="http://www.semanticweb.org/ontologies/2009/4/14/Ontology1242290150984.owl">
    <SubClassOf>
        <Class URI="&Ontology1242290150984;Administrator"/>
        <Class URI="&Ontology1242290150984;User"/>
    </SubClassOf>
    <Declaration>
        <Class URI="&Ontology1242290150984;Administrator"/>
    </Declaration>
    <SubClassOf>
        <Class URI="&Ontology1242290150984;Archiving"/>
        <Class URI="&Ontology1242290150984;Task"/>
    </SubClassOf>
    <Declaration>
        <Class URI="&Ontology1242290150984;Archiving"/>
    </Declaration>
    <SubClassOf>
        <Class URI="&Ontology1242290150984;Assigning"/>
        <Class URI="&Ontology1242290150984;Task"/>
    </SubClassOf>
    <Declaration>
        <Class URI="&Ontology1242290150984;Assigning"/>
    </Declaration>
    <SubClassOf>
        <Class URI="&Ontology1242290150984;Book_Info"/>
        <Class URI="&Ontology1242290150984;Help_File"/>
    </SubClassOf>
    <Declaration>
        <Class URI="&Ontology1242290150984;Book_Info"/>
    </Declaration>
```

```
<SubClassOf>
    <Class URI="&Ontology1242290150984;Building"/>
    <Class URI="&Ontology1242290150984;Task"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Building"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Button"/>
    <Class URI="&Ontology1242290150984;GUI"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Button"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Canceling"/>
    <Class URI="&Ontology1242290150984;Task"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Canceling"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Case"/>
    <Class URI="&Ontology1242290150984;Topic_Concept"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Case"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Chapter"/>
    <Class URI="&Ontology1242290150984;Help_File"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Chapter"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Checking"/>
    <Class URI="&Ontology1242290150984;Task"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Checking"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Closing"/>
    <Class URI="&Ontology1242290150984;Task"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Closing"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;CommandLine"/>
    <Class URI="&Ontology1242290150984;Software_Feature"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;CommandLine"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Comment"/>
    <Class URI="&Ontology1242290150984;Topic_Concept"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Comment"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Configuring"/>
    <Class URI="&Ontology1242290150984;Task"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Configuring"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Copying"/>
    <Class URI="&Ontology1242290150984;Task"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Copying"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Creating"/>
    <Class URI="&Ontology1242290150984;Task"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Creating"/>
```

```
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Customer"/>
    <Class URI="&Ontology1242290150984;User"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Customer"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Customizing"/>
    <Class URI="&Ontology1242290150984;Task"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Customizing"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Data"/>
    <Class URI="&Ontology1242290150984;Topic_Concept"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Data"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Database"/>
    <Class URI="&Ontology1242290150984;Software_Feature"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Database"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Deleting"/>
    <Class URI="&Ontology1242290150984;Task"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Deleting"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Deligate"/>
    <Class URI="&Ontology1242290150984;User"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Deligate"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Deveolper"/>
    <Class URI="&Ontology1242290150984;User"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Deveolper"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Downloading"/>
    <Class URI="&Ontology1242290150984;Task"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Downloading"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Editing"/>
    <Class URI="&Ontology1242290150984;Task"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Editing"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Employee"/>
    <Class URI="&Ontology1242290150984;User"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Employee"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;End_User"/>
    <Class URI="&Ontology1242290150984;User"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;End_User"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Exporting"/>
    <Class URI="&Ontology1242290150984;Task"/>
</SubClassOf>
<Declaration>
```

```
    <Class URI="&Ontology1242290150984;Exporting"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;File"/>
    <Class URI="&Ontology1242290150984;Software_Feature"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;File"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Folders"/>
    <Class URI="&Ontology1242290150984;Topic_Concept"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Folders"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;GUI"/>
    <Class URI="&Ontology1242290150984;Software_Feature"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;GUI"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Guidelines"/>
    <Class URI="&Ontology1242290150984;Topic_Concept"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Guidelines"/>
</Declaration>
<Declaration>
    <Class URI="&Ontology1242290150984;Help_File"/>
</Declaration>
<Declaration>
    <Class URI="&Ontology1242290150984;Help_and_query_Structure"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Hot_Words"/>
    <Class URI="&Ontology1242290150984;Topic_Concept"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Hot_Words"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Importing"/>
    <Class URI="&Ontology1242290150984;Task"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Importing"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Index"/>
    <Class URI="&Ontology1242290150984;Help_File"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Index"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Items"/>
    <Class URI="&Ontology1242290150984;Topic_Concept"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Items"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Marks_and_Schemes"/>
    <Class URI="&Ontology1242290150984;Topic_Concept"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Marks_and_Schemes"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Menu"/>
    <Class URI="&Ontology1242290150984;GUI"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Menu"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Messages"/>
    <Class URI="&Ontology1242290150984;Topic_Concept"/>
</SubClassOf>
<Declaration>
```

```
    <Class URI="&Ontology1242290150984;Messages"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Opening"/>
    <Class URI="&Ontology1242290150984;Task"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Opening"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Para"/>
    <Class URI="&Ontology1242290150984;Chapter"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Para"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Pausing"/>
    <Class URI="&Ontology1242290150984;Task"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Pausing"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Permissions"/>
    <Class URI="&Ontology1242290150984;Topic_Concept"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Permissions"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Procedure"/>
    <Class URI="&Ontology1242290150984;Chapter"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Procedure"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Reference"/>
    <Class URI="&Ontology1242290150984;Help_and_query_Structure"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Reference"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Removing"/>
    <Class URI="&Ontology1242290150984;Task"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Removing"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Renaming"/>
    <Class URI="&Ontology1242290150984;Task"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Renaming"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Reports"/>
    <Class URI="&Ontology1242290150984;Topic_Concept"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Reports"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Responsibilities"/>
    <Class URI="&Ontology1242290150984;Topic_Concept"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Responsibilities"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Restoring"/>
    <Class URI="&Ontology1242290150984;Task"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Restoring"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Resuming"/>
    <Class URI="&Ontology1242290150984;Task"/>
</SubClassOf>
```

```
<Declaration>
    <Class URI="&Ontology1242290150984;Resuming"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Reviewer"/>
    <Class URI="&Ontology1242290150984;User"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Reviewer"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Roles"/>
    <Class URI="&Ontology1242290150984;Topic_Concept"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Roles"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Searching"/>
    <Class URI="&Ontology1242290150984;Task"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Searching"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Section"/>
    <Class URI="&Ontology1242290150984;Chapter"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Section"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Sharing"/>
    <Class URI="&Ontology1242290150984;Task"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Sharing"/>
</Declaration>
<Declaration>
    <Class URI="&Ontology1242290150984;Software_Application"/>
</Declaration>
<Declaration>
    <Class URI="&Ontology1242290150984;Software_Feature"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Steps"/>
    <Class URI="&Ontology1242290150984;Procedure"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Steps"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Subtitle"/>
    <Class URI="&Ontology1242290150984;Help_File"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Subtitle"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Supervisor"/>
    <Class URI="&Ontology1242290150984;User"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Supervisor"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Task"/>
    <Class URI="&Ontology1242290150984;Help_and_query_Structure"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Task"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Title"/>
    <Class URI="&Ontology1242290150984;Help_File"/>
</SubClassOf>
<Declaration>
    <Class URI="&Ontology1242290150984;Title"/>
</Declaration>
<SubClassOf>
    <Class URI="&Ontology1242290150984;Topic_Concept"/>
    <Class URI="&Ontology1242290150984;Help_and_query_Structure"/>
</SubClassOf>
```

```
    <Declaration>
        <Class URI="&Ontology1242290150984;Topic_Concept"/>
    </Declaration>
    <SubClassOf>
        <Class URI="&Ontology1242290150984;Turning_Off"/>
        <Class URI="&Ontology1242290150984;Task"/>
    </SubClassOf>
    <Declaration>
        <Class URI="&Ontology1242290150984;Turning_Off"/>
    </Declaration>
    <SubClassOf>
        <Class URI="&Ontology1242290150984;Turning_On"/>
        <Class URI="&Ontology1242290150984;Task"/>
    </SubClassOf>
    <Declaration>
        <Class URI="&Ontology1242290150984;Turning_On"/>
    </Declaration>
    <Declaration>
        <Class URI="&Ontology1242290150984;User"/>
    </Declaration>
    <SubClassOf>
        <Class URI="&Ontology1242290150984;Viewing"/>
        <Class URI="&Ontology1242290150984;Task"/>
    </SubClassOf>
    <Declaration>
        <Class URI="&Ontology1242290150984;Viewing"/>
    </Declaration>
    <SubClassOf>
        <Class URI="&Ontology1242290150984;Window"/>
        <Class URI="&Ontology1242290150984;GUI"/>
    </SubClassOf>
    <Declaration>
        <Class URI="&Ontology1242290150984;Window"/>
    </Declaration>
    <SubClassOf>
        <Class URI="&Ontology1242290150984;synchronizing"/>
        <Class URI="&Ontology1242290150984;Task"/>
    </SubClassOf>
    <Declaration>
        <Class URI="&Ontology1242290150984;synchronizing"/>
    </Declaration>
</Ontology>
```

Appendix G

Composite-level Evolution Strategies

In this appendix, we give a list of proposed composite-level evolution strategies for composite change operators that can be utilized to perform a composite change according to the needs of a user. As different atomic change operators can be combined together to define new composite changes, the proposed evolution strategies are customizable to meet one's own needs.

`Pull up class (X, C)`

Definition: Attach a class (as a child) to the parent(s) of its previous parent(s).

Structural impact: The class `X` is pulled up in the class hierarchy and became a sibling of its previous parent class `C`.

Semantic impact: Instances of `X` are not instance of `C` anymore (inference).

Resolution point: Given, a class `X` is disjoint to class `C`,
$type(\texttt{I}) = \texttt{X} \Rightarrow \neg\ type(\texttt{I}) = \texttt{C}$ (and vice versa)

The composite change may make ontology inconsistent if a predefined disjointness exists among the siblings of the class `C` (i.e. `X` and `C` become disjoint classes in version 2). In such case, any instantiation of a property `P` whose domain/range consist class `C`, by any instance of class `X`, is not valid anymore. If such instantiations of property `P` exists, ontology will become inconsistent.

Evolution Strategies:

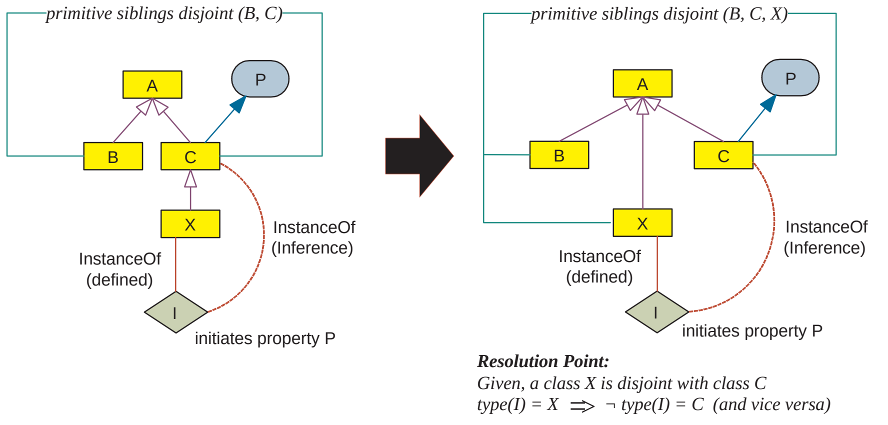

primitive siblings disjoint (B, C)
A
P
B
C
X
InstanceOf (defined)
InstanceOf (Inference)
I
initiates property P
primitive siblings disjoint (B, C, X)
A
P
B
C
X
InstanceOf (defined)
InstanceOf (Inference)
I
initiates property P
Resolution Point:
Given, a class X is disjoint with class C
type(I) = X ⇒ ¬ type(I) = C (and vice versa)

Figure G.1: Pull up class (X, C)

If the property `P` does not fit for the class `C` anymore, user can delete the instantiation of the property `P` for the instances of `C`, OR

If the instances of class `X` can still be instances of `C`, user can delete the disjointness between the classes `C` and `X`, OR

If instances of class `X` cannot be considered as instances of class `C` anymore however the property `P` is still valid for the instances of class `C`, in such case user can explicitly add class `X` as domain/range of the property `P` i.e. `domainOf(P) = C or X`.

`Pull down class (A, B)`

Definition: Attach a class (as a child) to its previous sibling class(s).

Structural impact: The class `A` is pulled down in the class hierarchy and became a child of its previous sibling class `B`.

Semantic impact: Instances of class `A` are instances of class `B` as well (inference).

Resolution point: Given two classes `A` and `B`,
$subclassOf(A, B) \Rightarrow \neg\ disjointClasses(A, B)$

Domain ontology will become inconsistent if the classes `A` and `B` were disjoint to each other before the execution of the composite change. In such case, instances of class `A` cannot be referred as instances of class `B`.

Evolution Strategies:

In order to resolve the resolution point, user may delete the disjointness between the classes `A` and `B`.

`Split class (X, (C1, C2))`

Definition: Split a class into two (or more) classes.

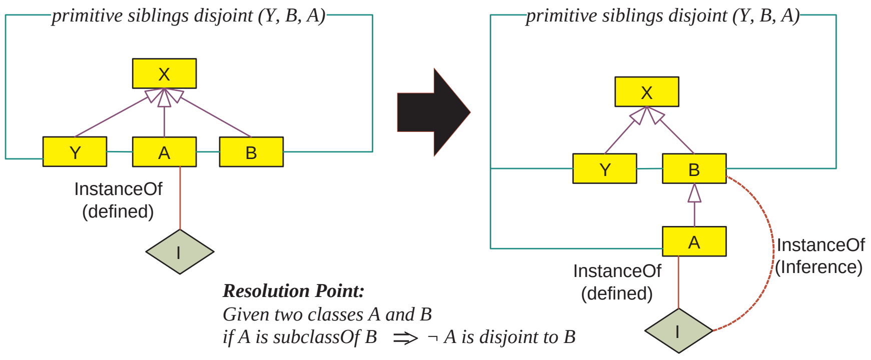

primitive siblings disjoint (Y, B, A)
X
Y
A
B
InstanceOf (defined)
I
primitive siblings disjoint (Y, B, A)
X
Y
B
A
InstanceOf (Inference)
InstanceOf (defined)
I
Resolution Point:
Given two classes A and B
if A is subclassOf B ⇒ ¬ A is disjoint to B

Figure G.2: Pull down class (A, B)

Structural impact: Class `X` is replaced by two sibling classes `C1` and `C2`.

Semantic impact: Class `X` is split into two sibling classes and the roles (relationships) of class `X` (that had to be inherited by the newly added classes) become unattached.

Resolution point: The newly added sibling classes `C1` and `C2` inherit relationships from the split class `X`. Thus, the deleted relationships (axioms) of class `X` must be preserved and re-attached to the newly added classes.

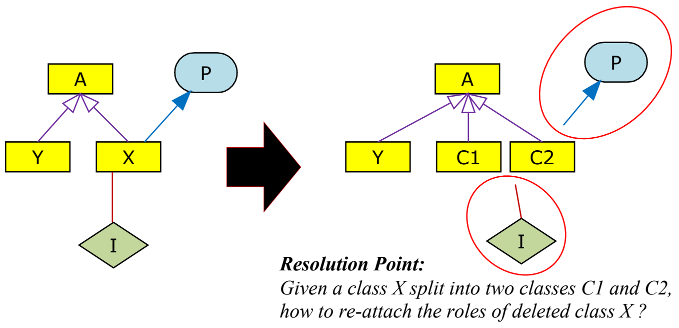

A
P
Y
X
I
A
P
Y
C1
C2
I
Resolution Point:
Given a class X split into two classes C1 and C2,
how to re-attach the roles of deleted class X ?

Figure G.3: Split class (X, (C1, C2))

Evolution Strategies: In order to resolve the resolution point, user can either

distribute the deleted roles of class `X` among the newly added replacement classes, OR

re-attach the roles to one of the newly added replacement class, OR

re-attach roles to all the newly added replacement classes, OR

do nothing.

Note, a role can be re-attached to two or more newly added classes using "or" or "and" property. For example, if a class "`person`" is split into two sibling classes "`male`" and "`female`", we can re-attach property "`hasAge`" as `domainOf(hasAge) = male or female`. That means, if an individual instantiate the property "`hasAge`", the individual is either a male or a female. If a class "`ResearchStudent`" is split into two sibling classes "`Student`" and "`Researcher`", we can re-attach property “`hasAuthor`” for PhD and MS by research students as `rangeOf(hasAuthor) = Student and Researcher`. That means, if an individual instantiate the property “`hasAuthor`”, the individual is a student as well as a researcher.

`Merge classes ((C1, C2), X)`

Definition: Merge two (or more) classes into one single class.

Structural impact: Classes `C1` and `C2` are replaced by one single class `X`.

Semantic impact: Classes `C1` and `C2` are merged into one single class `X` and the roles (relationships) of classes `C1` and `C2` (that had to be inherited by the class `X`) become unattached.

Resolution point: The newly added class `X` inherits relationships from the classes `C1` and `C2`. Thus, the deleted relationships (axioms) of classes `C1` and `C2` must be preserved and re-attached to the newly added class `X`.

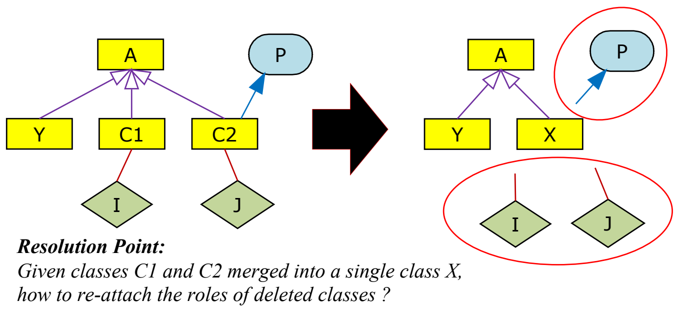

A
P
Y
C1
C2
I
J
A
P
Y
X
I
J
Resolution Point:
Given classes C1 and C2 merged into a single class X,
how to re-attach the roles of deleted classes ?

Figure G.4: Merge classes ((C1, C2), X)

Evolution Strategies: In order to resolve the resolution point, user can either

aggregate all the deleted roles of classes `C1` and `C2` to the replacement class `X`, OR

aggregate selected roles classes `C1` and `C2` to the replacement class `X`, OR

do nothing.

`Pull up property (P, A, B)`

Definition: Pull a property higher in the class hierarchy and attach it to a parent class of its previous domain/range class.

Structural impact: The property `P` is attached to the parent class `A` of its earlier domain/range class `B`.

Semantic impact: Earlier, the individuals that instantiate property `P`, were inferred as instances of class `B` as well as instances of class `A` (due to subclass hierarchy). After replacing the domain/range of the property `P` (i.e. class `B`) by the parent class `A`, the individuals will be inferred as instances of class `A` but not of class `B`.

Resolution point: Earlier (inferred) instances of class `B` (through instantiation of property `P`) are not valid anymore.

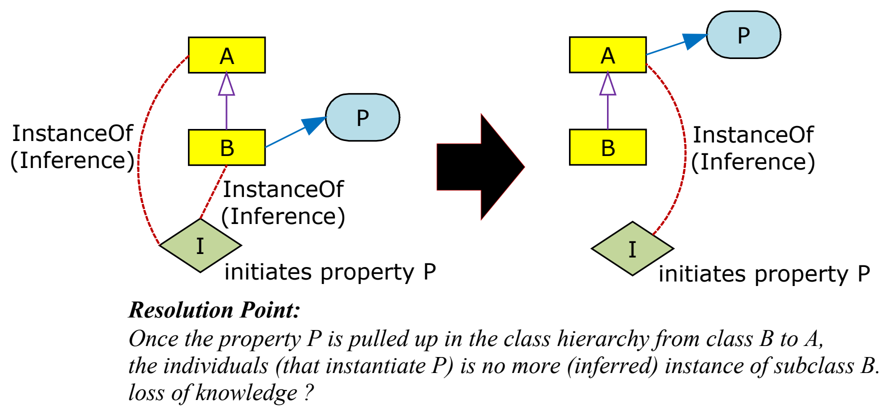

A
B
P
InstanceOf (Inference)
InstanceOf (Inference)
I
initiates property P
P
A
B
InstanceOf (Inference)
I
initiates property P
Resolution Point:
Once the property P is pulled up in the class hierarchy from class B to A,
the individuals (that instantiate P) is no more (inferred) instance of subclass B.
loss of knowledge ?

Figure G.5: Pull up property(P, A, B)

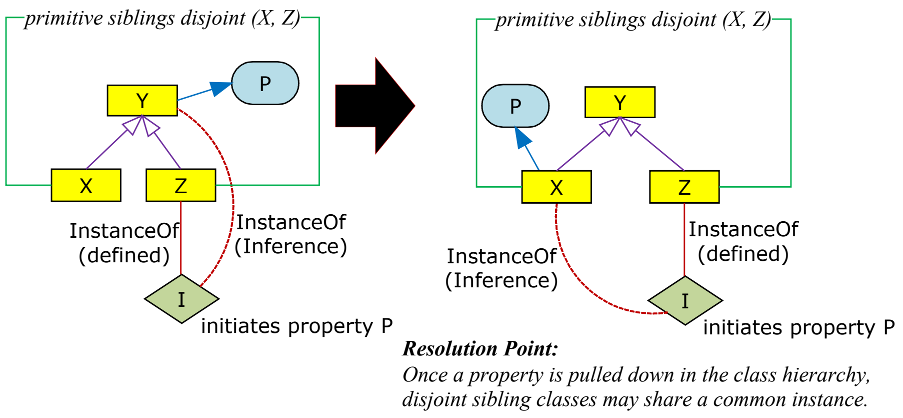

primitive siblings disjoint (X, Z)
Y
P
X
Z
InstanceOf (defined)
InstanceOf (Inference)
I
initiates property P
primitive siblings disjoint (X, Z)
P
Y
X
Z
InstanceOf (Inference)
InstanceOf (defined)
I
initiates property P
Resolution Point:
Once a property is pulled down in the class hierarchy,
disjoint sibling classes may share a common instance.

Figure G.6: Pull down property(P, X, Y)

Evolution Strategies:

In cases, where a user like to make sure that there is no loss of previous knowledge, i.e. all earlier inferred instances of child class `B` may still be recognized, user can assert the instances explicitly as defined instances of class `B`.

`Pull down property (P, X, Y)`

Definition: Pull a property down in the class hierarchy and attach it to a child class of its previous domain/range class.

Structural impact: The property `P` is attached to the child class `X` of its earlier domain/range class `Y`.

Semantic impact: Earlier, the individuals that instantiate property `P` could be inferred as instances of class `Y` only. After replacing the domain/range of the property `P` (i.e. class `Y`) by the child class `X`, the individuals will be inferred as instances of class `X` as well as of class `Y` (due to subclass hierarchy).

Resolution point: Given,

$type(\mathtt{I}) = \mathtt{Z} \wedge siblingClasses(\mathtt{X}, \mathtt{Z}) \wedge disjointClasses\ (\mathtt{X}, \mathtt{Z})$

if, `I` *instantiates* `P` $\Rightarrow type(\mathtt{I}) = \mathtt{X}$.

This unsatisfied disjointness rule (i.e. two disjoint classes cannot share a common instance).

Evolution Strategies: In order to resolve such resolution point, a user can

remove the disjointness between class `X` and its sibling classes. In such case, an (inferred/defined) individual of `X`'s sibling class, that instantiate property `P`, will also be inferred as instance of class `X`. OR

where disjointness between the class `X` and its sibling classes is desired, a user can delete the instantiation of the property `P` by the instances of disjoint sibling classes of class `X`. In such case, the (inferred/defined) individuals of `X`'s sibling classes will no longer be inferred as instances of class `X`.

Appendix H

Results of Composite change pattern detection algorithms

We performed a user case study in order to evaluate the composite change detection algorithm. The algorithm has been implemented using Java language. In this appendix, we give the result list of the case study. The results have be compared with the manual approach to verify the algorithm's performance.

```
Split class - Result (Candidate):

59:Add class (MSTaughtStudent)
60:Add subClassAxiom (MSTaughtStudent, Student)
63:Add class (MSByResearchStudent)
64:Add subClassAxiom (MSByResearchStudent, Student)
67:Delete subClassAxiom (MSStudent, Student)
68:Delete class (MSStudent)
69:Add class (TaughtStudent)
70:Add subClassAxiom (TaughtStudent, Student)
75:Add class (ResearchStudent)
76:Add subClassAxiom (ResearchStudent, Student)

Split class - Result (Roles Distributed):

59:Add class (MSTaughtStudent)
60:Add subClassAxiom (MSTaughtStudent, Student)
61:Delete classAssertionAxiom (Zubair, MSStudent)
62:Add classAssertionAxiom (Zubair, MSTaughtStudent)
63:Add class (MSByResearchStudent)
64:Add subClassAxiom (MSByResearchStudent, Student)
```

```
65:Add classAssertionAxiom (Robert, MSByResearchStudent)
66:Delete classAssertionAxiom (Robert, MSStudent)
67:Delete subClassAxiom (MSStudent, Student)
68:Delete class (MSStudent)

Add specialise class - Result:

106:Add class (Publication)
107:Add subClassAxiom (Publication, Content)
110:Add subClassAxiom (Article, Publication)
111:Delete subClassAxiom (Article, Content)

131:Add class (AcademicOrganisation)
132:Add subClassAxiom (AcademicOrganisation, Organisation)
133:Add subClassAxiom (ResearchCentre, AcademicOrganisation)
134:Delete subClassAxiom (ResearchCentre, Organisation)
135:Add subClassAxiom (University, AcademicOrganisation)
136:Delete subClassAxiom (University, Organisation)

Add interior class - Result:

69:Add class (TaughtStudent)
70:Add subClassAxiom (TaughtStudent, Student)
73:Add subClassAxiom (UGStudent, TaughtStudent)
74:Delete subClassAxiom (UGStudent, Student)

75:Add class (ResearchStudent)
76:Add subClassAxiom (ResearchStudent, Student)
77:Add subClassAxiom (PhDStudent, ResearchStudent)
79:Delete subClassAxiom (PhDStudent, Student)

92:Add class (SocialEvent)
93:Add subClassAxiom (SocialEvent, Event)
94:Add subClassAxiom (SocietyEvent, SocialEvent)
96:Delete subClassAxiom (SocietyEvent, Event)

131:Add class (AcademicOrganisation)
132:Add subClassAxiom (AcademicOrganisation, Organisation)
133:Add subClassAxiom (ResearchCentre, AcademicOrganisation)
134:Delete subClassAxiom (ResearchCentre, Organisation)

Group classes - Result:

69:Add class (TaughtStudent)
70:Add subClassAxiom (TaughtStudent, Student)
71:Add subClassAxiom (MSTaughtStudent, TaughtStudent)
72:Delete subClassAxiom (MSTaughtStudent, Student)
73:Add subClassAxiom (UGStudent, TaughtStudent)
74:Delete subClassAxiom (UGStudent, Student)

75:Add class (ResearchStudent)
76:Add subClassAxiom (ResearchStudent, Student)
77:Add subClassAxiom (PhDStudent, ResearchStudent)
```

```
79:Delete subClassAxiom (PhDStudent, Student)
78:Add subClassAxiom (MSByResearchStudent, ResearchStudent)
80:Delete subClassAxiom (MSByResearchStudent, Student)

92:Add class (SocialEvent)
93:Add subClassAxiom (SocialEvent, Event)
94:Add subClassAxiom (SocietyEvent, SocialEvent)
96:Delete subClassAxiom (SocietyEvent, Event)
95:Add subClassAxiom (SportsEvent, SocialEvent)
97:Delete subClassAxiom (SportsEvent, Event)

131:Add class (AcademicOrganisation)
132:Add subClassAxiom (AcademicOrganisation, Organisation)
133:Add subClassAxiom (ResearchCentre, AcademicOrganisation)
134:Delete subClassAxiom (ResearchCentre, Organisation)
135:Add subClassAxiom (University, AcademicOrganisation)
136:Delete subClassAxiom (University, Organisation)

Pull up property - Result:

Pull up object property (Domain):
81:Delete domainOfObjectPropertyAxiom (hasSupervisor, PhDStudent)
82:Add domainOfObjectPropertyAxiom (hasSupervisor, ResearchStudent)

85:Delete domainOfObjectPropertyAxiom (affiliatedTo, PhDStudent)
86:Add domainOfObjectPropertyAxiom (affiliatedTo, ResearchStudent)

Pull down property - Result:

Pull down object property (Domain):
122:Delete domainOfObjectPropertyAxiom (registeredIn, Student)
123:Add domainOfObjectPropertyAxiom (registeredIn, TaughtStudent)
```