# Graph-based Discovery of Ontology Change Patterns

Muhammad Javed[1], Yalemisew M. Abgaz[2], Claus Pahl[3]

Centre for Next Generation Localisation (CNGL),
School of Computing, Dublin City University, Dublin 9, Ireland
{mjaved[1], yabgaz[2], cpahl[3]}@computing.dcu.ie

**Abstract.** Ontologies can support a variety of purposes, ranging from capturing 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. We investigate ontology change representation and discovery of change patterns. Ontology changes are formalised as graph-based change logs. We use attributed graphs, which are typed over a generic graph with node and edge attribution. We analyse ontology change logs, represented as graphs, and identify frequent change sequences. Such sequences are applied as a reference in order to discover reusable, often domain-specific and usage-driven change patterns. We describe the pattern discovery algorithms and measure their performance using experimental results.
**Keywords:** Pattern Discovery Algorithm, Ontology Change Representation, Ontology Evolution, Change Log Graph.

## 1 Introduction

Ontologies can support tasks ranging from capturing the conceptual knowledge to the organisation of digital content and other information artefacts. However, information systems are always subject to change and ontology change management can pose challenges. Ontologies become essential for knowledge sharing activities in dynamic information systems, in areas such as bioinformatics, educational technology systems, e-learning, indexing and retrieval.

The dynamic nature of knowledge requires ontologies to change over time. The reason for change in associated content can be change in the domain, the specification, or the conceptualization [1]. A change in an ontology may originate from a domain knowledge expert, a user of the ontology or a change in the application area [2]. While some generic ontologies (like upper ontologies) evolve at a slower pace, we have been working with non-public ontologies used to annotate content in large-scale information systems, for instance an ontology-annotated help system for a large content management architecture. In this context, changes happen on a daily basis, triggered by changes in software, its technical or domain environment. Systematic ontology change becomes here a necessity to enable controlled, accountable and predictable evolution.

The operationalisation of changes is a vital part of ontology evolution. Elementary, composite and complex change operators and detection of higher level changes have been suggested in past [3–6]. For semantically enhanced information systems, a coherent representation of ontology change is essential. We present a graph-based approach for ontology change representation. We can identify recurring change sequences from an ontology change log that captures changes at an operational level. Graphs are used to formalise change. As we are interested in changes which are applied on a particular ontology entity in different sessions, we use a gap-constrained pattern discovery approach. Some central features of our approach are:

- Fine-granular ontology change representation (RDF triples) helps sharing semantics of the data and representing intent and scope of change explicitly.
- An operational representation of the ontology changes using a graph-based formalisation. Ontology change log graphs enable efficient search, analysis and categorisation of ontology changes.
- Discovery of recurring change pattern from an ontology change log, which provides an opportunity to define reusable domain-specific change patterns that can be implemented in existing knowledge management systems.

The paper is structured as follows. We discuss ontology change representation in Section 2. Section 3 introduces pattern discovery concepts. A detailed description of change patterns discovery algorithms is given in Section 4. Experimental results and an illustration of the technique are given in Section 5. Related work is discussed in Section 6 and we end with some conclusions.

## 2 Ontology Change Representation

We studied the evolution of ontologies empirically in order to determine a fine-granular ontology change representation and to identify reusable, usually domain-specific change patterns which cannot be identified by simply exploring or querying changes. As case studies, the domains *university administration* and *software application* were looked at. The former represents an organisation involving people, organisational units and processes. The latter, based on our work with an industrial partner, takes a content-centric perspective on a software system. The application system is a content management and archiving system there.

To record of applied changes, we use ontology change logs. They provide operational as well as analytical support in the evolution process. Change logs are also used to keep the evolution process transparent and centrally manageable. It captures all the changes applied to entities of the ontology.

A graph-based formalisation is an operational representation for the ontology changes. Graphs enable efficient search and analysis and can also communicate information visually. We followed the idea of attributed graphs. Graphs with node and edge attribution are typed over an attribute type graph (ATG) [7]. Attributed graphs 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

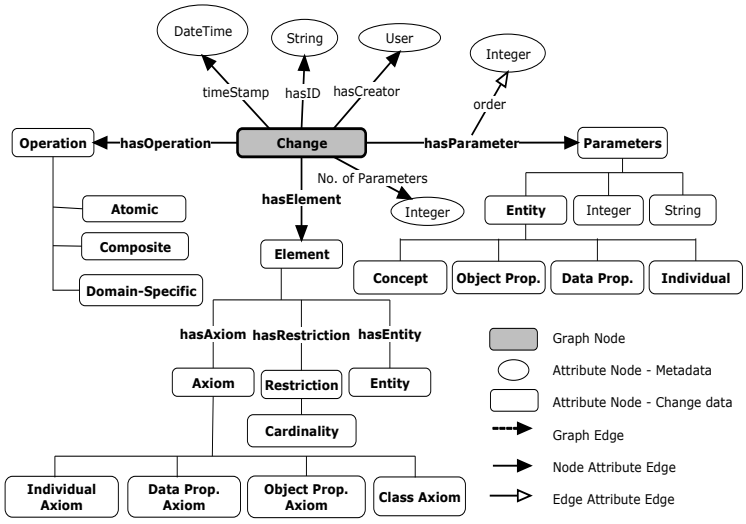


**Fig. 1.** Attribute Type Graph (ATG) for an Ontology Change

1). Based on the idea of attribute type graphs, a 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 an atomic change).
- $N_A = \{n(a_i)|i = 1, \ldots, q\}$ is the set of attribute nodes. Attribute nodes are of two types, i) attribute nodes which symbolize the metadata, e.g. Id, user, time of change operation, and ii) attribute nodes which symbolize the change data (and its subtypes), e.g. operation, axiom, target entity.
- $E_G = \{e(g_i)|i = 1, \ldots, p-1\}$ is the set of graph edges which connects two graph nodes $n(g)$.
- $E_{NA} = \{e(na_i)|i = 1, \ldots, r\}$ is the set of node attribute edges which joins a graph node $n(g)$ to an attribute node $n(a)$.
- $E_{EA} = \{e(ea_i)|i = 1, \ldots, s\}$ is the set of edge attribute edges which joins a node attribute edge $e(na)$ to an attribute node $n(a)$.

An attributed graph (AG) typed over an ATG is given in Figure 2. The types defined on the nodes can be given as $t(Add) = Operation$, $t(classAssertion) = ClassAxiom$, $t(John) = Individual$ and $t(PhD\_Student) = Concept$. The attributed graph actually represents a single ontology change operation where graph node $n(g)$ is the central part of AG. Such graph nodes are linked to each other using graph edges $e(g)$ to represent a complete ontology change log graph.

## 3 Pattern Discovery in Ontology Change Logs

Good patterns always arise from practical experience [8]. Change patterns, created in a collaborative environment, provide guidelines to ontology change man-

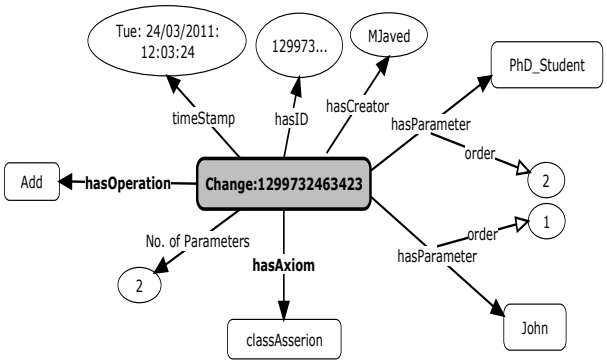


**Fig. 2.** Attributed Graph typed over ATG (Add classAssertion (John, PhD_Student))

agement and has important implications for ontology-development tools [9]. Identifying recurring changes define reusable, often domain-specific change patterns. The aim is the discovery of usage-driven change patterns in order to support the pattern-based ontology evolution.

In order to conceptualise the changes, we constructed a static *metadata ontology* by looking into concrete structure of OWL-DL syntax-based domain ontologies. The metadata ontology represents different categories of ontology changes based on our layered change operator framework [10], types of ontology elements (such as concept, axioms, restriction etc.) and other concepts such as change, users, timestamp etc. Each instance of the change log is of type *Change*, available in the metadata ontology. We used an RDF triple-store to record the change log, domain ontologies and static metadata ontology. Our ontology editing framework (OnE) offers a graph API, which is used for generating graphs from change log, based on SPARQL queries. We captured more than 500 ontology changes from the university ontology change log (>5000 log triples).

The change log graph allows us to identify and classify frequent changes that occur in 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.

While patterns are sometimes used in their exact form, often more flexibility is needed. Users often use different orderings of change operations to perform the same (semantically equivalent) change at different times. To capture semantically equivalent, but operationally different patterns, we introduce a metric that captures a node gap between two adjacent graph nodes in a sequence called *sequence gap* or *n-distance*. It refers to the distance between two adjacent graph nodes in a change log graph. This will help us to more flexible pattern notions. We merge different types of patterns into two basic subdivisions, i.e. *ordered change patterns (OP)* and *unordered change patterns (UP)*. The instances of ordered change patterns comprise change operations in exact same sequential order from change log graph. Thus, such complete (OCP) or partial (OPP) patterns may have only a positive node distance value, starting from zero to user given

threshold ($x$). The instances of unordered change patterns comprise change operation which may or may not be in exact same sequential order from 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.

We considered identifying recurring sequenced change operations from a change log as a problem of recognition of frequent pattern in a graph. We describe two key metrics by introducing the following definitions. First, the *pattern support* of a pattern $p$ is the number of occurrences of such 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)*. Second, 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)*.

A set of candidate pattern sequences $CP$, to be considered as a *change pattern* $P$, must meet the following criteria:

- The length of the each candidate pattern sequence $cp$ must be equal to or greater than the threshold value set by the minimum pattern length $len(cp) \geq min\_len(p) : cp \in CP$.
- The support for a candidate pattern $cp$ in a change log graph $G$ must be above the threshold value of minimum pattern support of a pattern $sup(cp) \geq min\_sup(p) : cp$.

Each candidate pattern sequence $cp$ must also reach a consistent ontology state. There may be internal graph nodes in $cp$, reaching non-consistent states, however, at the end of the sequence the ontology must be back in a consistent state.

## 4 Pattern Discovery Algorithms

The discovery of 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 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.

**OCP Discovery Algorithm.** To discover ordered complete change patterns (OCP), the identified sequences are of 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 particular graph node and search the similar sequences within the graph G. The *OCP* algorithm is defined in algorithms 4.1 – 4.2. 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. If the candidate node $cn_k$ contains only one parameter, the parameter is considered as *target entity*. If node $cn_k$ contains two parameters, the first parameter is considered as *target entity* and the second parameter is considered as the *primary context* of the candidate node. Once the candidate node and its target entity are captured, an iterative process of expansion of candidate node $cn_k$ to its adjacent nodes $cn_{k++}$ starts and continues until more extensions are not possible (i.e. adjacent nodes 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*).

**Algorithm 4.1** Ordered Complete Change Pattern Discovery Algorithm

**Input:** Graph ($G$), Minimum Pattern Support ($min\_sup$), Minimum Pattern Length ($min\_len$), Maximum n-distance ($x$)
**Output:** Set of Domain-Specific Change Patterns ($S$)

1: **for** $i = 0$ to $N_G.size$ **do**
2: $k = 0$
3: $cs \leftarrow GenerateCandidateSequence(n(g_i))$
4: **if** ($cs.size < min\_len$) **then**
5: go back to step 1.
6: **end if**
7: $DN \leftarrow DiscoverMatchingNodes(cn_k)$
8: $DS \leftarrow DN$
9: **if** ($DS.size < min\_sup$) **then**

```
10:       go back to step 1.
11:    end if
12:    while (DS.size ≥ 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 of the minimum pattern length, the next step is to search for the matching nodes (i.e. discovered nodes $dn$) of the same type as 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 $x$).

**Algorithm 4.2** Method: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 extension of that node is not possible or the expansion has reached to 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.

**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 algorithms 4.3 – 4.4.

Like OCP, UCP iterates over each graph node and selects it as a candidate node ($cn_k$). An iteration process 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 is to identify the discovered nodes $dn$ and adding them as the first member of the discovered sequence set $DS$. There are two main 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.

**Algorithm 4.3** Unordered Complete Change Pattern Discovery Algorithm

**Input:** Graph ($G$), Minimum Pattern Support ($min\_sup$), Minimum Pattern Length ($min\_len$), Maximum n-distance ($x$)
**Output:** Set of Domain-Specific Change Patterns ($S$)

```
1: for i = 0 to N_G.size do
2:    k = 0
3:    cs ← 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:       setSearchSpace(ds)
16:       a ← searchInSpace(ds, cn_k++, t)
17:       if (found) then
18:         ds ← a
19:         ascendSequence(ds)
20:         setSearchSpace(ds)
21:         if (!ul.isEmpty()) then
22:           nodeFound = true
23:           while (!ul.isEmpty() && nodeFound) do
24:             nodeFound ← searchUnidentifiedNodes(ul, ds)
25:             ascendSequence(ds)
26:             setSearchSpace(ds)
27:           end while
28:         end if
29:       else
30:         ul ← 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 ← (ds + cs)
43:   S ← 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, then 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 process is applied on $ul$ to search the unidentified nodes in the updated search space. If an unidentified candidate node is found and matched (to a discovered node) in the updated search space, the node 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 the result list $S$ and the algorithm goes back to step 1 and selects the next graph node as a candidate node.

**Algorithm 4.4** Method: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: $\quad r_s = 1$
6: **end if**
7: $r_e = n_2.getNodeID() + x + 1$
8: **if** ($r_e > G.size$) **then**
9: $\quad r_e = G.size$
10: **end if**

## 5 Experimental Results and Illustration

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 and consistency management.

All experiments with change pattern discovery algorithms have been done on a 3.0 GHz Intel Core 2 Duo CPU with 3.25 GB of RAM, running MS Windows XP. 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. 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 1 and 2.

### 5.1 Pattern Discovery Example

Elsewhere [10, 11], we presented pattern-based ontology change operators and motivated the benefits of pattern-based change management where patterns are usually domain-specific compositions of change operators. Our work here can be utilised 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.

Change pattern discovery shall now be illustrated using a simple example. The example in Table 1 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).

**Table 1.** ABox-based change pattern (extracted from University Ontology)

| Id | Change Operations |
|---|---|
| 17 | Add individual <TargetEntity_i> |
| 18 | Add classAssertion <TargetEntity_i> <Univ:PhD_Student> |
| 19 | Add objectPropertyAssertion <TargetEntity_i> <Univ:isStudentOf> <Univ:Department_i> |
| 20 | Add dataPropertyAssertion <TargetEntity_i> <Univ:StudentID> <xsd:int> |
| 21 | Add dataPropertyAssertion <TargetEntity_i> <Univ:EmailID> <xsd:string> |
| 22 | Add objectPropertyAssertion <TargetEntity_i> <Univ:hasSupervisor> <Univ:Faculty_i> |
| 23 | Add objectPropertyAssertion <TargetEntity_i> <Univ:MemberOf> <Univ:ResearchGroup_i> |

The example in Table 2 is a TBox-based change pattern from 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 activity are linked to each other using an object property *Software:hasProcedure*.

**Table 2.** TBox-based change pattern (extracted from Software Ontology)

| Id | Change Operations |
|---|---|
| 32 | Add concept <TargetEntity_c1> |
| 33 | Add subclassOf <TargetEntity_c1> <Software:Activity> |
| 34 | Add concept <TargetEntity_c2> |
| 35 | Add subclassOf <TargetEntity_c2> <Software:Procedure> |
| 36 | Add isDomainOf <TargetEntity_c1> <Software:hasProcedure> |
| 37 | Add isRangeOf <TargetEntity_c2> <Software:hasProcedure> |

### 5.2 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. Later, we compare the two types of algorithms and outline the known limitations.

**Tools Support and Practical Benefits:** The core benefits of discovery of ontology change patterns are *change impact determination* and *cost reduction* (in terms of time and consistency management). The possible applications of our change pattern discovery algorithms range from supporting the change tracking tools, identification of user's behavioural dependency & classification of users [9], change request recommendations, to the analysis of change patterns and discovery of causal dependencies.

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

*Change Request Recommendation:* The identified change patterns can also be used for change request recommendations. For example, whenever a user adds a new PhD student 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 certain activity from the domain, deletion of the relevant help files can also be recommended to the user.

*Discovery of Causal Dependencies:* Discovered change patterns benefited us in terms of identification of correlations and the causal dependencies (which can be integrity constraints) across the ontology taxonomy. An example of a captured causal dependency in university ontology is the *introduction of a new course*. Whenever a new course is introduced, new subjects (and subject codes) have to be introduced, course-related books have to be provided by the library, or new vacancies might have to be advertised. This is a domain-specific pattern for organisations where the introduction of a new product or service entails for instance further personnel changes. Causal dependencies across the ontology subsumption hierarchy can easily be overlooked. Pattern discovery can identify successful changes based on these dependencies. These are association rules that capture non-obvious relationships.

**Lesson learnt from Real-World Scenario:** A number of lessons were learnt by running our algorithms on a real-world software application ontology. We compared different versions of the software application (a document archiving system) and analysed how software application evolves from one version to another. Based on pattern discovery mechanism, the application is represented in a form of layered framework, consisting of *explanatory*, *functional* and *system implementation* layers. The implementation layer consists of the key components of the software application. The functional layer represents the management activities of each software component. The explanatory layer contains the software application help files. These help files provide details about software component and the content management activities. We ran our change pattern discovery algorithms and identified the changes at each layer of software application. At the implementation layer, the most frequent patterns includes *addition*, *deletion*, *splitting* and *merging* of different software components etc. At the functional layer, *addition* or *deletion* of archiving activities and gui items are the most common actions. The change patterns at the explanatory layer are generally linked to the changes at implementation and functional layers. For example, whenever pattern *add new component* is implemented, the relevant description files has been added (*add new description files*) at explanatory layer. Similarly, whenever a new activity has been added at functional layer, the relevant procedural files has been added (*add new procedural files*) at explanatory layer.

**Comparison between OCP and UCP:** We conducted a performance study on our case study datasets and looked at their effectiveness for pattern discovery. 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. Unordered change operations make the UCP algorithm complex 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. However, UCP is more efficient in terms of numbers of discovered patterns. Similarly, in terms of the size of maximal patterns, UCP discovers patterns of greater size than OCP. The detailed comparison table between two algorithms is available on our web[1].

**Limitations:** One of our algorithms limitations is that it cannot be applied on the change parameters which are represented as complex expressions. Our algorithm considers all parameters as atomic classes, properties or individuals. Secondly, our algorithms used an assumption, i.e. the target entity is always the first parameter of any ontology change operations. This assumption does not suit, for example in case of inverse properties, such as *hasSupervisor* and *isSupervisorOf*. In our future work, deep comparison of ontology change operations will be made in order to identify the target entity (context) in relation to identified change operations of a sequence.

## 6 Related Work

The mining of sequential patterns was first proposed by Agrawal and Srikant in [12]. Since then, many sequential pattern mining algorithms often based on specific domains [13–17] have been suggested.

In the domain of DNA or protein sequences, BLAST [17] is one of the most famous 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 [16], 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 [18–21]. In [18], the author propose an apriori-based algorithm, called AGM, to discover frequent substructures. In [19], the authors proposed the gSpan (graph-based Substructure pattern mining) algorithm for mining frequent closed graphs and adopted a depth-first search strategy. In constrast 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 [20] algorithm. The performance study shows that the gSpan outperforms FSG algorithm and is capable of mining large frequent subgraphs. The Fast Frequent Subgraph Mining (FFSM) algorithm [21] 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 [19] and FSG [20] 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).

[1] `www.computing.dcu.ie/~mjaved/phdwork.html`

We adopted ideas from sequential pattern discovery approaches in other domains, such as sequence-independent structure pattern [21], gap-constraint [13] 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 [5, 6]. 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. This requires the identification of equivalency between unordered change sequences.

## 7 Conclusion

Ontologies will continue to evolve. Scalability beyond manual evolution and change support is required for both stand-alone ontologies that live over very long periods of time and also ontologies used in conjunction with other information systems that themselves can trigger change. Supporting this process by automated management of change and its analysis is a solution.

The presented work here continues our previous research [10, 11] by adding operational ontology change representation and pattern discovery algorithms. In this paper, we proposed a graph-based formalism for ontology change representation using attributed graphs, which are typed over a generic attribute type graph. We studied the change log graphs empirically and observed that the users perform similar groups of change operations multiple times as a combination of single atomic change operations. The contribution of our work are the algorithms for the discovery of domain-specific change patterns to support pattern-based ontology evolution. On the basis of our empirical study, we identified two fundamental types of change patterns i.e., *Ordered Change Patterns (OP)* and *Unordered Change Patterns (UP)*.

The experimental results signify that the solution is valid and adequate to efficiently handle pattern-based ontology evolution. Our approach benefits in different ways. First, the discovered patterns are suitable to provide pattern-level ontology change support. Second, as the discovered patterns are based on the actual frequent changes applied by the users, patterns assist in a structured pattern-based evolution process. Third, domain-specific change patterns can be shared among other ontologies that have similar conceptualizations.

The next step would clearly be to take the discovered patterns and try to identify them in other change logs in order to analyse the use of patterns. This would require our discovery algorithms to be revised accordingly. We have mentioned the partial patterns types OPP and UPP in section 3. Partial patterns usage is more a concern for this pattern identification context (and was not dealt with here for that reason).

## Acknowledgement

This research is supported by the Science Foundation Ireland (Grant 07/CE/I1142) as part of the Centre for Next Generation Localisation at Dublin City University.

## References


1. Noy, N.F., Klein, M.: Ontology evolution: Not the same as schema evolution. In: Journal of Knowledge and Information Systems. Volume 6(4). (2004) 328–440
2. Liang, Y., Alani, H., Shadbolt, N.: Ontology change management in protégé. In: Proceedings of AKT DTA Colloquium, Milton Keynes, UK. (2005)
3. Stojanovic, L.: Methods and tools for ontology evolution. PhD thesis, University of Karlsruhe (2004)
4. 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)
5. 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. Volume 5823 of LNCS., Springer (2009) 473–488
6. Groner, G., Staab, S.: Categorization and recognition of ontology refactoring pattern. Technical Report 09/2010, Institut WeST, Univ. Koblenz-Landau (2010)
7. Ehrig, H., Prange, U., Taentzer, G.: Fundamental theory for typed attributed graph transformation. In: Intl. Conf. on Graph Transformation. (2004) 161–177
8. Schmidt, D., Fayad, M., Johnson, R.: Software patterns. In: Communications of the ACM, Special Issue on Patterns & Pattern Lang. Volume 39(10). (1996) 37–39
9. 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 (2011) 25–32
10. Javed, M., Abgaz, Y.M., Pahl, C.: A pattern-based framework of change operators for ontology evolution. In: On the Move to Meaningful Internet Systems: OTM Workshops. Volume 5872 of LNCS., Springer (2009) 544–553.
11. 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)
12. Agrawal, R., Srikant, R.: Mining sequential patterns. In: Proc. of the Int. Conf. on Data Engineering. IEEE Computer Society. (1995) 3–14
13. Li, C., Wang, J.: Efficiently mining closed subsequences with gap constraints. In: Proc. SIAM Int. Conf. on Data Mining (SDM'08), USA. (2008) 13–322
14. 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. Volume 4(1). (2010)
15. Stefanowski, J.: Algorithms for context based sequential pattern mining. In: Fundamenta Informaticae. Volume 76(4). (2007) 495–510
16. Zhu, X., Wu, X.: Mining complex patterns across sequences with gap requirements. In: 20th International Joint Conference on Artificial Intelligence. (2007) 2934–2940
17. Altschul, S., Gish, W., Miller, W., Myers, E., Lipman, D.: Basic local alignment search tool. In: Journal of Molecular Biology. Volume 215(3). (1990) 403–410
18. 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. (2000) 13–23
19. Yan, X., Han, J.: gspan: Graph-based substructure pattern mining. In: IEEE International Conference on Data Mining. (2002) 721–724
20. Kuramochi, M., Karypis, G.: Frequent subgraph discovery. In: 1st IEEE Conference on Data Mining. (2001) 313–320
21. Huan, J.: Graph Based Pattern Discovery in Protein Structures. PhD thesis, Department of Computer Science, University of North Carolina (2006)