myjournal manuscript No.
(will be inserted by the editor)

Ontology Change Management and Identification of Change Patterns

Muhammad Javed · Yalemisew M. Abgaz · Claus Pahl

Received: date / Accepted: date

Abstract 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. 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. In this paper, we propose a fourphase process that covers the operationalization, representation and detection of higherlevel changes in ontology evolution life cycle. We present different levels of change operators based on the granularity and domainspecificity of changes. The first layer is based on generic atomic level change operators, whereas the next two layers are user-defined (generic/domainspecific) change patterns. We introduce layered change logs for the explicit operational representation of ontology changes. We formalised the change log using a graph-based approach. We introduce a technique to identify composite changes that not only assists in formulating ontology change log data in a more concise manner, but also helps in realizing the semantics and intent behind any applied change. Furthermore, we identify frequent change sequences that are applied as a reference in order to discover reusable, often domainspecific and usage-driven change patterns. We describe the pattern identification algorithms and evaluate their performance.

Keywords Customizable Ontology Evolution · Ontology Change Patterns · Pattern-based Ontology Evolution · Change Log Graph · Graph-based Composite Change Detection · Change Pattern Discovery Algorithms

Muhammad Javed
Centre for Next Generation Localisation (CNGL)
School of Computing, Dublin City University,
Dublin 09, Ireland
Tel.: +353-1-700 6912
Fax: +353-1-700 6702
E-mail: mjaved@computing.dcu.ie

Yalemisew M. Abgaz
Centre for Next Generation Localisation (CNGL)
School of Computing, Dublin City University,
Dublin 09, Ireland
Tel.: +353-1-700 6912
Fax: +353-1-700 6702
E-mail: yabgaz@computing.dcu.ie

Claus Pahl
Centre for Next Generation Localisation (CNGL)
School of Computing, Dublin City University,
Dublin 09, Ireland
Tel.: +353-1-700 5620
Fax: +353-1-700 5442
E-mail: claus.pahl@dcu.ie

1 Introduction

Ontologies become essential for knowledge sharing activities, especially in areas such as bioinformatics, semantic web, educational technology systems, indexing and retrieval, etc. Ontologybased content models help to take a step forward from traditional content management systems (CMS) to conceptual knowledge modelling, to meet the requirements of the semantically aware content-based systems (CBS). While some generic ontologies (like upper ontologies) evolve at a slower pace, we have been working with non-public ontologies (formalised using the Web Ontology Language (OWL)) used to annotate content in large-scale information systems. In this context, changes happen on a daily basis, triggered by changes in software, its technical or domain environment. Systematic change becomes here a neces-

sity to enable controlled, accountable and predictable ontology evolution.

Ontology evolution is defined in different ways [13, 16,45]. 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"* [45]. The change operators are the building blocks of ontology evolution. Different layers of change operators have been suggested in past [27,34,44]. However, the identified change operators focus on generic and structural changes lacking domain-specificity and abstraction. These solutions lack adequate support for different levels of granularity at different levels of abstraction. Furthermore, for semantically enhanced information systems, a coherent representation of such ontology changes conveying the semantics of changes is essential.

In this paper, we present a four-phase ontology change management system that covers the operationalisation and the identification of higher-level ontology change patterns (Figure 1). These phases include change operationalisation, change representation, change semantic capturing and change pattern discovery. Few sections of the presented ontology change management system have already been presented in our previously published papers. The explicit distinction between already published and the new work is given below in each section of the phases.

Phase 1 - change operationalisation: We present a layered change operator framework (discussed in Section 4) which consists of three different levels of change operators, based on granularity and domain-specificity. These layered change operations capture the real changes in the selected domains. The first two layers are generic change operators that can be applied on any domain. The changes at a higher level of granularity, which are frequent in a domain and are often neglected by the lower-level compositional change operators addressed in the literature, are captured as domain-specific change operators at level three. The layered change operator framework has been introduced in [22]. However, an underlying structural model has been added to complement the behaviour model.

Phase 2 - change representation: The implementation of the change operator framework is supported through layered change logs (discussed in Section 5). Representing ontology changes as higher-level change operations describes the semantics behind any of the applied change operation. Using higher-level representation of ontology changes, the intent of the applied changes can be explicitly expressed. While the layered change log and a graph-based foundation has been suggested in [24], we substantially expand this here. We exploit the ontology change logs and the pattern recognition techniques to identify the ontology change patterns. To do so, we formalised the ontology change logs by using a graph-based approach (discussed in Section 5.3).

Phase 3-change semantic capturing: The atomic change operations can only represent the addition or deletion of any particular knowledge in the ontology. We utilized the graph-based representation of ontology changes to identify the composite change patterns (discussed in Section 6), which cannot be captured by simple queries on ontology change logs. The composite change operations provide more semantic information of how an ontology changed as well as specific reasons and consequences of operations at a higher level.

Phase 4-change pattern discovery: The discovery of domain-specific change patterns (discussed in Section 7) provides an opportunity to define reusable change patterns that can be implemented in existing knowledge management systems. One of the key benefits of change pattern discovery approach is its integration with an ontology editing framework for pattern-driven ontology evolution. While the graph-based patterns discovery approach along with the algorithms has been presented [23], we provide a detailed evaluation here.

The paper is structured as follows. Related work is discussed in Section 2. In Section 3, we talk about ontology change management in general. Sections 4-7 give detailed description of each phase of the proposed ontology change management system. Experimental results and an evaluation is discussed in Section 8 and we end with some conclusion in Section 9.

2 Related Work

The dynamic nature of knowledge in every conceptual domain requires ontologies to change over time. The reason for change in knowledge can be changes in the domain, the specification, the conceptualization or any combination of them [33]. Some changes are about the introduction of new concepts, removal of outdated concepts and changes in the structures and the description of concepts. A change in an ontology may originate from a domain knowledge expert, a user of the ontology or a change in the application area [32].

Based on the different perspectives of the researchers, there are different solutions provided to handle ontology evolution [4,7,16,17,28,29,32,39,41,44,50]. The

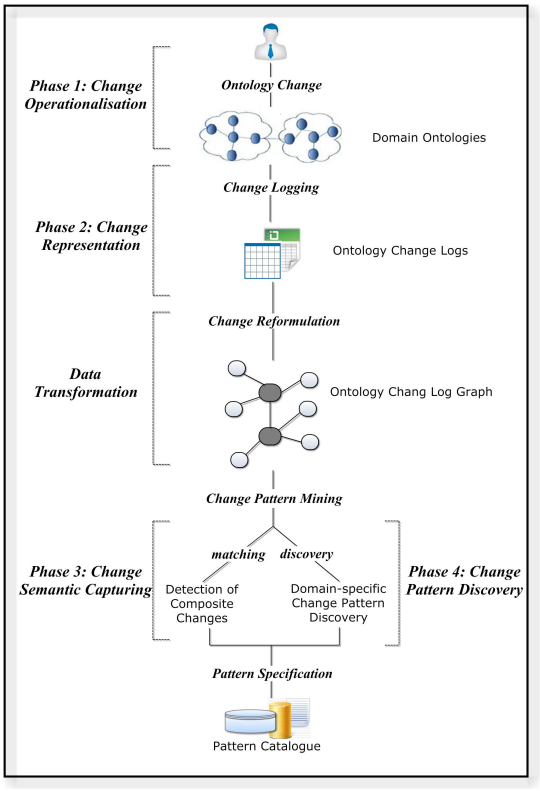

Fig. 1 Proposed Four-phase Ontology Change Management

author in [44] discusses the complexity of the ontology change management process and presents a six phase ontology evolution process. She discusses the representation of generic changes and categorized them into elementary, composite and complex. In contrast to our work, aspects such as granularity, domain-specificity and abstraction are not included. In [7], the authors present a pattern-driven ontology evolution approach with defined participants and execution steps. In [41], the authors present a pattern-driven approach for the evolution of RDF knowledge bases. The approach is based on a declarative definition of evolution patterns.

In [33], the authors provide a set of possible ontology change operations based on the effect with respect to the protection of the instance-data availability. Their aim is ensuring the validity of the instance level data rather than the schema level or domain specific operations. In [40], the impact of ontology change to the validity of the instance availability is discussed and changes are subdivided into two categories, i.e. structural and semantic changes. Though their work addresses semantic changes, our work takes the semantic changes further and proposes domain specific change patterns for semantic changes. In [46], the authors present a declarative approach to represent the semantics of changes, considered as a reconfiguration-design problem. Their work is focused on the realization of the changes, whereas our work is focused on identifying domain specific change patterns.

Representation of ontology changes using higher-level change operations was first proposed by Stojanovic [45] and Klein [27]. Recently, some researchers have focused on representation and detection of higher-level ontology changes [35,14]. In [35], the author proposed a language that allows formulating the intuition behind any applied change in an RDF graph and provided the change detection algorithm with respect to the proposed language. In order to detect composite changes, an algorithm compares two versions of the RDF graph (given in the form of triples in RDF/S). The algorithm first picks up a triple added to (or deleted from) the previous version of the knowledge base and looks for potential changes in a look-up table. Based on the potential changes identified, the algorithm searches for other added (or deleted) triples in order to detect certain type of ontology change. In contrast to their approach, we identify the composite changes from a single ontology change log graph and do not use different versions of an ontology for comparison. Thus, in order to realize that a certain ontology element exists in the previous version of the ontology, we search for an inclusion change operation (that adds the element in the ontology) in all previous change log sessions. If we find one, there must not exist a (later) exclusion change operation that cancels out the previous inclusion change until the start of the current session is reached.

Evolutionary strategies were first proposed by Stojanovic [44] where she considered them as solutions for keeping the ontology consistent at each resolution point. The resolution point refers to the places in ontology evolution where the user may adopt more than one option to keep the ontology consistent. For example, the instances of a deleted concept x will either be i) deleted, ii) the unique instance will be deleted or iii) are linked to the parent concepts of x.

A lot of research has been done on mining of process models from event log data [47,1,5,48,36,8]. An early work that relied on the activity logs for producing formal process models corresponding to actual process execution is given in [5]. Metrics such as event frequency and regularity were taken into consideration to discover process models. Results show its usefulness in activities such as process model discovery, re-engineering, software process improvement etc. In [48], the focus is on detection of invisible tasks from event logs. Their definition of invisible tasks is *tasks that exist in a process model, but not in its event log (such as initialize, skip, switch, redo, etc.)*. In contrast to their work, we are

interested in the detection of composite changes such as *split*, *move*, *merge* etc. In [36], the author defined the *Event* and *Process Mining* ontologies. These two ontologies can be used to incorporate semantics in the log, related to the event types and the process instances.

The mining of sequential patterns was first proposed by Agrawal and Srikant in [2]. Since then, many sequential pattern mining algorithms, often based on specific domains [31,37,43,51,3], have been suggested. In the domain of DNA or protein sequences, BLAST [3] is one of the most well-known algorithms. Given a query sequence (candidate sequence), it searches for a match from the databases. In contrast, we focus on mining of change sequences (patterns) from an ontology change database. In [51], the author 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.

Several algorithms focus on graph-based pattern discovery [21,49,30,20]. In [21], the author proposes an apriori-based algorithm, called AGM, to discover frequent substructures. In [49], the authors propose 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 [30] algorithm. The performance study shows that the gSpan outperforms the FSG algorithm and is capable of mining large frequent subgraphs. The Fast Frequent Subgraph Mining (FFSM) algorithm [20] 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 [49] and FSG [30] 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).

We adopted ideas from sequential pattern discovery approaches in other domains, such as sequence-independent structure pattern [20] and gap-constraint [31] for setting cutoffs in terms of node matching mechanism. Yet, discovery of change patterns from ontology change logs is relatively different from sequential pattern discovery in other contexts (such as the biomedical domain). Recently, few researchers have focused on detection of higher level generic ontology changes [35,14]. In contrast to their work, our approach is to discover the change patterns and is based on context-aware, semantic matching of different graph sequences.

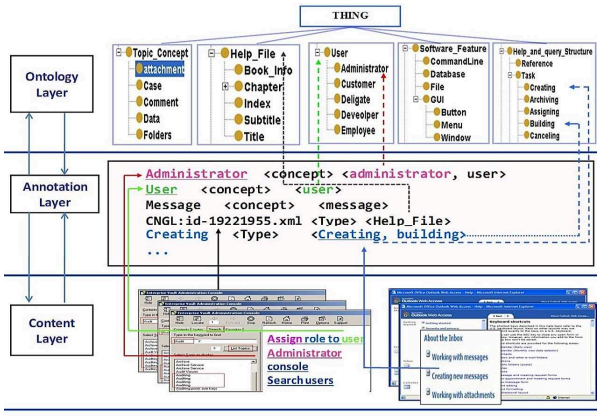

Fig. 2 Ontology-Driven Content-Based Systems (ODCBS)

This requires the identification of equivalency between unordered change sequences.

Our work regarding the formalization and storage of the discovered change patterns is relatively similar to the work of Henninger [19] and Kampffmeyer [26]. Similar to our generic metadata change ontology, Henninger uses ontology-based metamodels to formally present the software patterns. The core properties of the metamodels include `hasProblem`, `hasSolution`, `hasContext` etc. In order to represent the relationship among the patterns, the core metamodel was extended using properties `uses`, `requires`, `alternatives` etc. In contrast to our work, the focus of their ontology-based metamodel is to represent the relationships between the software patterns, rather than providing support of selecting the suitable pattern for a given task. In [26], author proposed a Design Pattern Intent Ontology (DIPO) for formalizing the patterns, based on their intent. The aim of DIPO is to support the software developers in choosing a design pattern, suitable for a particular task.

3 Ontology Change Management

We have been working with non-public ontologies used to annotate the content in large-scale information systems. The aim here is to facilitate accessibility of content for both humans and machines by integrating semantics in the content using ontologies. In this regard, a content change will ultimately affect all the artefacts - the access files being updated, the information system entities being improved. The latter causes knock-on effects on the access files and also the ontology-driven content management model (Figure 2).

We distinguish two categories of changes - changes to the content artefacts (content management infrastructure artefacts) and changes to the ontologies as the knowledge on top of the artifact layer [15].

– *Ontological Changes* can include changes in the concept hierarchy; some concepts may get modified, removed, pulled up/down in the hierarchy etc. More description (in a form of object/data properties) can be added to the available concepts. Ontological changes could reflect the general changes in the domain, flaws in the earlier conceptualization, addition of new concepts in the domain etc.
– *Content Changes* can affect any of the artifacts. Particular interest here is the cascading impact. A content change may have direct impact on access files. A change in access file may have direct impact on the annotations that link the files to the domain ontologies; and thus the underlying domain ontology may also evolve accordingly.

In this paper, we focus on ontological changes only. We propose a four-phase ontology change management system (Figure 1). These phases include, *change operationalisation (phase-1)*, *change representation (phase-2)*, *change semantic capturing (phase-3)* and *change pattern discovery (phase-4)*. In the following sections, we discuss each phase one after the other.

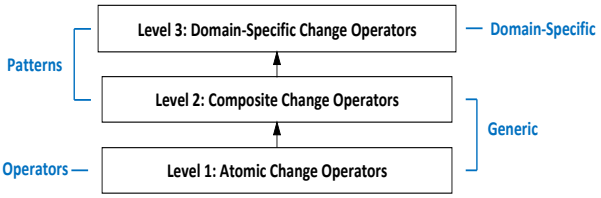

Fig. 3 Layered Framework of Change Operators and Patterns

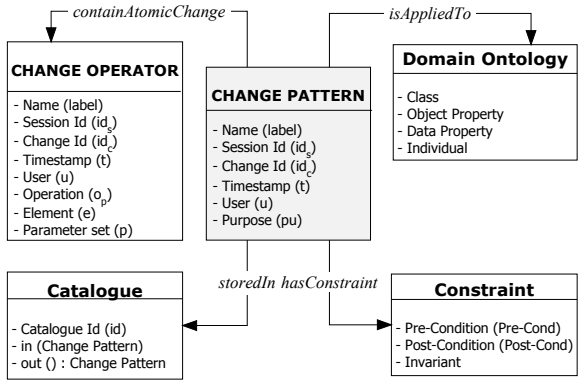

Fig. 4 Structural Model of Pattern-based Ontology Evolution

4 Layered Change Operator (LCO) Framework

Based on an empirical observation of common changes in different ontologies, we defined a layered framework of change operators (Figure 3). The first two layers are based on generic and structural change operators. The next layer covers domain-specific changes. We consider level two and three change operators as "change patterns".

We define a *change pattern* as a frequently occurring composed operation. The change pattern represents a frequently occurring constrained composition of lower-level change operations over the ontology elements. The main difference between a pattern and a composite operation is the frequency. It is the result of a mining process, whereas composite operations are language elements. These change patterns can either be generic or domain-specific. A generic change pattern can be applied to any domain ontology whereas, a domain-specific change pattern can only be applied to a specific domain ontology. The structural model of pattern-based ontology evolution is given in Figure 4.

Level One Change Operators - Atomic change operations: These change operators are the elementary change operations used to perform a single add or delete operation on a single targeted entity (i.e., addition or deletion of any particular axiom in the ontology). *"Add classDeclarationAxiom(Student)"*, *"Delete subClassOfAxiom(Student, Person)"*, etc., are the examples of level one change operations.

Level Two Change Operators - composite change patterns: These are aggregated changes to represent composite tasks. Many evolution tasks cannot be done by a single atomic operation. These change operators are identified by grouping atomic operations of level one to perform a composite task on target entities.

Composite change operations comes into two layers. First layer of composite change operations includes group of those atomic change operations that in general are executed together. For example, *"Remove Concept Context"* which not only deletes a concept from the class hierarchy, but also deletes all its roles. To delete a single concept *"faculty"* in a university ontology, removing the concept from the concept hierarchy is not sufficient. Before we remove the concept, we have to remove all its roles, such as, removing from the domain and the range of properties like *"isSupervisorOf"* or *"hasPublication"*, etc. In addition, we need to either delete its orphaned subclasses or link them to the parent concept, in order to keep the ontology consistent.

If an ontology engineer wants to merge two or more concepts, the operation requires operators higher than the integrate/remove concept context. In such a case, composite change operations from level two can be used. *"merge concepts"*, *"move concepts"* or *"pull up property"* are examples of layer two composite change operations.

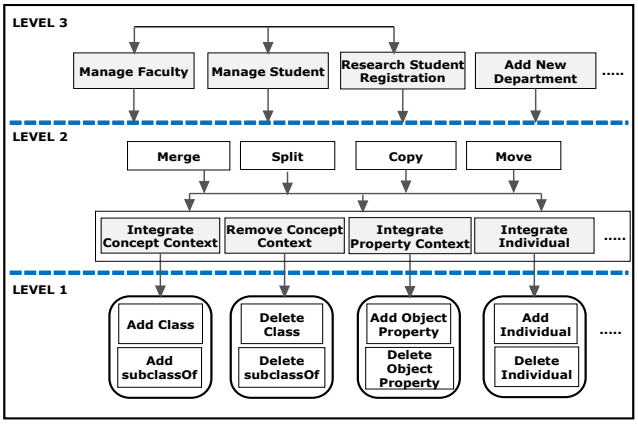

Fig. 5 Architecture of layered change operators (*University Ontology*)

Level Three Change Operators - domain-specific change patterns: The changes at a higher level of granularity, which are frequent in a domain, can be represented as domain-specific patterns - which are often neglected by the lower-level compositional change operators. Domain-specific perspective links the structural changes to the aspects represented in domain ontologies. In order to execute a single domain-specific change, operations at level one and two are used. In case of the university administration domain, level three may contain change patterns such as *"manage faculty"*, *"add new department"*, *"student registration"*, etc (Figure 5). If a user needs to register a new category of faculty using the *"manage faculty"* change pattern, say a *"JuniorLecturer"*, then he creates a concept *"JuniorLecturer"* and attaches properties like *"hasPublication"* and *"supervise"* from level two to the newly created concept. Another ontology engineer may create a new concept *"JuniorLecturer"* without including the *"supervise"* property. This is due to the different viewpoints and perspectives of the users.

More details about the change operator framework can be found in [22].

5 Layered Change Log Representation

Ontology change logs can play a significant role in ontology evolution. If there is a need to reverse a change, we use the change log to undo/redo the changes applied in the past. This is a common function in e.g. software versioning support. In collaborative environments, change logs are also used to keep the evolution process transparent and centrally manageable. It captures all changes ever applied to any entity of ontology using elementary changes. We propose a mechanism of representing ontology changes expressively at different levels of granularity (i.e. fine-grained changes such as the creation of a single class and also coarse-grained changes such as merging two sibling classes [38]). The higher-level change representation is used for

- describing how an ontology is evolved from one version to the other.
- In distributed environment where complex relationships can exist between domain ontologies and other artefacts, an ontology change may need to propagate to dependent artefacts. In such cases, higher level change representation assists in understanding the ontology change and the impact (consequence) of the applied changes.
- bridging between operational and analytical aspects of the ontology evolution.

5.1 Layered Change Log Model

Capturing and representing the ontology changes at the elementary level in a change log does not suffice. As the intent of the ontology change is missing from such change logs (and mostly specified at higher level of granularity), the ontology engineer is unable to understand why changes were performed, whether it is an elementary level change or a part of composite change and what the impact of such change is. We attempt to mine valuable information from a change log, making it easy for the ontology engineer, (other) users and machines to understand and interpret the ontology modifications. We propose a layered change log model, containing two different levels of granularity, i.e. an Atomic Change Log (ACL) and a Pattern Change Log (PCL), shown in Figure 6. The layered change log works with the lay-

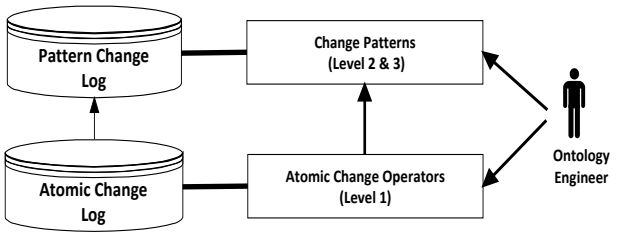

Fig. 6 Layered Ontology Change Framework

ered change operator framework presented in section 4. The atomic change log records the applied ontology changes at the atomic level. The pattern change log records the applied ontology changes in terms of higher level (composite and domain-specific) change patterns. These change patterns can either be implicit if they are mined from the atomic change log using data mining techniques or explicitly defined by the user as higher

level (level two and three) change operators. A pattern change log supports capturing the objective of the ontology changes at a higher level of abstraction and helps an comprehensive understanding of ontology evolution. Higher level change representation is more concise, intuitive and closer to the intentions of the ontology editors and captures the semantics of the change [35]. Storing ontology changes at two different levels of abstraction also helps us in identifying recurring change patterns from low level logs (discussed in Sec. 6 and 7).

Atomic Change Log (ACL): An atomic change log consists of an ordered list of atomic change operations, $ACL =< ac_1, ac_2, ac_3 \cdots ac_n >$ where n refers to the sequence of ontology changes in a change log. Each ontology change contains two types of data, i.e. *Metadata* (MD) and the *Change data* (CD) (Figure 7). 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. Metadata can be given as $MD = (id_s, id_c, u, t)$ where id_s, id_c, u and t represent session id, change id, user and timestamp, respectively. The change data contains the central information about the change request and can be given as $CD = (o_p, e, p)$ where, o_p, e and p represent the change operation, element and parameter set of a particular change.

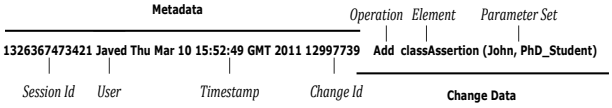

Fig. 7 Representation of an Atomic Ontology Change

Pattern Change Log (PCL): 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 sequence of ontology change patterns in a pattern change log. These change patterns can either be level two generic composite change patterns or level three domain-specific change patterns (c.f. Figure 5). Similar to ACL, each ontology change pattern *pc* consists of two types of data i.e. *Metadata (D)* and *Pattern data (P)*. The metadata provides meta details about the change pattern and can be given as $D = (id_s, id_c, u, t, p_u)$ where, id_s, id_c, u, t and p_u represent the session id, change id, user, timestamp and intention of the change pattern, respectively.

The pattern data (P) provides description about the involved change operations. Here, P refers to the sequence of the change operations available in a change pattern $P = (ac_1, ac_2, \ldots ac_s)$ where, s is the total number of change operations in a pattern. 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 8).

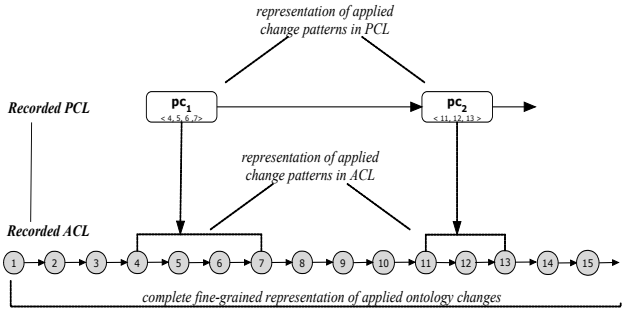

Fig. 8 Operational setup of ontology change logging

5.2 RDF Framework Format

We use RDF triple-based representation, i.e., *subject - predicate - object* (spo), to conceptualize the ontology changes in change logs. To do so, we constructed a generic metadata ontology[1] based on specification of OWL-DL 2.0. The classes and properties available in the metadata ontology assist the ontology engineer to construct the RDF triples, representing an applied ontology change. Similar to the approaches opted for [34] and [36], the idea here is to provide a metadata ontology that is generic, independent and extendable to represent the changes of the domain ontologies. We used an RDF triple store to record the change log, domain ontologies and metadata ontology. Thus, all ontology changes, stored in the ontology change log, are in a form of triples. Below, we give description of the metadata ontology, using an example of stored change pattern, given in Figure 9.

The central class in the metadata ontology is *Change*. Based on our proposed change operator framework (cf. Sec. 4), the class *Change* is subdivided into *AtomicChange*, *CompositeChange* and *PatternChange*. Each stored domain-specific change pattern is an instance of (rdf:type) *PatternChange* (line 1: Figure 9). The descriptive data of a change pattern is given using properties *sessionId*, *changeId*, *PatternName*, *Timestamp*, *Purpose* etc. (line 2-7). In order to express that the change pattern is the combination of lower level change operations (i.e. atomic, composite or combination of them), the class *PatternChange* is associated to the

[1] Available at `www.computing.dcu.ie/~mjaved/MO.owl`

```
Namespaces:  MO:          http://www.cngl.ie/ontology/MO.owl#
             University:  http://www.cngl.ie/ontology/University.owl#

1.  <MO:ResearchStudentRegistration> <rdf:type> <mo:PatternChange> .

2.  <MO:ResearchStudentRegistration> <MO:sessionId> "7536902801513" .
3.  <MO:ResearchStudentRegistration> <MO:changeId> "1323865264484" .
4.  <MO:ResearchStudentRegistration> <MO:hasCreator> <MO:Javed> .
5.  <MO:ResearchStudentRegistration> <MO:Timestamp> "Wed Dec 14 12:21:04 GMT 2011" .
6.  <MO:ResearchStudentRegistration> <MO:PatternName> "PhD Student Registration" .
7.  <MO:ResearchStudentRegistration> <MO:Purpose> "Purpose is to register a new phd student" .

8.  <MO:ResearchStudentRegistration> <MO:containAtomicChange> <MO:AddIndividual> .
9.  <MO:ResearchStudentRegistration> <MO:containAtomicChange> <MO:AddIndividualType> .
10. <MO:ResearchStudentRegistration> <MO:containAtomicChange> <MO:AddObjectPropertyAssertion> .
..
..
11. <MO:AddIndividual> <rdf:type> <MO:AtomicChange> .
12. <MO:AddIndividual> <MO:changeId> "13238652644840" .
13. <MO:AddIndividual> <MO:hasEntity> <MO:Individual> .
14. <MO:AddIndividual> <MO:hasTargetParam> <MO:param1> .
15. <MO:AddIndividual> <MO:hasOperation> <MO:Add> .

16. <MO:AddIndividualType> <rdf:type> <MO:AtomicChange> .
17. <MO:AddIndividualType> <MO:changeId> "13238652644841" .
18. <MO:AddIndividualType> <MO:hasAuxParam1> <University:PhD_Student> .
19. <MO:AddIndividualType> <MO:hasIndividualAxiom> <MO:classAssertionAxiom> .
20. <MO:AddIndividualType> <MO:hasTargetParam> <University:param1> .
21. <MO:AddIndividualType> <MO:hasOperation> <MO:Add> .
..
..
```

Fig. 9 RDF triple-based specification of stored change pattern "ResearchStudentRegistration"

class *AtomicChange* and the class *CompositeChange* using object properties *containAtomicChange* and *containCompositeChange*, respectively (line 8-10).

5.3 Graph-based Ontology Change Formalization

Ontology change logs provide operational as well as analytical support in the evolution process. As discussed in section 5.2, the ontology change logs are stored in the form of RDF triples. RDF triple format is used due to its fine-grained level representation and interoperability (i.e., conversion from triple format to others standard formats such as RDF, XML etc.). Fine-grained representation of ontology changes help the ontology engineer to construct complex queries and extract different types of knowledge from the log. Furthermore, storing domain ontologies, metadata ontology and change log in one single location helps in navigating through them simultaneously and identifying relationships among them. However, as RDF triples represent the ontology changes at fine-grained level (1 ontology change is represented by 8-10 triples), efficiently visualizing and navigating through the change log alone is not realistic. Graphs can cover this gap. Graphs 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 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 [11]. 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 10). The benefit of using ATGs and AGs is their similarity with the 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.

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. Here, we assume that the concurrent ontology change operations (if any) are sequenced; i.e., each ontology change operations is executed one after another.
- $N_A = \{n_a^i | i = 1, \ldots, q\}$ 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 7. The term q refers to the total number of attributes attached to a single graph node n_g.
- $E_G = \{e_g^i | i = 1, \ldots, p-1\}$ 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.

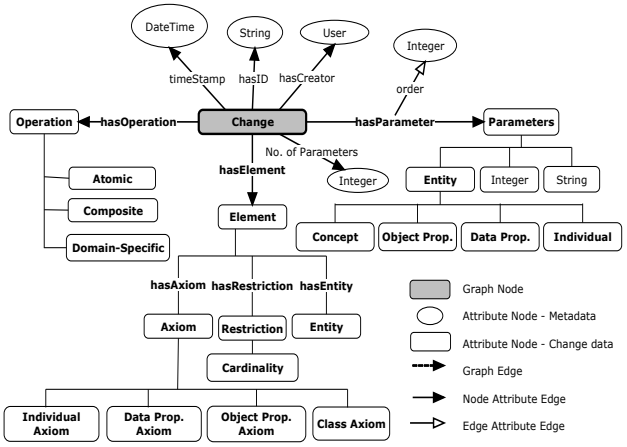

Fig. 10 Attribute Type Graph (ATG) for an Ontology Change

- $E_{NA} = \{e^i_{na} | i = 1, \ldots, r\}$ is the set of node attribute edges which joins an attribute node n_a to a graph node n_g.
- $E_{EA} = \{e^i_{ea}) | i = 1, \ldots, q - r\}$ is the set of edge attribute edges which joins an attribute node n_a to a node attribute edge e_{na}.

A single graph node of an attributed graph (AG) which is typed over an ATG is given in Figure 11. The types defined on the (graph and attributed) nodes can be given as $t(Add)$=$Operation$, $t(classAssertion) = Element$, $t(John) = Individual$ and $t(PhD\_Student) = Concept$. This node represents a single ontology change operation where graph node n_g is the central part of it. These graph nodes are linked to each other using graph edges e_g to represent a complete ontology change log graph.

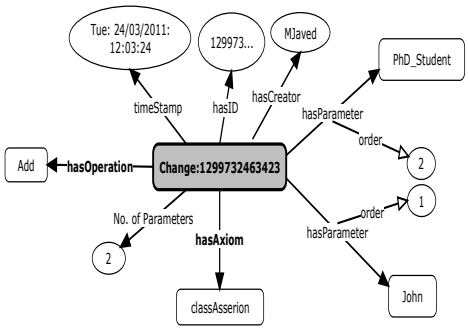

Fig. 11 Attributed Graph Node typed over ATG (Add classAssertion (John, PhD_Student))

6 Detection of Composite Change Patterns

As we discussed in section 5, representing a change at the atomic level is not sufficient. Such representation of ontology changes can only describe the addition or deletion of an ontology element. The semantics of an applied change (or a group of changes) are missing from such representation and most of the time is present at higher level of granularity. We can represent such semantics of the applied atomic changes in the form of composite change patterns and can be represented in the pattern change log (PCL). For example, if a concept x is removed (as a subclass) from a parent concept y and has been attached (as a subclass) to another concept s, the semantics behind such change (at atomic level) only refers to a change of the class hierarchy for concept x, i.e. `Move Concept (x, y, s)`. In this case, instances of concept x that inherit properties from concept y or any of its parents, need to be revalidated. However, if we identify this knowledge that concept s is actually a superclass of concept y, the semantics behind such a change will refer to a "pull up concept" change - `Pull up Concept (x, y, s)`. In this case, instances of concept x that inherit properties of concept y only need to be revalidated. This example signifies the importance of capturing the semantics of any change at a higher level. We operationalize the change semantic captured using composite change detection algorithms. The algorithms lead us to the detection of composite changes from ACLs and their (semantically) enhanced representation in the PCL. It is common to find some overlapping change patterns. Such overlapping of the change patterns can be either *complete* or *partial* (Figure 12).

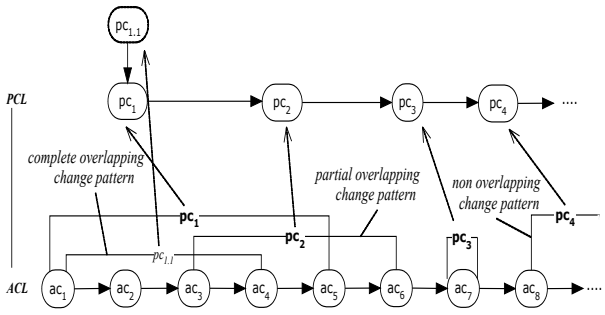

Fig. 12 Layered Change Log Framework

6.1 Composite Change

A composite change is a sequence containing a group of elementary (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 data to the domain ontology, whereas the exclusion type change op-

erations remove data from the domain ontology. Thus, a composite change c can be given as $< \delta_1, \delta_2, \phi >$, where:

- δ_1 is a sequence of atomic level exclusion change operations.
- δ_2 is a sequence of atomic level inclusion change operations.
- ϕ refers to the conditions to be satisfied.

As composite change operations are applied at the entity level, an exclusion change operation (δ_1) deletes certain axioms from the target ontology entity. The inclusion change operation (δ_2) adds some new axioms regarding the target ontology entity. Further, to consider a group of (add/delete) change operations as a composite change, the change operations must satisfy certain conditions. The term ϕ refers to the *conditions* on the existence of any knowledge in the ontology. Such conditions can be either existential conditions (ϕ_e) or correlations (ϕ_c) among the ontology change parameters. The existential conditions (ϕ_e) of any change operation can be given in terms of pre and post conditions. For example, in case of change operation `Add concept` (`Researcher`), the individual `Researcher` must not exist in the current version of ontology (O_1) and must exist (as a concept) in the next version of ontology (O_2).

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

Similarly, in case of `Add subclassOf` (`Researcher` , `Person`), concepts `Researcher` and `Person` must exist in the current version of ontology (O_1) and `Researcher` should be a child of (rdfs:subClassOf) `Person` in subsequent version of ontology (O_2).

- Pre-Cond: `Researcher`, `Person` $\in O_1$
- Post-Cond: (`Researcher` rdfs:subClassOf `Person`) $\in O_2$

The correlations (ϕ_c) refer to the relationships among the parameters of the available atomic change operations in a composite change. Such relationships are not explicitly given in the atomic change log. For example, in case of composite change operation `Pull up concept` (`Researcher`, `Student`), where the concept `Researcher` is being pulled up in the concept hierarchy and becomes a sibling class to its previous parent concept `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 correlations can be given as

- `Student subclassOf Person.` (ϕ_c)

We utilized the given definition of a composite change in defining the graph transformation rules and conditions. In other words, we can say that a source ontology subgraph has been transformed into target ontology subgraph based on the given conditions, i.e., existential and correlation conditions.

6.2 Graph-based Specification of a Ontology Change

We specify the ontology changes of composite types using a graph transformation approach where a source ontology subgraph is transformed into a target ontology subgraph, while preserving the defined conditions. We opt for the double pushout (DPO) [10] approach that allows us to specify the *graph transformation rules* and *gluing conditions* in a form of pairs of graph morphisms ($L \xleftarrow{l} K \xrightarrow{r} R$) – Figure 13. First, we describe the DPO approach.

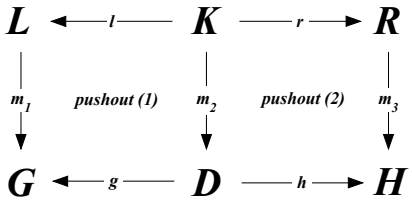

Fig. 13 Double-pushout approach for Graph Transformation

Referenced vs. Ontology subgraphs. The DPO approach is termed as "double pushout" as the complete transformation of input ontology subgraph G into target ontology subgraph H is translated into two types of changes, i.e., exclusion and inclusion change operations. The DPO approach uses a graph mapping approach where subgraphs L, K and R represent the *referenced* subgraphs and subgraphs G, D and H represent the *ontology* subgraphs. Thus, we can say that the ontology subgraphs G, D and H are mapped to referenced subgraphs L, K and R, respectively (Figure 14).

The graph L is the referenced input subgraph representing items (i.e., nodes or edges) that must be present 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. The graph R is the referenced output subgraph representing the items that must be present in the resulting target ontology subgraph H, after the application of the composite change, i.e., representing the final state of the ontology; whereas, 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 - not the transformation of a "change log" subgraph. Each node in a DPO graph represents an ontology entity (i.e. class, property or individual). In Figure 14, the nodes and the edges represent the ontology classes and the subclassOf axioms, respectively. The change log graphs, are mentioned here in the form of *productions* and *co-productions* (discussed below), representing the set of atomic change operations.

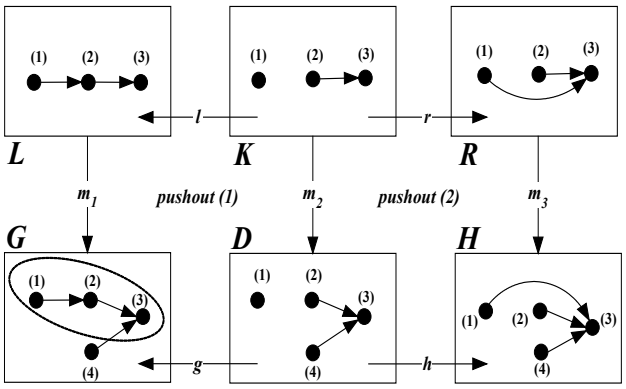

Fig. 14 DPO Approach - An example

The *graph transformation rules*, also known as *productions* (p), represent the change operations being applied on the subgraphs during the two pushouts. The rules define the correspondence between the source and the target subgraph determining what is to be deleted, preserved or constructed. For example, in Figure 13, first production, represented as l, refers to the exclusion change operations of pushout 1 that delete certain items (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 (nodes or edges) into the reference gluing graph K. The productions representing the changes being applied on the input ontology subgraph G are known as *co-productions* and are given as g and h in Figure 14.

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 node/edge of graph L is mapped to a node/edge in graph G in such a way that graphical structure and labels are preserved [6].

The context gluing graph D is obtained by deleting all items (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 (nodes and edges) of subgraph R thats 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 a node v is deleted from graph G, all the edges that contain 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., 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) [18].

6.3 Triple pushout (TPO) Approach

We take pushout 1 and 2 of DPO approach as "structural pushouts", as they refer to completeness and correctness of the structure of a graph. The dangling condition for edges in pushout 1 of DPO approach does ensure that the graph D is a genuine graph by deleting the dangling edges. However, the semantics behind the applied composite change may be lost, e.g. in the case of *split concept* change. Let x be an ontology concept that is split into two concepts x_1 and x_2 (Figure 15). In pushout 1 of split concept change, concept x is deleted from the concept hierarchy. In order to satisfy the dangling condition, the roles (edges) of the concept x are also being deleted. In pushout 2, two new concepts x_1 and x_2 are added replacing concept x in the concept hierarchy. As concepts x_1 and x_2 inherit relationships from the split concept x, the deleted roles (edges) are not the consumed entities in this graph transformation. Thus, the relationships must be added back to the newly added concepts x_1 and x_2.

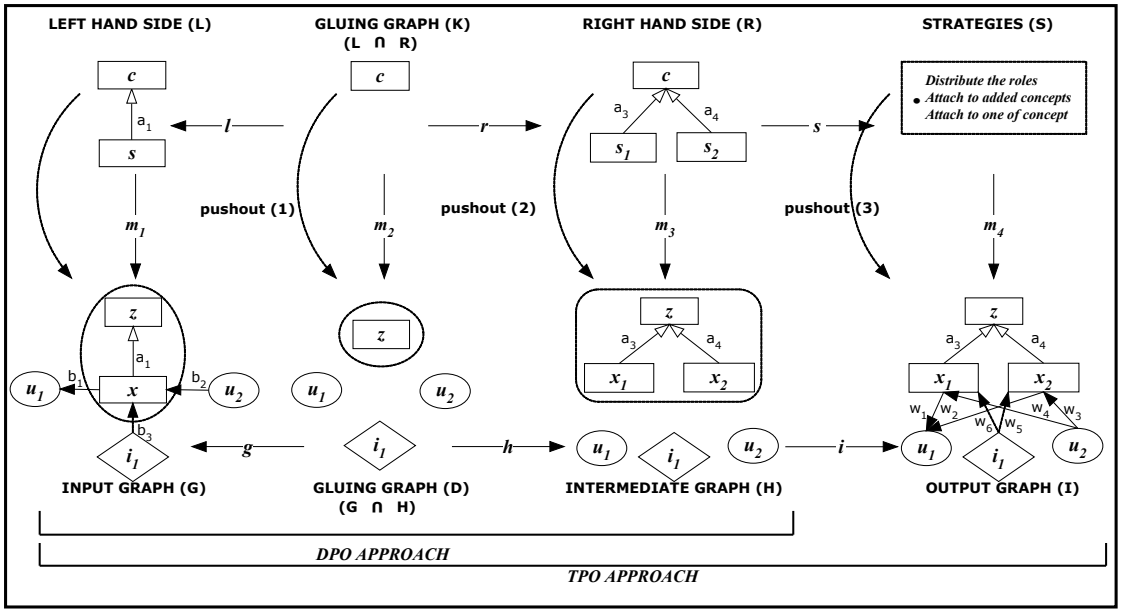

Fig. 15 Split Concept (x, (x_1, x_2)) - triple push out (TPO) Approach

To do so, we extended the DPO approach by adding an additional production that formulates the pushout 3 (semantic pushout) allowing a user to preserve the deleted dangling edges of pushout 1; hence, named "triple pushout approach". In pushout 3, the user can select different evolution strategies [25] in order to resolve the above mentioned issues. Thus, a derivation i from subgraph H to I resulting from an application of production s (pushout 3) at a match m_3 can be given as $i = H \overset{s,m_3}{\Longrightarrow} I$ (Figure 15). Now, we explain the TPO approach in detail using the "split concept" composite change case scenario.

"Split concept" Change Scenario. The composite change "split concept" refers to splitting a concept into two (or more) sibling concepts. For example in Figure 15, the concept x ($x \in G$) has been split into two sibling concepts x_1 and x_2 ($x_1, x_2 \in I$). The composite change adds two new concepts in the ontology (inclusion operations) and deletes the concept that has been split (exclusion operation). The nodes and edges, given in Figure 15, represent the following ontology elements: square node $\longrightarrow$ concept(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 1 gives the formal definition of the split concept composite change example given in Figure 15, in terms of ontology and TPO graph changes and conditions. Now we discuss each pushout and the involved change operations one after the other.

Table 1 Formal Definition of Composite Change Operation: *Split Concept(x, (x_1, x_2))*

Split Concept (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)
Inclusion Changes (δ_2)	**Pushout–3**	(Type)
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)	**Graph 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 1 : First, we identify the occurrence of the reference subgraph L in ontology graph (i.e., $m_1 : L \longrightarrow$

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 concept x from the ontology concept hierarchy. Thus, in Figure 15, 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.

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) resulting into an intermediate graph H (i.e., $h = D \overset{r,m_2}{\Longrightarrow} H$). The co-production h represent the addition of two concepts x_1 and x_2 in the ontology concept hierarchy. Thus, in Figure 15, the nodes x_1 and x_2 are added to the gluing graph D and are linked to available node z through edges a_3 and a_4.

pushout 3 : In order to ensure that the roles (c.f. Figure 16) of the deleted concept have been transferred to the newly added concepts, the effect of the dangling condition must be reversed. We call it *inverse dangling condition* (C'_d). Thus, all the edges that had been removed from the graph in pushout 1 (due to dangling condition), must be added back to the newly added concepts in pushout 3.

In pushout 3, the user can select different evolution strategies for inheriting the roles of the deleted concept by the newly added concept. For example, in case of the split change operation, the user can either, 1) distribute the roles among the newly added concepts, 2) add the roles to both concepts or add the roles to one of the added concept ($i = H \overset{s,m_3}{\Longrightarrow} I$). As in our running example, we chose option 2, the nodes u_1, u_2 and i_1 are linked to the nodes x_1 and x_2.

6.4 Detection of Composite Changes

We operationalize the composite change detection in terms of graph matching. The TPO approach can be applied directly, if one preserves the different versions of the ontology. As we log the applied change operations, rather than the different versions of the ontology, we input the productions 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. Below, we describe some frequently used terms:

Session(s): The ontology change log graph is a collection of sessions S, where each session s consists of the change log entries, from the time the domain ontology is loaded into the ontology editor, till the time it is closed. Thus, whenever an ontology is loaded into the editor, a new session starts and all the applied changes are recorded into the following session.

Graph Node vs. TPO Node: One should differentiate between a graph node of a change log graph and a node given in a TPO diagram. In this paper, the term "graph node" represents a single ontology change log entry (i.e., representing a single atomic change) in a change log graph, where each graph node comprises of a number of attributes such as *target/auxiliary parameters*, *operation*, *element*, *session id* etc - (c.f. Figure 11). The term "TPO node", represents an ontology entity (i.e., concept, property, individual, etc.) in the TPO Figure 15.

Role: The term "role" refers to the usage of an ontology entity in a specific ontology version. For example, in Figure 16, the concept *ResearchStudent* has five roles i.e, subclass of concept *Student*, range of object property *isSupervisorOf* and the type of individuals *Joe*, *Karl* and *John*.

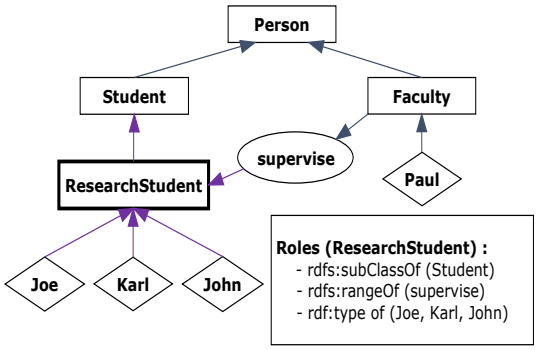

Fig. 16 Role of a Concept *ResearchStudent*

6.4.1 Algorithms for Composite Change Detection

There is no agreed standard set of composite change operations that one could be based on. It is obvious (and also mentioned in previous research [44,27]) that one can combine different atomic level change operations in order to construct new composite changes. Thus, providing an exhaustive list of composite change operations is not feasible. In our current work, we select the

Table 2 List of composite change patterns and their definitions

Composite Change	Description
Split Concept (x, (x_1, x_2))	Split a concept x into two newly created sibling concepts x_1 andx_2.
Merge Concept ((x_1, x_2), x)	Merge two existing concepts x_1 and x_2 into one newly created concept x and cumulate all roles of x_1 and x_2 into x.
Pull up Concept (x, x_1)	Pull concept x up in its class hierarchy and attach it to all parents of its previous parent x_1.
Pull up Concept (x)	Pull concept x up in its class hierarchy and attach it to all parents of all its previous parents.
Pull down Concept (x, x_1)	Pull concept x down in its class hierarchy and attach it as a child to its previous sibling concept x_1.
Pull down Concept (x)	Pull concept x down in its class hierarchy and attach it as a child to all its previous sibling concepts.
Move Concept (x, x_1)	Detach concept x from its previous superclass and attach it as a subclass to a concept x_1 (which previously was not a direct/indirect superclass of concept x).
Group Concepts (x, (x_1, x_2))	Create a common parent concept x for sibling concepts x_1 and x_2 and transfer the common properties to it.
Add Generalisation Concept (x, x_1)	Add a new concept x between x_1 and all its super classes.
Add Specialization Concept (x, x_1)	Add a new concept 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 concept 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 concept x_1.

composite change patterns and their definitions from [44] and they are given in Table 2.

The basic idea of the composite change detection algorithm is to iterate over each session of the change log graph and find the location from where an applied composite change may start. Pass the identified location's session node n_g and the reference graph G_r to a function that extracts the complete sequence of nodes (starting from n_g) that maps completely to G_r. In the mapping step, ensure that the correlations among the parameters of the identified change operations are satisfied.

6.4.2 Description of Algorithm

The complexity of the presented algorithm is linear $O(n)$. The composite change detection algorithm is given in listings 1.1 and 1.2, where listing 1.1 describes the main algorithm and listing 1.2 presents algorithm for one of the function (method). Below, we describe the algorithm in form of 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 group 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 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 a method, i.e., *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 composite change was not identified at particular location of the session) (line 13–16).

Listing 1.1: Composite Change Detection Algorithm

Input: Change Log Graph (G) and Reference Graph (G_r)
Output: Set of Split 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 ← getFirstNode(G_r)
 9:      n_g ← 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 ≠ null then
15:        SC ← 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.

Listing 1.2: Method: matchPattern()

```
Input: Matched Graph Nodes n_g, n_r and session s
Output: Split Composite Change (list)
 1: list ← n_g
 2: while list is not complete do
 3:    n_r ← getNextNode(G_r)
 4:    n_g ← 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 Change Patterns Discovery

Graph-based formalisation (discussed in Sec. 5.3) 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 that combinations of change operations occur repeatedly during the evolution of ontologies. We identified these as frequent recurring change patterns that can be reused.

7.1 Analysis of Change Log Graph

While patterns are sometimes used in their exact form, users often use different orderings of change operations to perform the same (semantically equivalent) changes at different times. To capture semantically equivalent, but operationally different patterns, more flexibility is needed. We introduce a metric, called *sequence gap* or generally *n-distance*, that captures a node gap between two adjacent graph nodes in a sequence [23]. It refers to the distance between two adjacent graph nodes in a change log graph. This helps us to define a more flexible pattern notion. We merge different types of patterns into two basic subdivisions, i.e.

- *Ordered Change Patterns (OP)*
- *Unordered Change Patterns (UP)*

The instances of the ordered change patterns comprise change operations in exact same sequential order from a change log graph. Thus, such complete (OCP) or partial (OPP) patterns may have only a positive node distance value, starting from zero to a user given threshold (x). The instances of unordered change patterns comprise change operations which may or may not be in the exact same sequential order in a change log graph. These complete (UCP) or partial (UPP) patterns may have a node distance that ranges from a negative node distance value ($-x$) to a positive node distance value (x). Completeness means that all pattern nodes are used in the concrete graph; partiality refers to a subset of nodes. For the remainder, we focus on complete change patterns, but we discuss the relevance of partial change patterns in our conclusions.

Metrics: We consider identifying recurring sequenced change operations from a change log as a problem of recognition of frequent patterns in a graph. First we describe the key metrics.
Definition 1 - Pattern Support: The *pattern support* of a pattern p is the number of occurrences of such a pattern in the change log graph G. Pattern support is denoted by *sup(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_sup(p)*.
Definition 2 - Pattern Length: The *pattern length* of a pattern p is the number of change operations in it, denoted by *len(p)*. 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)*.
Definition 3 - Candidate Change Pattern Sequence: For a given $ACL =< ac_1, ac_2, ac_3 \cdots ac_n >$, a candidate

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 $\text{pos}(ac_{pi}) < \text{pos}(ac_{pj})$ in cs, then $\text{pos}(ac_{pi}) \leq \text{pos}(ac_{pj})$ in ACL ...
for all $i = 1 \cdots k-1$ and $j = 2 \cdots k$.

Definition 4 - 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 pattern sequence cs is equal to or greater than the threshold value set by the minimum pattern length.
- $sup(cs) \geq min\_sup(p)$.
  i.e., the support for the candidate pattern sequence cs in a change log graph G is above the threshold value of minimum pattern support.

Definition 5 - Ordered Change Pattern: Let a change pattern $p =< s_1, s_2 \cdots s_d >$ be a set consisting of a candidate change pattern sequence cs ($cs = s_1$) and the change pattern support sequences ($s_2, s_3 \cdots s_d$) for cs. The change pattern p is an ordered change pattern (OP) with
- $s_i =< ac_{i1}, ac_{i2} \cdots ac_{in} > \in p$ for $i = 1 \cdots d$
- if $\text{pos}(ac_{ix}) < \text{pos}(ac_{iy})$ in s_i then $\text{pos}(ac_{ix}) \leq \text{pos}(ac_{iy})$ in ACL
for all $x = 1 \cdots n-1$, $y = 2 \cdots n$.

Definition 6 - Unordered Change Pattern: Let u and v be the minimum and maximum positions in an identified change pattern sequence s_i, respectively. The change pattern p ($p =< s_1, s_2 \cdots s_d >$) is an unordered change pattern (UP) with
- $s_i =< ac_{i1}, ac_{i2} \cdots ac_{in} > \in p$ for $i = 1 \cdots d$
- if $u = min\_pos(ac_{i1}, ac_{i2} \cdots ac_{in})$ in s_i and $v = max\_pos(ac_{i1}, ac_{i2} \cdots ac_{in})$ in s_i, then $u \leq \text{pos}(ac_i) \leq v$ in ACL.

7.2 Change Patterns Discovery Algorithms

The identification of domain-specific change patterns is operationalised in the form of discovery algorithms. The section is divided into two parts, i.e. algorithms for searching ordered complete change patterns (OCP) and algorithms for searching unordered complete change patterns (UCP). The inputs to the pattern discovery algorithms comprise the graph G representing change log triples, the minimum pattern support *min_sup*, the minimum pattern length *min_len* and the maximum node-distance x. Before we describe each algorithm, we introduce some concepts.

- *Target entity, primary/auxiliary context of change*: The *target entity* is the ontology entity to which the change is applied; *primary/auxiliary context* refers to entities which are directly/indirectly affected by such a change.
- *Candidate node (cn)*: A candidate node cn is a graph node selected at the start of the node iteration process (discussed later). Each graph node will act as a candidate node cn in one iteration each of the algorithm.
- *Candidate sequence (cs)*: The candidate sequence cs is the context-aware set of graph nodes starting from a particular candidate node cn.
- *Discovered node (dn)*: The discovered node dn is a graph node that matches the candidate node cn (in a particular iteration) in terms of its operation, element and type of context. DN refers to the set of discovered nodes.
- *Discovered sequence (ds)*: ds is the context-aware set of graph nodes starting from a discovered node dn that matches candidate sequence cs (in an iteration). DS refers to the set of discovered node sequences.

7.2.1 OCP Discovery Algorithm

To discover ordered complete change patterns (OCP), the identified sequences are of the same length and contain change operations in the exact same chronological order. The basic idea of the algorithm is to iterate over the graph nodes, generate the candidate sequence starting from a particular graph node and search the similar sequences within the graph G. The *OCP* algorithm is defined in listing 2.1 – 2.2. The algorithm iterates over each graph node and selects it as a candidate node (cn_k), where k refers to the identification key of the node. Once the candidate node is selected, an iterative process of expansion of candidate node cn_k to its adjacent nodes cn_{k++} starts and continues until more expansion is not possible (i.e. adjacent node do not share the same target entity). If the target entity of the adjacent node is matched with the target entity of the candidate node, it is 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 the target entity of the candidate node. The iteration continues based on the user threshold x, i.e. the allowed gap between two adjacent graph nodes of a pattern (*n-distance*).

Listing 2.1: OCP Discovery Algorithm

Input: Graph (G), Min. Pattern Support (*min_sup*), Min. Pattern Length (*min_len*), Max. 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 ← GenerateCandidateSequence(n(g_i))
4:   if (cs.size < min_len) then
5:     go back to step 1.
6:   end if
7:   DN ← DiscoverMatchingNodes(cn_k)
8:   DS ← DN
9:   if (DS.size < min_sup) then
10:    go back to step 1.
11:  end if
12:  while (DS.size ≥ min_sup) do
13:    for each discovered sequence ds in DS do
14:      t ← getTargetEntity(ds)
15:      Expand(dn_j, x)
16:      Match(dn_{j++}, cn_{k++}, t)
17:      if (Expanded && Matched) then
18:        ds ← 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 ← get Maximum length of sequences such that
     (max ≥ min_sup)
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} ← (ds + cs)
36:  S ← P_{domain_specific}
37: end for
```

Once the candidate sequence is constructed and is above the threshold value for the minimum pattern length, the next step is to search for the matching nodes (i.e. discovered nodes dn) of the same type as the candidate node cn_k. If the number of discovered nodes is above the threshold value (minimum pattern support), the 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 the next node of a discovered sequence ds is an iterative process (depending on x).

Listing 2.2: GenerateCandidateSequence()

```
Input: Graph (G), Maximum n-distance (x), Graph Node
    (n)
Output: Candidate Sequence (cs)
1: k = 0
2: cn_k ← n
3: cs ← cn_k
4: context = true
5: while (context) do
6:   Expand(cn_k, x)
7:   if (Exanded) then
8:     cs ← cn_{k++}
9:   else
10:    context = false
11:  end if
12: end while
13: return cs
```

The expansion of a discovered node dn stops if either more expansions of that node are not possible or the expansion has reached the size of the candidate sequence (i.e. the length of ds is equal to the length of cs). 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 the minimum pattern length, it must be discarded from the set of discovered sequences.

Once the expansion of discovered nodes is finished, in order to identify the change patterns of greater size, 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 the minimum pattern length and the number of identified sequences is greater than or equal to the threshold value of minimum pattern support. 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 max are truncated to size max.

As a last step, the candidate sequence along with the 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 the next graph node as a candidate node.

7.2.2 UCP Discovery Algorithm

A collection of change operations is not always executed in same chronological order, even if the result is the same. As then the change operations in a sequence can be reordered, the aim is to discover unordered complete change patterns by modifying the node search space in each iteration. The pseudocode of the *UCP* algorithm is given in listing 3.1 – 3.2.

Like OCP, UCP iterates over each graph node and selects it as a candidate node (cn_k). An iteration is used to construct a candidate sequence cs by expanding candidate node cn_k to its subsequent context-matching nodes cn_{k++}. The next step identifies the discovered nodes dn and adds them as first member of the discovered sequence set DS. There are two differences in the expansion of discovered sequences in UCP and OCP. Firstly, the *search space* in which the mapping graph node will searched and, secondly, the introduction of an *unidentified-nodes list (ul)* which records the unidentified nodes of a candidate sequence.

Listing 3.1: UCP Discovery Algorithm

```
Input: Graph (G), Min. Pattern Support (min_sup), Min.
    Pattern Length (min_len), Max. 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** ($DS.size \geq min\_sup$) **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: $nodeFound \leftarrow unIdentifiedNodes(ul, ds)$
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**

Before the expansion process on any discovered node starts, the search space (i.e. range of graph nodes in which node will be searched) has to be set. It is described using two integer variables *start_range* (r_s) and *end_range* (r_e), where r_s and r_e represent the node ids of the start and end graph nodes of search space. The search space can be given as $r_s = min(id) - x - 1$ and $r_e = max(id) + x + 1$.

Values $min(id)$ and $max(id)$ are the minimum and maximum id values of the graph nodes in the discovered sequence ds in a particular iteration. 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 ($x = 1$) and a discovered sequence ds that contains two graph nodes $ds = \{n_9, n_{11}\}$ in a particular iteration, the search space (in which the next discovered node will be searched) is $n_7 - n_{13}$. As the algorithm scans the whole graph only once (i.e. in step 7 of algorithm 4.3 to get the discovered node set) and narrows the search space later, the search space defining technique improves the performance of the algorithm.

The unidentified nodes list (ul) records all candidate nodes that are not matched in the ds expansion process. If a new node is added to a discovered sequence, the sequence will be converted into ascending form (based on their id values) and the search space is reset. If there is no match and ds is not expanded, the respective candidate node is added to ul. Once the discovered sequence ds is expanded, an iteration is applied on ul to search the unidentified nodes in the updated search space. If an unidentified candidate node is matched to a discovered node in the updated search space, the node is added to 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 recalculated.

At the end of the expansion of a discovered sequence, if the length of an expanded discovered sequence is less than the minimum pattern length threshold, it must be discarded from the set of discovered sequences. Then, all discovered sequences whose length is less than the length of a candidate sequence are discarded. As a last step, the candidate sequence along with discovered sequences are saved as a change pattern in result list S and the algorithm goes back to step 1 and selects the next graph node as a candidate.

Listing 3.2: setSearchSpace()

Input: Graph (G), Discovered Sequence (ds), Maximum n-distance (x)
Output: Updated search space (r_s - r_e)
1: $n_1 \leftarrow getFirstNodeOfSequence(ds)$
2: $n_2 \leftarrow getLastNodeOfSequence(ds)$
3: $r_s = n_1.getNodeID() - x - 1$
4: **if** ($r_s \leq 0$) **then**
5: $r_s = 1$
6: **end if**
7: $r_e = n_2.getNodeID() + x + 1$
8: **if** ($r_e > G.size$) **then**
9: $r_e = G.size$
10: **end if**

8 Experimental Results and Evaluation

The main concern in evaluating the layered change operator and log framework is its practical validity and the adequacy. How useful the proposed solution is and how effectively it solves the problems faced in the real world. In terms of change pattern identification algorithms, the effectiveness of the algorithms in terms of correctness and completeness are the key factors. Empirical case studies and lab-based experiments, in a controlled environment, can be used to evaluate any sys-

tem and to accept or reject the effectiveness of methods, techniques or tools [9]. We selected an *empirical case study* and *controlled experiments* as our evaluation strategies. In Section 8.1, the user-based evaluation of the proposed change operator framework is given. In Section 8.2, the results and evaluation of our controlled experiments, being done in order to identify the composite change patterns from an atomic change log, are given. The change pattern discovery algorithms have been evaluated based on the experiments in a few domain ontologies. Rresults are given in Section 8.3.

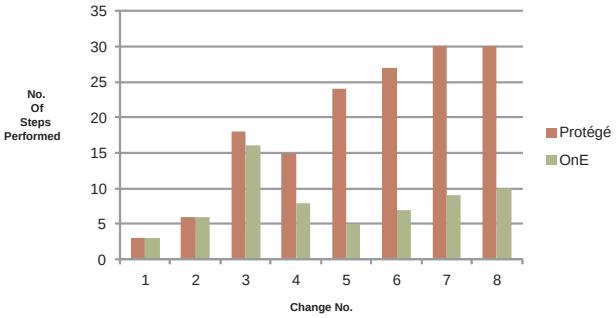

Fig. 17 Protege Vs. OnE - Number of steps performed

8.1 Layered Change Operator 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.

8.1.1 Evaluation

We involved few ontology engineers for evaluating 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 performing 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 3.

Operational cost in terms of number of steps: We evaluated the framework on the basis of the number of steps required to perform the specific changes given in Table 3. To do so, we make use of our Ontology Editor (OnE) and widely used Protégé framework. Results are given in Figure 17 in the form of bar chart. It is evident that in case of atomic level change operations, both frameworks require the same number of steps to be performed. However, usage of evolution strategies [25] 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 case of composite change operation `Merge classes` (change 4), user need to take eight (8) steps (in OnE) in comparison to fifteen (15) changes in Protégé. The biggest difference was seen in case of the `Split class` composite change where the selected strategy was to join the roles to both the newly added classes (change 6). The result is fairly understandable as in case of Protégé, users need to attach each role one after the other and hence increase the number of required steps. The more roles to be attached, the more the steps it requires. On the other hand, in case of OnE, users only need to select the appropriate evolution strategy and all roles will automatically be attached to the newly added split classes. Hence, increase or decrease of roles does not have any effect on the number of required steps.

Operational cost in terms of time: 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 4. Learning affects on the performance of different users have been considered and factored in 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 case of performing `Merge classes` (change 4) in Protégé, user need 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" reduced the time required for attaching all the roles. Similarly, in case of `split class` (change 6), by selecting evolution strategy "Attach to both classes" the user did not need to attach roles to the split classes one after the other.

8.1.2 Practical benefits

The ontology change patterns can be used as data entry forms in an existing ontology editing toolkit. These (generic/domain-specific) change patterns (along with the different defined evolution strategies) are useful for the ontology engineers to modify domain ontologies mo-

Table 3 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)

Table 4 Comparison between OnE and Protégé (min:sec)

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

re easily and more correctly. The consistency issues during ontology evolution can be resolved using evolution strategies at each layer of change operator framework. As discussed above, the usage of pattern-driven data entry forms (for performing higher-level change operations) significantly reduces the evolution effort in terms of time and manual effort. Furthermore, the change pattern data entry forms also make the evolution process intuitive and simple for a non-expert.

8.2 Composite Change Detection Algorithms

Detection of composite changes not only helps in understanding the evolution of domain ontologies, but also reduce the effort required in terms of time and consistency management. Based on the identified composite changes, more appropriate (composite level) strategies can be employed in order to keep the validity and consistency.

In this section, first we illustrate a few examples of the identified composite changes. Second, we evaluate the composite change pattern detection algorithms based on the controlled experiments and their comparison with a manual approach. At the end, we describe the learnt lessons (methodology) from the controlled experiments and their results.

8.2.1 Illustration of Results

Two examples from the identified composite changes are given in Figures 18 and 19.

The example given in Figure 18 represents an identified split change, where, *"Distribute the roles"* was the selected evolution strategy. In a previous version of the ontology V_1, concept *Student* was classified into *MSStudent*, *PhDStudent* and *UGStudent*. Thus, all the master's students (OWL:Individual), whether taught or research-based, were direct instances of concept *MSStudent*. In subsequent version of ontology V_2, in order to distinguish between research-based and course-based students of master's degree, the concept *MSStudent* is splitted into two sibling concepts (i.e., *MSByResearchStudent* and *MSTaughtStudent*). Based on the selected evolution strategy, the direct instances of the deleted concept *MSStudent* are distributed among the newly added concepts.

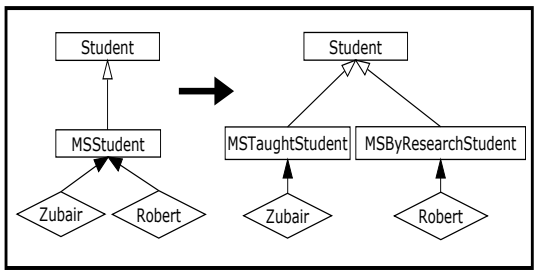

Fig. 18 Example of an Identified Composite Change

The example given in Figure 19 represents an identified pull up property change on concept *PhDStudent*; where concepts *MSByResearchStudent* and *PhDStudent* were direct subclasses of concept *student*. In a previous version of the ontology V_1, the concepts *MSByResearchStudent* and *PhDStudent* are grouped under the concept *ResearchStudent*. In this regard, the next step

was to pull up the properties to the common superclass *ResearchStudent* in the subsequent version V_2. We identified the composite changes such as "Pull up property (ResearchTrack)", "Pull up property (isSupervisorOf)" etc.

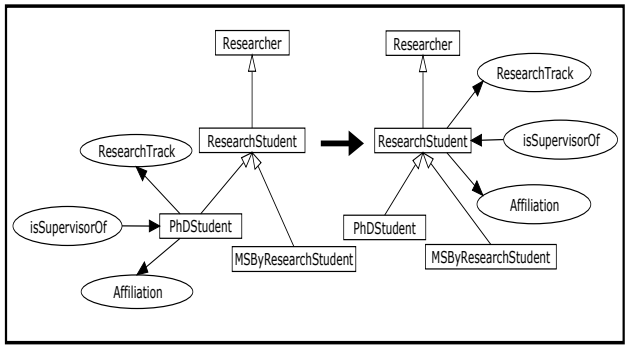

Fig. 19 Identified Composite Change- *"Pull Up Property"*

As we mentioned earlier, the semantics of any change must be captured at a higher level. Knowing that domain/range of a property p has been changed from one concept to another, let's say from x to z, certainly exhibits that the individuals of concept x and of any other concept in its subclass hierarchy, who instantiate property p, are not valid anymore. However, knowing that the domain/range of a property p has been generalised from x to z (vz. the concept x is a subclass of concept z), assures that the validity of any of the individual is not violated. Similarly, in case of specialization of domain/range of a property p from a parent concept a to its (direct) child concept b, assures that the only those individuals of concept a who instantiate property p, are not valid anymore. All other individuals of concept a and others in the subclass concept hierarchy of a (vz. the concept hierarchy starts from concept a) are still valid.

8.2.2 Evaluation

We evaluate the algorithms based on their *completeness* and *correctness*. In terms of completeness, the algorithms written to identify the composite change patterns should capture all types of available composite changes from the change log. In terms of correctness, there should not exist any false identified composite change in the result list. It is obvious that an automated solution to identify change patterns from the change log will be faster than the manual identification of change pattern; thus, reduction in *time consumption* has not been considered as an evaluation criterion, but as a benefit.

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 a small group of ontology engineers together and gave them a brief description about the domain (i.e., *university administration*), the *composite changes* and their definitions. We performed the evaluation in two steps:

- Step 1: We distributed among them the first five sessions of the ontology change log and asked them to identify the discussed composite changes from these change log sessions - (Completeness). To perform the evaluation on a small scale, we selected only six types of composite change patterns (i.e., split concept, add specialize concept, group concepts, add interior concept, pull up property and pull down property) and a small size of atomic change log (i.e., 120 atomic ontology changes).
- Step 2: At the end of step 1, we gave them 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).

Table 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 part of them can be acknowledged as a composite change pattern. The candidate change patterns identified through the manual or automated approach needs to be reviewed again by an expert ontology engineer, before confirming them as correctly identified composite change patterns.

Table 5 Comparison between Manual and Automated Composite Change Pattern Detection

	Manual	Automated
Change Log size	120 atomic changes	
Identified Change Patterns	10	11
Candidate Change Patterns	1	1
Complete Change Patterns	9	10
False Change Patterns	0	0
Missed Change Patterns	1	0
Time Taken	55 min	$<$ 1 sec

The ontology engineers were able to identify ten composite changes in comparison to the automated approach where the number of detected composite changes was eleven. It has been observed the ontology engineers were able to identify almost all the composite changes, but the main difference lies in two cases, i.e., the time taken to identify these changes (from such a

small change log) and the missing of composite change patterns having a positive *n-distance* (c.f. Section 7.1) .

- Ontology engineers took almost an hour to go through 120 atomic ontology changes and to identify correct change patterns. This result shows that identifying composite change patterns manually, on a small scale change log, is possible but yet 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). In real world case scenario, the ontology change logs are of larger size and an automated solution is a necessity there. As the size of change log increases, the time required to identify composite change pattern manually will increase intensively and using some automated approach is inevitable.
- The manual approach missed the identification of an *"add specialise concept"* composite change pattern, during to the availability of few extra change operations in between the change operations of the composite change. 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.

8.3 Change Pattern Discovery Algorithms

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 in section 4 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. In the first place, 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 pattern. For instance, a reordering of semantically equivalent operations needs to be recognised by the algorithms.

Table 6 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:Dept_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:ResGroup_i>)

Table 7 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>)

8.3.1 Illustration of Results

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 6 and 7.

The example in Table 6 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).

The example in Table 7 is a TBox-based change pattern from a software application ontology, representing the introduction of a new software activity. First, a new concept (*TargetEntity_c1*) has been added as a subclass of concept *Software:Activity*. Later, to perform this activity, a new procedure has been added as a subclass of concept *Software:Procedure* in the help infrastructure section of the ontology. Finally, the activity and the procedure to perform such an activity are linked to each other using an object property *Software:hasProcedure*.

8.3.2 Evaluation

We conducted a performance study on our case study datasets (Table 8). We utilized our algorithms to discover the domain-specific change patterns in ontology change log graphs. Given a fixed user input value for minimum pattern length and minimum pattern support, we executed the algorithms, varied the node-distance value and evaluated their results.

OCP 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 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 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) converting a sequence to ascending form in each iteration. UCP is more efficient in terms of numbers of discovered patterns. It discovers more change patterns compared to OCP (9:5). Similarly, in terms of the size of maximal patterns, UCP discovers patterns of greater size than OCP.

8.3.3 Analysis of Discovered Change Patterns

In this section, we examine the practical benefits of the discovered change patterns and lessons learnt in existing real world scenario. Possible applications of our pattern discovery algorithms range from supporting the change tracking tools, identification of user's behavioural dependency and classification of users [12], change request recommendations, analysis of change patterns and discovery of causal dependencies.

1. 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.). 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 these in the form of transactions.
2. 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 in the university ontology, based on the identified *PhD Student Registration* change pattern, it can be recommended to the user to add *student id*, *email id* of the student and *assign a supervisor* to him (using object property *hasSupervisor*). Similarly in software application domain, whenever a user deletes a certain activity from the domain, the deletion of the relevant help files can also be recommended to the user.

Table 8 Comparison b/w OCP and UCP Algorithm with Minimum Pattern Support (min_sup) = 5 and Minimum Pattern Length (min_len) = 5.

	a - **OCP Algorithm**		b - **UCP Algorithm**	
Node Dist.	Patterns Found	Time (ms)	Patterns Found	Time (ms)
0	0	469	4	1359
1	3	609	7	2282
2	5	875 6	6	3906
3	5	985	8	4968
4	5	1110	8	6078
5	5	1203	9	7141

A *limitation* of our algorithms is that they cannot be applied on the change parameters which are represented as a complex expressions. Our algorithm considers all parameters as atomic classes, properties or individuals based on the OWL 2 working draft specification.

9 Conclusion

In this paper, we discussed our approach for ontology evolution as a pattern-based compositional framework. The approach focuses on a four-phase ontology change management system that performs and records changes at a higher level of granularity. We presented 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 level one and level two, other users (such as domain experts, content managers) can focus on domain-specific changes at level three. Such a layered change operator framework enables us to deal with structural and semantic changes at two separate levels without losing their interdependence. Plus, 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 once-off specification of domain-specific change patterns. The empirical study indicates that the solution is valid and adequate to efficiently 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 change, based on the actual change activity data from change log. We operationalized the identification of higher-level changes using graph-based matching and pattern discovery approaches. We noticed that 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 a user to construct customizable domain-specific data entry forms and entering the instance-level data. As the concept hierarchy as well as the description about any concept will evolve through 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" domain-specific change patterns, but can also aid in customization and editing of already available "user-defined" data entry forms. As good patterns always arise from practical experience [42], such change patterns, created in a collaborative environment, provide guidelines to ontology change management and can be used in any change recommendation system.

More research work needs to be done to address the limitations regarding the reusability of the (defined) change patterns. A highly reused 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.

Good documentation is vital for effective reuse of any framework. To address the limitations regarding documentation, 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 and change data information.

Acknowledgements This research is supported by Science Foundation Ireland (Grant 07/CE/I1142) as part of the Centre for Next Generation Localisation at Dublin City University. The authors also acknowledge the insightful comments given by the reviewers, which have greatly helped to improve the paper.

References

1. Agrawal, R., Gunopulos, D., Leymann, F.: Mining process models from workflow logs (1998)
2. Agrawal, R., Srikant, R.: Mining sequential patterns. In: Proc. of the Int. Conf. on Data Engineering. IEEE Computer Society, pp. 3–14 (1995)
3. Altschul, S., Gish, W., Miller, W., Myers, E., Lipman, D.: Basic local alignment search tool. In: Journal of Molecular Biology, vol. 215(3), pp. 403–410 (1990)
4. 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, PSI'06, pp. 55–69. Springer-Verlag (2007)
5. Cook, J., Wolf, A.: Discovering models of software prosses from event-based data. In: ACM Transactions on Software Engineering and Methodology., vol. 5(3), pp. 215–249 (1998)
6. 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. In: Technical Report Tr-96-17, Universita di Pisa, Dipartimento di Informatica. (1996)
7. Djedidi, R., Aufaure, M.A.: Onto-evoal an ontology evolution approach guided by pattern modeling and quality evaluation. In: Proceedings of the 6th international conference on Foundations of Information and Knowledge Systems, FoIKS'10, pp. 286–305. Springer-Verlag (2010)
8. Dongen, B.F.V., van der Aalst, W.M.P.: Multi-phase process mining: Building instance graphs. In: International Conference on Conceptual Modeling (ER 2004), volume 3288 of LNCS, pp. 362–376. Springer-Verlag (2004)
9. Easterbrook, S., Singer, J., Storey, M.A., Damian., D.: Selecting empirical methods for software engineering research. In: Guide to Advanced Empirical Software Engineering, pp. 285–311. Springer (2008)
10. Ehrig, H., Pfender, M., Schneider, H.: Graph grammars: an algebraic approach. In: Proc. of 14th Annual IEEE

Symposium on Switching and Automata Theory, pp. 167–180 (1973)
11. Ehrig, H., Prange, U., Taentzer, G.: Fundamental theory for typed attributed graph transformation. In: Intl. Conf. on Graph Transformation, pp. 161–177 (2004)
12. Falconer, S., Tudorache, T., Noy, N.F.: An analysis of collaborative patterns in large-scale ontology development projects. In: Proceedings of the sixth international conference on Knowledge capture, K-CAP '11, pp. 25–32 (2011)
13. Flouris, G., Manakanatas, D., Kondylakis, H., Plexousakis, D., Antoniou, G.: Ontology change: Classification and survey. In Cambridge Journal: The Knowledge Engineering Review, vol. 23(02), pp. 117–152 (2008)
14. Groner, G., Staab, S.: Categorization and recognition of ontology refactoring pattern. Tech. Rep. 09/2010, Institut WeST, Univ. Koblenz-Landau (2010)
15. Gruhn, V., Pahl, C., Wever, M.: Data model evolution as basis of business process management. In: In: Proceedings of the 14th Int. Conference on Object-Oriented and Entity-Relationship Modelling (OOER '95), pp. 270–281. Springer (1995)
16. Haase, P., Sure, Y.: User-driven ontology evolution management. State-of-the-Art on Ontology Evolution. EU IST Project SEKT Deliverable D3 1.1.b (2004)
17. Hartung, M., Gross, A., Rahm, E.: Conto-diff: generation of complex evolution mappings for life science ontologies. Journal of Biomedical Informatics (2012)
18. Heckel, R., Kuster, J.M., Taentzer, G.: Confluence of typed attributed graph transformation systems. In: In: Proc. ICGT 2002. Volume 2505 of LNCS, pp. 161–176. Springer (2002)
19. Henninger, S., Ashokkumar, P.: An ontology-based metamodel for software patterns. In: 18th Int. Conf. on Software Engineering and Knowledge Engineering (Seke), pp. 327–330 (2006)
20. Huan, J.: Graph based pattern discovery in protein structures. Ph.D. thesis, Department of Computer Science, University of North Carolina (2006)
21. 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, pp. 13–23 (2000)
22. 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, *LNCS*, vol. 5872, pp. 544–553. Springer (2009)
23. Javed, M., Abgaz, Y.M., Pahl, C.: Graph-based discovery of ontology change patterns. In: ISWC Workshops: Joint Workshop on Knowledge Evolution and Ontology Dynamics (EvoDyn), 24th October, Bonn, Germany. (2011)
24. Javed, M., Abgaz, Y.M., Pahl, C.: A layered framework for pattern-based ontology evolution. In: 3rd International Workshop Ontology-Driven Information System Engineering (ODISE), London, UK (2011)
25. Javed, M., Abgaz, Y.M., Pahl, C.: Composite ontology change operators and their customizable evolution strategies. In: ISWC Workshops: Joint Workshop on Knowledge Evolution and Ontology Dynamics (EvoDyn), 12th November, Boston, USA. (2012)
26. Kampffmeyer, H., Zschaler, S.: Finding the pattern you need: The Design Pattern Intent Ontology. In: Model Driven Engineering Languages and Systems (MODELS), pp. 211–225 (2007)
27. Klein, M.: Change management for distributed ontologies. Ph.D. thesis, Vrije University Amsterdam (2004)
28. Kondylakis, H., Plexousakis, D.: Ontology evolution: Assisting query migration. In: P. Atzeni, D. Cheung, and R. Sudha (Eds.), *LNCS*, vol. 7532, pp. 331–344. Springer (2012)
29. Konstantinidis, G., Flouris, G., Antoniou, G., Christophides, V.: A formal approach for rdf/s ontology evolution. In: Proceedings of the 2008 conference on ECAI 2008: 18th European Conference on Artificial Intelligence, pp. 70–74. IOS Press (2008)
30. Kuramochi, M., Karypis, G.: Frequent subgraph discovery. In: 1st IEEE Conference on Data Mining, pp. 313–320 (2001)
31. Li, C., Wang, J.: Efficiently mining closed subsequences with gap constraints. In: Proc. SIAM Int. Conf. on Data Mining (SDM'08), USA, pp. 313–322 (2008)
32. Liang, Y., Alani, H., Shadbolt, N.: Ontology change management in protégé. In: Proceedings of AKT DTA Colloquium, Milton Keynes, UK (2005)
33. Noy, N.F., Klein, M.: Ontology evolution: Not the same as schema evolution. In: Journal of Knowledge and Information Systems, vol. 6(4), pp. 328–440 (2004)
34. Palma, R., Haase, P., Corcho, O., Gomez-Perez, A.: Change representation for owl 2 ontologies. In: Proceedings of the sixth international workshop on OWL: Experiences and Directions (OWLED) (2009)
35. Papavassiliou, V., Flouris, G., Fundulaki, I., Kotzinos, D., Christophides, V.: On detecting high-level changes in RDF/S KBs. In: 8th International Semantic Web Conference, *LNCS*, vol. 5823, pp. 473–488. Springer (2009)
36. Pedrinaci, C., Domingue, J.: Towards an ontology for process monitoring and mining. In: Proceedings of the Workshop on Semantic Business Process and Product Lifecycle Management (2007)
37. Plantevit, M., Laurent, A., Laurent, D., Teisseire, M., Choong, Y.W.: Mining multidimensional and multilevel sequential patterns. In: ACM Transactions on Knowledge Discovery from Data, Article 4, vol. 4(1) (2010)
38. Plessers, P., De Troyer, O.: Ontology change detection using a version log. In: 4th International Semantic Web Conference, pp. 578–592. Springer (2005)
39. Plessers, P., De Troyer, O., Casteleyn, S.: Understanding ontology evolution: A change detection approach. Web Semantics: Science, Services and Agents on the World Wide Web. **5**(1), 39–49 (2007)
40. Qin, L., Atluri, V.: Evaluating the validity of data instances against ontology evolution over the semantic web. Information and Software Technology. **51**(1), 83–97 (2009)
41. 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, ISWC'10, pp. 647–662. Springer-Verlag (2010)
42. Schmidt, D., Fayad, M., Johnson, R.: Software patterns. In: Communications of the ACM, Special Issue on Patterns & Pattern Lang., vol. 39(10), pp. 37–39 (1996)
43. Stefanowski, J.: Algorithms for context based sequential pattern mining. In: Fundamenta Informaticae, vol. 76(4), pp. 495–510 (2007)
44. Stojanovic, L.: Methods and tools for ontology evolution. Ph.D. thesis, University of Karlsruhe (2004)
45. Stojanovic, L., Maedche, A., Motik, B., Stojanovic, N.: User-driven ontology evolution management. In: Proceedings of the 13th International Conference on Knowl-

edge Engineering and Knowledge Management. Ontologies and the Semantic Web, EKAW '02, pp. 285–300. Springer-Verlag (2002)
46. Stojanovic, L., Maedche, A., Stojanovic, N., Studer, R.: Ontology evolution as reconfiguration-design problem solving. Proceedings of the 2nd international conference on Knowledge capture (2003)
47. Weijters, A.J.M.M., van der Aalst, W.M.P.: Process mining discovering workflow models from event-based data. In: Proceedings of the ECAI Workshop on Knowledge Discovery and Spatial Data, pp. 283–290 (2001)
48. Wen, L., Wang, J., van der Aalst, W.M.P., Huang, B., Sun, J.: Mining process models with prime invisible tasks. **69(10)**, 999–1021 (2010)
49. Yan, X., Han, J.: gspan: Graph-based substructure pattern mining. In: IEEE International Conference on Data Mining, pp. 721–724 (2002)
50. Zablith, F.: Dynamic ontology evolution. International Semantic Web Conference (ISWC) Doctoral Consortium, Karlsruhe, Germany (2008)
51. Zhu, X., Wu, X.: Mining complex patterns across sequences with gap requirements. In: 20th International Joint Conference on Artificial Intelligence, pp. 2934–2940 (2007)