Ontology Evolution Patterns Based on Hierarchical Versioning

Sören Auer and Heinrich Herre
University of Leipzig, 04109 Leipzig, Germany,
auer@informatik.uni-leipzig.de,
WWW home page: http://www.informatik.uni-leipzig.de/~auer/

Abstract  We present an approach to support the evolution of online, distributed, reusable, and extendable ontologies based on the RDF data model. The approach works on the basis of atomic changes, basically additions or deletions of statements to or from an RDF graph. Such atomic changes are aggregated to more complex changes, resulting in a hierarchy of changes, thus facilitating the human reviewing process on various levels of detail. These derived compound changes may be annotated with meta-information and classified as ontology evolution patterns. The introduced ontology evolution patterns in conjunction with appropriate data migration algorithms enable the automatic migration of instance data in distributed environments.

1 Introduction

The goal of the envisaged next generation of the Web (called Semantic Web [3]) is to smoothly interconnect personal information management, enterprise application integration, and the global sharing of commercial, scientific, and cultural data\(^1\). In this vision, ontologies play an important role in defining and relating concepts that are used to describe data on the web [8]. In a distributed, dynamic environment such as the Semantic Web, it is further crucial to keep track of changes in its documents to ensure the consistency of data, to document their evolution, and to enable concurrent changes.

Versioning and revision control mechanisms have already been developed and successfully applied in areas such as software engineering, databases, and web publishing. In software engineering versioning is used to track and provide controls over changes to a project’s source code. In database systems versioning is usually provided by a database log, which is a history of actions executed by a database management system. For web publishing the Web-based Distributed Authoring and Versioning (WebDAV) [14] standard was released as an extension to the Hyper Text Transfer Protocol (HTTP) with the intention of making the World Wide Web a readable and writable medium.

For revision control of semantic web data, unfortunately these developed technologies are insufficient. In software engineering and web publishing revision

\(^1\) http://www.w3.org/2001/sw/Activity
control is based on unique serializations, enabled by their data models. Such unique serializations are not available for semantic web data, usually consisting of unordered collections of statements. Database logs on the other hand cope with a multitude of different interrelated objects of their data model (e.g. databases, tables, rows, columns/cells) in comparison with only graphs and statements of semantic web data. Moreover the tracking and formal description of changes on higher conceptual levels is not supported.

In this paper, we present an approach for the versioning of online, distributed, reusable, and extendable ontologies based on the RDF data model, supporting ontology evolution. Under ontology versioning we understand to keep track of different versions of an ontology and possibly to allow branching and merging operations. Ontology evolution additionally shall identify and formally represent the conceptual changes leading to different versions and branches. On the basis of this information, ontology evolution should support the migration of data agreeing to a distinct ontology version.

To present our approach we recall for short some used RDF data model concepts in Section 2. The approach works on the basis of atomic changes which are determined by additions or deletions of certain groups of statements to or from an RDF graph. Such atomic changes are aggregated to more complex changes, resulting in a hierarchy of changes, thus facilitating the human reviewing process on various levels of detail (Section 3). The derived compound changes may be annotated with meta-information such as the user executing the change or the time when the change occurred. We present a simple OWL ontology capturing such information which enables the distribution of change sets (Section 5). Assuming that there will be no control of evolution, it must be clarified which changes are compatible with a concurrent branch of the same root ontology. We present a compatibility concept for applying a change to an ontology on the level of statements (Section 4). To enabling smooth ontology evolution we introduce the ontology evolution patterns and give examples for corresponding data migration algorithms (Section 6). We further give account of the successful implementation of the approach in Powl, a development framework for Semantic Web Applications, we summarize related work and give an outlook on planned directions for future work (Sections 7).

2 Evolution of Ontologies

There is a rapidly growing body on ontologies in information systems which has been boosted by the vision of the Semantic web. Accordingly, there are strong ongoing efforts in the development of ontologies in the sense of Gruber, i.e. as sharable conceptual specifications. Likewise, research in formal tools and techniques supporting the ontology development (or ontological engineering) is very active. The notion of ontology has been applied in various areas and is used currently in a very broad sense. In general, a particular ontology is understood to be a conceptual description of a given domain which can be accepted and reused in all information systems referring to this domain. In the broad sense
even terminology systems including natural language descriptions are considered as ontologies. Ontologies in a more specific sense are formal specifications of conceptualizations which are presented by expressions of a formal representation language. In the sequel we use the notion of ontology in this restricted sense.

2.1 Ontologies

Formal ontology is the science which is concerned with the systematic development of formal theories of forms, modes and views of being of different levels of abstraction and granularity. On the most general level of abstraction formal ontology is concerned with the kinds, modes, views, and structures which apply to every area of the world. we call this level of description General Ontology, in contrast to the various Domain Ontologies which are applicable to more restricted fields of interest. We assume that every domain-specific ontology must use as a framework some general ontology, sometimes called top-level ontology, which describes the most general categories of the world. Ontologies inevitably change over time. Since every ontology is considered as a set KB of formal expressions, as a formal knowledge system, we may ask which kinds of changes of KB are in principle possible. Since KB refers to a domain D of the world the domain D may change which may imply a change of KB; we call these changes of KB domain-based changes. Furthermore, the conceptualization - the ontological vocabulary on which KB is based - may change, we call them conceptual changes. Finally, KB may be formally changed without referring to domain changes or conceptual changes, we call them formal changes. There might be formal changes which are not domain-based changes or conceptual changes. An example is the removal of an inconsistency which is not caused by the change of the domain or of the conceptualization. In the sequel we restrict ourself to the analysis of changes pertaining to ontologies which are based on the RDF model.

2.2 RDF Ontologies

We consider a set of RDF-statements as an ontology which may be founded on an abstract core ontology. An abstract core ontology describes the most general basic entities and basic relations of ontologies, as for example class/category, individual, property, relation etc. In the sequel we use the semantics of RDF which is based on model theory as expounded in [6]. The abstract core ontology - in this case - is set theory. A RDF-statement (S,P,O) is interpreted, then, as an atomic formula P(S,O), where P is interpreted as binary relation and S and O denote certain entities of a domain. Under this interpretation a set of RDF-statements is understood as a conjunction of atomic formulas whose blank nodes are existentially quantified variables. These formulas exceed the usual FOL because it is admitted that relations are arguments of relations.

Let us now recall some preliminary definitions from [9]. Some of the main building blocks of the semantic web paradigm are Universal Resource Identifier (URI) and their RDF counterparts URI References, whose quite technical definitions we omit here.
Definition 1 (Literal) A Literal is a string combined with either a language identifier (plain literal) or a datatype (typed literal).

Definition 2 (Blank Node) Blank Nodes are identifiers local to a graph. The set of Blank Nodes, the set of all URI references, and the set of all literals are pairwise disjoint. Otherwise, the set of blank nodes is arbitrary.

Definition 3 (Statement) A Statement is a triple \((S, P, O)\), where
- \(S\) is either a URI reference or a blank node (Subject).
- \(P\) is a URI reference (Predicate).
- \(O\) is either a URI reference or a literal or a blank node (Object).

Definition 4 (Graph) A Graph is a set of statements.

The set of nodes of an graph is the set of subjects and objects of triples in the graph. Consequently the blank nodes of a graph are the members of the subset of the set of nodes of the graph which consists only of blank nodes.

Definition 5 (Graph Equivalence) Two RDF graphs \(G\) and \(G'\) are equivalent if there is a bijection \(M\) between the sets of nodes of the two graphs, such that:

1. \(M\) maps blank nodes to blank nodes.
2. \(M(\text{lit}) = \text{lit}\) for all literals \(\text{lit}\) which are nodes of \(G\).
3. \(M(\text{uri}) = \text{uri}\) for all URI references \(\text{uri}\) which are nodes of \(G\).
4. The triple \((s, p, o)\) is in \(G\) if and only if the triple \((M(s), p, M(o))\) is in \(G'\).

With this definition, \(M\) shows how each blank node in \(G\) can be replaced with a new blank node to give \(G'\). With respect to the model-theoretic semantics equivalence of graphs implies that their associated (existentially quantified) formulas are logically equivalent. The above mentioned bijection on the blank nodes corresponds to a special case of renaming variables in a logical framework.

3 Changes and Revisions of Ontologies

Based on the definitions in the preceding section we want to elaborate the notion of possible changes on graphs. Ognyanov and Kiryakov identify in [11] RDF statements to be the smallest manageable piece of knowledge. They justify their view by the fact that there is no way to add, remove, or update a resource or literal without changing at least one statement, while the opposite does not hold. We adopt their view but require the smallest manageable pieces of knowledge to be somehow closed regarding the usage of blank nodes. Moreover we want to be able to construct larger changes out of smaller ones, and since the order of additions and deletions of statements to a graph may matter, we distinguish between Positive and Negative Atomic Changes.
Definition 6 (Atomic Graph) A graph is atomic if it may not be split into two nonempty graphs whose blank nodes are disjoint.

Obviously, a graph without any blank node is atomic if it consists of exactly one statement. Hence, any statement which does not contain a blank node as subject or object is an atomic graph. An example for an atomic graph is given in Example 1.

Definition 7 (Positive Atomic Change) A atomic graph $C_G$ is said to be an Positive Atomic Change on a graph $G$ if the sets of blank nodes occurring in statements of $G$ and $C_G$ are disjoint.

Positive atomic changes thus are basically sets of statements, that fulfill certain conditions with respect to some graph. Of course we would like to applicate them to the graph.

Positive atomic changes on a graph $G$ are not yet themselves changes of $G$, but they may be applied to $G$ to yield new graphs as a result. For this purpose we introduce a (partial) function $Apl^+(X,Y)$ whose arguments are graphs.

Definition 8 (Application of a Positive Atomic Change) Let $C_G$ be a positive atomic change on the graph $G$. Then the function $Apl^+$ is defined for the arguments $G, C_G$ and it holds $Apl^+(G, C_G) = G \cup C_G = G'$ which is symbolized by $G \xrightarrow{C_G} G'$. We say that $C_G$ is applied to the graph $G$ with result $G'$.

Definition 9 (Negative Atomic Change) A subgraph $C_G$ of $G$ is said to be a Negative Atomic Change on a graph $G$ if $C_G$ is atomic and contains all statements of $G$ whose blank nodes occur in the statements of $C_G$.

Analogously to the case of positive changes we introduce a function $Apl^-(G, C_G)$ which pertains to negative atomic changes.

Definition 10 (Application of a Negative Atomic Change) Let $C_G$ be a negative atomic change on the graph $G$. Then the function $Apl^-$ is defined for the arguments $G, C_G$ and is determined by $Apl^-(G, C_G) = G \setminus C_G = G'$ which is symbolized by $G \xrightarrow{C_G} G'$. We say that $G_C$ is applied to $G$ with result $G'$.

These definitions require changes involving blank nodes to be somehow independent from the graph in the sense that blank nodes in the change and in the (remaining) graph do not overlap. This is crucial for changes being exchangeable between different RDF storage systems, since the concrete identifiers of the blank nodes may differ. It may have the negative effect though that large subgraphs, which are only interconnected by blank nodes, have to be deleted completely and added - slightly modified - afterwards. One example are additions or deletions of elements to or from RDF collections as shown in the following example.

Example 1 (Changes of RDF collections) Consider the following list of students represented as RDF collection (blank nodes are prefixed by '_:'): 
The need of adding one student to the list will result in the deletion of the whole list first and addition of the new list thereafter.

The evolution of a knowledge base typically results in a multitude of sequentially applied atomic changes, which are usually small, and may often contain only a single statement. On the other hand in many cases multiple atomic changes form one larger "logical" change. Consider for example the case where the arrival of the information of being of German nationality for a person, results not only in adding this fact to the knowledge base, but also in using the right spelling for the persons name using umlauts. As shown in Example 2 this could result in three atomic changes. The information that those three changes somehow belong together should not be lost, as we would like to enable human users to observe the evolution of a knowledge base on various levels of detail. This could be achieved by constructing a hierarchy of changes on a graph.

To achieve this goal we first call Positive and Negative Atomic Changes
Changes of Level 0 and then inductively define changes of higher levels. Let \( At \) be the set of atomic changes. General changes, which are simply called changes, are defined as sequences over the set \( At \). The set \( Changes(At) \) of changes over \( At \) is the smallest set containing the empty sequence (\( \) ) and closed with respect to the following condition: if \( \{ C_1, \ldots, C_k \} \subseteq Changes(At) \cup At \), then \( (C_1, \ldots, C_k) \in Changes(At) \). An annotated change is an expression of the form \( CA \) where \( C \in Change(At) \), and \( A \) is an annotation object. We impose no restriction on the annotation object \( A \) which is attached to a change. In Section 5 we present a simple ontology schema by example, which may be used for capturing change annotations.

The changes of level at least \( n \), denoted by \( Ch(n) \), are defined inductively. Every change has a level at least 0, i.e \( Ch(0) = Changes(At) \). If \( C_1, \ldots, C_k \) are changes in \( Ch(n) \), then \( (C_1, \ldots, C_k) \in Ch(n+1) \). A change \( C \) is of level (exactly) \( n \) if \( C \in Ch(n) \setminus Ch(n+1) \), i.e. \( C \) has level at least \( n \) but not level at least \( n+1 \). The application functions \( App^+, App^- \) may be extended to a function \( App(G, C) \) whose first argument is a graph, and second argument is a change. \( App \) is recursively defined on the level of the second argument \( C \). We skip the complete and cumbersome definition and demonstrate the procedure carried out by \( App \) by some examples.

We give a rough idea of applying a change \( C \) of level > 0 to a graph. \( C \) is a sequence of changes of smaller level. These changes - being components of \( C \) - are consecutively applied to certain intermediate graphs. We demonstrate this in Example 2.
Figure 1. Schematic visualisation of a change hierarchy. Black dots represent atomic changes and gray triangles compound changes.

C1 is applied to some ontology containing information about people G and results in a new revision of G, namely G':

\[ G \xrightarrow{C1} G' \]

Since C1 consists of C2 and C3, C1 it may be resolved into:

\[ G \xrightarrow{C2} G^{(1)} \xrightarrow{C3} G' \]

And finally since C3 = (C4, C5):

\[ G \xrightarrow{C2} G^{(1)} \xrightarrow{C4} G^{(2)} \xrightarrow{C5} G' \]

C2, C4, and C5 are atomic changes and may be applied as proposed in Definitions 8 and 10.

We call a change of a level \( n > 1 \) a \textit{Compound Change}. As visualized in Figure 1 it may be seen as a tree of changes with atomic changes on its leaves.

Example 2 (Change Hierarchy) Consider the following update of the description of a person:

Resource changed \((C1)\)
- Resource classified \((C2)\)
  - http://auer.cx/Soeren hasNationality German
- Labels changed \((C3)\)
  - Label removed \((C4)\)
    - http://auer.cx/Soeren rdfs:label "Soeren Auer"
  - Label added \((C5)\)
    - http://auer.cx/Soeren rdfs:label "Sören Auer"

C1 represents a compound change with \( C1 = (C2, C3) \) and \( C3 = (C4, C5) \); C2, C4, and C5 here are atomic changes. It may be visualized as in Figure 1.
A further advantage in addition to improved change examination of compound changes is, that on their basis a knowledge transaction processing may be implemented. Assuming that a Relational Database Management System supporting transactions is used as a triple store for knowledge bases, as for example provided by the Jena [4], Sesame [7], Redland [2], and Powl [1] systems, every compound change may then be encapsulated within a database transaction. Meanwhile the repository will be blocked for other write accesses. Compound Changes thus should not be nested arbitrarily deep but up to some compound change, which was for example triggered by a user interaction. We call such a top-level compound change Upper Compound Change. Multiple, possibly semantically related compound changes can be collected in a Patch for easy distribution, for example in a Peer-to-Peer environment.

4 Change Conflict Detection

Tracking additions and deletions of statements as described in the last section enables the implementation of linear undo / redo functionality. In distributed or web based environments usually several people such as knowledge engineers and domain experts contribute changes to a knowledge base. In such a setting it is highly demandable to rollback only distinct earlier changes. Of course, this will not be possible for arbitrary changes.

Consider the case when some statements were added to a graph in the change \( C_1 \) and removed later in the change \( C_2 \). The rollback of the change \( C_1 \) should not be possible any longer after \( C_2 \) took place. In the opposite case when statements are removed from the knowledge base first and added again later, the rollback of the deletion should not be possible either. The following definitions clarify which atomic changes are compatible with a distinct knowledge base in this sense.

**Definition 11 (Compatibility of a Positive Atomic Change with a Graph)**

A Positive Atomic Change \( C_G \) is compatible with a graph \( G' \), iff \( C_G \) is not equivalent to some subgraph of \( G' \).

**Definition 12 (Compatibility of a Negative Atomic Change with a Graph)**

A Negative Atomic Change \( C_G \) is compatible with a graph \( G' \), iff \( C_G \) is equivalent to some subgraph of \( G' \).

If a positive (negative) atomic change \( C_G \) is compatible with some graph \( G' \) then it may be easily applied to \( G' \) by simply adding (respectively removing) the statements of \( C_G \) to \( G' \). Possibly blank node identifiers have to be renamed in \( C_G \) if the same occurs in \( G' \).

The notion of compatibility may be easily generalized to compound changes. Since the changes belonging to a compound change are ordered, every compound change may be broken up into a corresponding sequence of atomic changes \( (C_1, \ldots, C_n) \).

**Example 3 (Sequence of Atomic Changes)** Considering the compound change from Example 2 the corresponding sequence of atomic changes will be \( (C_2, C_4, C_5) \).
Definition 13 (Compatibility of a Compound Change with a Graph)

A compound change \( C_G' \) is compatible with a graph \( G \), iff

- the first atomic change in the corresponding sequence of atomic changes \( (C_1, ..., C_n) \) is compatible with \( G \) and results in \( G^1 \)
- every following atomic change \( C_i \) (\( 1 < i \leq n \)) from the sequence is compatible with the intermediate graph \( G^{i-1} \) and its application results in \( G^i \).

The compatibility is especially interesting if \( G' \) is some prior version of \( G \), since it supports the decision if the change may be rolled back or reapplied.

Our compatibility concept only deals with possible conflicts on the level of statements. In the remaining part of this section we would like to point out directions as to how coping with incompatibilities on higher conceptual levels could be enhanced.

When reviewing changes on a graph, we further want to distinguish changes operating on resources that do no longer belong to the knowledge base. Changes which result in graphs where no corresponding statement with \texttt{rdf:type} as predicate exists for one of the URI references occurring as a subject, could be marked to point out that that resource does no longer belong to the graph and that special attention is needed.

In [10] the impact of distinct change patterns on instance data is studied. Change patterns include all elementary operations on a knowledge base such as adding, deleting of classes, properties or instances. The effects on instances are categorized into change patterns which result in information preserving, translatable or information-loss changes. If a compound change contains an atomic change matching a change pattern of one of the latter two categories, this may be indicated to the user and possible solutions could be offered (see also Section 6 for details on ontology evolution patterns). If the graph represents some Web Ontology Language (OWL) knowledge base, furthermore a description logic reasoner may be used to check whether a model is consistent after a change is applied or not. Ideally an evolution enabled knowledge base editor provides an interface to dynamically plug-in functionality to check the applicability of a distinct change with respect to a certain graph.

5 Representation of Changes

To distribute changes on a graph (e.g. in a client server or peer-to-peer setting), a consistent representation of changes is needed. We propose to represent changes as instances of a class \texttt{log:Change}. Statements to be added or deleted by atomic changes are represented as reified statements and referenced by the properties \texttt{log:added} and \texttt{log:removed} from a change instance. The property \texttt{log:parentChange} relates a change instance to a compound change instance of higher level.

To achieve our goal of enhanced human change review, it should be possible to annotate changes with significant information, such as about the user making the change, the date and time on which the change took place, a human-readable
documentation about why the change was made, and which effects it may have, just to mention a few. Table 1 summarizes important properties attached to log:Change.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>A string or URI identifying predefined action classes</td>
<td>‘Resource changed’</td>
</tr>
<tr>
<td>User</td>
<td>A string or URI identifying the editing user</td>
<td><a href="http://auer.cx/Soeren">http://auer.cx/Soeren</a></td>
</tr>
<tr>
<td>DateTime</td>
<td>The timestamp in xsd:DateTime format when the change took place</td>
<td>“20050320T16:32:11”</td>
</tr>
<tr>
<td>Documentation</td>
<td>A string containing a human readable description of the change</td>
<td>Nationality added and name typing corrected correspondingly</td>
</tr>
<tr>
<td>ParentChange</td>
<td>Optional URI identifying a compound change this change belongs to</td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Properties for representing and annotating changes

The complete OWL ontology schema for capturing the change information is provided at http:/powl.sf.net/logOnt; Example 4 illustrates its usage.

On the other side changes have to be represented internally by the ontology editor or storage system. This may be done by attaching a change tracking ontology to every ontology stored. Change tracking for those attached change tracking ontologies, of course, has to be disabled to avoid endless loop of change tracking. Scalable ontology storage solutions usually use implementations on top of a Relational Database Management Systems (RDBMS). Here may be advantageous with respect to performance and storage space to save change tracking information separately from the actual ontologies.

6 Evolution Patterns

The versioning and change tracking strategy presented so far is applicable to arbitrary RDF graphs but also enables the representation and annotation of changes on higher conceptual levels than the one of pure statements. In this section we demonstrate how it may be used and extended to support OWL ontology evolution in distributed environments.

OWL ontologies consist of classes arranged in a class hierarchy, properties attached to those classes, and instances of the classes filled with values for the properties. Now we classify atomic changes operating on OWL ontologies according to specific patterns reflecting common evolutionary changes. The positive atomic change (hasAddress,rdf:type,owl:ObjectProperty) for example can be classified to be an object property addition, since the predicate of the statement in
Example 4 (RDF representation of Compound Changes)  This example represents the compound change from example 2.

```RDF
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
@prefix log: <http://powl.sf.net/logOnt>

C1 rdf:type log:Change
C1 log:Action log:ResourceChanged
C1 log:User http://auer.cx/Soeren
C1 log:DateTime "20050320T16:32:11"
C1 log:Documentation "Nationality added and name typing corrected correspondingly"

C2 rdf:type log:Change
C2 log:Action log:Classified
C2 log:DateTime "20050320T16:32:11"
C2 log:Documentation "Soeren Auer is of German Nationality!"
C2 log:ParentAction C1
C2 log:added S1

C3 rdf:type log:Change
C3 log:Action log:LabelChanged
C3 log:DateTime "20050320T16:32:11"
C3 log:Documentation "Label with"
C3 log:ParentAction C1

C4 rdf:type log:Change
C4 log:Action log:StatementsRemoved
C4 log:DateTime "20050320T16:32:11"
C4 log:ParentAction C3
C4 log:removed S2

C5 rdf:type log:Change
C5 log:Action log:StatementsAdded
C5 log:DateTime "20050320T16:32:11"
C5 log:ParentAction C3
C5 log:added S3

S1 rdf:type rdf:Statement
S1 rdf:subject http://auer.cx/Soeren
S1 rdf:predicate hasNationality
S1 rdf:object German

S2 rdf:type rdf:Statement
S2 rdf:subject http://auer.cx/Soeren
S2 rdf:predicate rdfs:label
S2 rdf:object "Soeren Auer"

S3 rdf:type rdf:Statement
S3 rdf:subject http://auer.cx/Soeren
S3 rdf:predicate rdfs:label
S3 rdf:object "Sören Auer"
```

S1, S2, and S3 here represent the statements to be added or removed in reified form.
the change is \texttt{rdf:type} and the object is \texttt{owl:ObjectProperty}). Complementary there is a category of \textit{object property deletions} for negative atomic changes with that predicate and object.

More complicated atomic changes consisting of several statements may be detected by SPARQL queries [12]. Consider for instance the following atomic change of adding a cardinality restriction for the property \texttt{nationality} attached to the class \texttt{Person}:

\begin{verbatim}
Person owl:subClassOf :_1
:_1 rdf:type owl:Restriction
:_1 owl:onProperty nationality
:_1 owl:maxCardinality 2
\end{verbatim}

It will be recognized by the following SPARQL query:

\begin{verbatim}
ASK {
  ?x owl:subClassOf ?y .
  ?y rdf:type owl:Restriction ;
  owl:onProperty nationality ;
  owl:maxCardinality ?z
}
\end{verbatim}

We define modification patterns to be compound changes consisting of a negative atomic change followed by a positive atomic change operating on the same resource (i.e. having the same subject) and belonging to complementary evolution patterns. We call patterns for atomic changes and modification patterns basic evolution patterns. A WonderWeb deliverable [8] gives a taxonomy of basic evolution patterns for classes and their possible effects on instance data.

Thoughtless applying changes to OWL ontologies may easily lead to inconsistencies or unwanted effects. Hence we propose to carefully evaluate possible effects on instance data caused by changes belonging to basic evolution patterns. Data migration algorithms may be developed for changes belonging to the basic evolution patterns. Complex evolution patterns then establish a relation between specific categories of compound changes and corresponding data migration algorithms. This is illustrated at the examples of complex evolution patterns for class deletions and re-classifications in the next two subsections.

\textbf{Class deletions} The deletion of some entity from an ontology corresponds to the deletion of all statements from the model where an URI referencing the entity occurs as subject, predicate, or object. The deletion of a distinct class thus will result in the following serious effects:

- former instances of the class are less specifically typed,
- former direct subclasses become independent top level classes,
- properties having the class as domain become universally applicable,
- properties having the class as range will lose this restriction,
In most cases some or all of these effects are not desired to be that rigoros, but have to be mitigated. Before actually deleting the class, we then have to cope with the following aspects of the classes usage.

- *What happens with instances of the class?* If instances of a class $C$ should be preserved they may be reclassified to be instances of a superclass of $C$ (labeled $I_R$). If $C$ has no explicit direct superclass the instances may be classified to be instances of the implicit superclass `owl:Thing`. Otherwise all its instances may be deleted ($I_D$).
- *How to deal with subclasses?* Subclasses may be deleted ($S_D$), reassigned in the class hierarchy ($S_R$) or kept as independent top level classes ($S_K$).
- *How to adjust properties having the class as domain (or range)?* The domain (or range) of properties having the class as domain (or range) may be extended (i.e. changed to a superclass - $P_E$) or restricted (i.e. changed to a subclass - $P_R$). A further possibility is to delete those properties ($P_D$).

Some combinations of those evolution strategies obviously do not make sense (i.e. $(I_D, S_D, P_D)$ - deleting all instances and subclasses and restricting the domain and range of directly attached properties) while others are heavily needed:

- $(I_D, S_D, P_D)$ - delete complete subtree including instances and directly attached properties
- $(I_D, S_R, P_E)$ - cut class off
- $(I_R, S_R, P_E)$ - merge class with superclass

As those different class deletions illustrate, different intentions to delete a class result in different combinations of data migration strategies and finally in different complex evolution patterns. Some other example for a complex ontology evolution pattern is the reclassification of a complete sub-class tree.

**Reclassification** Often the distinction between abstract categories and concrete entities is not easy, resulting in different modeling possibilities, when it is required to stay within OWL DL: representation as classes or instances. In a later modeling or usage stage the selected representation strategy (classes or instances) may turn out to be suboptimal and reclassification is required.

If all classes in a whole class tree below a class $C$ have no instances and directly attached properties, then they may be converted into instances. This can be done by defining a functional property $P$, which is used to preserve the hierarchical structure formerly encoded in the subclass-superclass relationship. Then for all classes $C_i$ in the the subtree:

- add $(C_i, \text{rdf:type}, C)$.
- if $C_i$ is a direct subclass of $C$ delete the statement $(C_i, \text{rdfs:subClassOf}, C)$, else delete all statements $(C_i, \text{rdfs:subClassOf}, C_j)$ and correspondingly add $(C_i, P, C_j)$. 
Conversely assuming we have a class $C$ and a functional property $P$ with $C$ as domain and range, which does not reference instances in cycles. Then the instances of $C$ then may be converted into subclasses of $C$ as follows:

- every statement $(I_1,P,I_2)$ is converted into $(I_1,rdfs:subClassOf,I_2)$,
- if there is for $I_1$ no triple $(I_1,P,I_2)$ add $(I_1,rdf:type,C)$.

Further complex ontology evolution patterns include:

- **Move a property** A property $P$ may be moved from a class $C_1$ to a referenced class $C_2$ (labeled log:PropertyMove).
- **"Widden" a restriction** For a property $P$ we may increase the number of allowed values or decrease the number of required values.
- **"Narrow" a restriction** For a property $P$ we may decrease the number of allowed values or increase the number of required values.
- **Split a class** A class $C$ may be split into two new classes $C_1$ and $C_2$ related to each other by some property $P$ (labeled log:ClassSplit).
- **Join two classes** Two classes $C_1$ and $C_2$ referencing each using a functional property may be joined.

These examples show that basic evolution patterns are far from being sufficient to capture the intentions for modeling changes. To support independently,
but synchronously evolving schema and instance data, as visualized at the example of splitting a class in Figure 2, we propose to annotate compound (schema) changes with their respective evolution patterns. Corresponding data migration algorithms then can be used to migrate instance data agreeing to a former version of some schema ontology.

The annotation of compound changes with ontology evolution patterns can be easily achieved within the framework showcased in Section 5. The move of a property $P_1$ from a class $C_1$ to a class $C_2$ referencing each other by a property $P_2$ could be represented for example as follows:

$$\text{C1 rdf:type log:PropertyMove}$$
$$\text{C1 log:pmProperty P1}$$
$$\text{C1 log:pmFrom C1}$$
$$\text{C1 log:pmTo C2}$$
$$\text{C1 log:pmReference P2}$$
$$\text{C1 log:removed S1}$$
$$\text{C1 log:added S2}$$

$$\text{S1 rdf:type rdf:Statement}$$
$$\text{S1 rdf:subject P1}$$
$$\text{S1 rdf:predicate rdfs:domain}$$
$$\text{S1 rdf:object C1}$$

$$\text{S2 rdf:type rdf:Statement}$$
$$\text{S2 rdf:subject P1}$$
$$\text{S2 rdf:predicate rdfs:domain}$$
$$\text{S2 rdf:object C2}$$

6.1 Data Migration Strategies

One of the main advantages of using ontologies in a distributed environment as the World Wide Web is the reuse of structural information (schemata) encoded in an ontology. If such an ontology representing structural information evolves, ontologies containing data bound to this structural information have to be adopted as well. To automate this task as much as possible it is therefore desirable to have data migration algorithms for distinct evolution patterns available. In the following two Subsections we give examples for data migration algorithms for the common evolution patterns \text{log:PropertyMove} and \text{log:ClassSplit}.

**Moving a Property** Assuming we have a change on a graph $G$ belonging to the evolution pattern \text{log:PropertyMove} moving a directly attached property $P_1$ from a class $C_1$ to some other class $C_2$ using a property $P_2$ relating $C_1$ to $C_2$. A data migration algorithm can be given as follows:
foreach triple (?i1,rdf:type,C1) in G
    find triple (i1,P1,?v) in G
    foreach triple (i1,P2,?i2) in G
        add triple (i2,P1,v) to G
    del triple (i1,P1,v) from G

It moves the $P_1$ property values of instances of $C_1$ to the related instances of $C_2$.

**Splitting a Class** Since splitting a class requires to move properties, an appropriate data migration algorithm for the log:ClassSplit evolution pattern may make use of the log:PropertyMove data migration:

1. add triple (C1,rdf:type,owl:class) to G
2. foreach triple (?i1,rdf:type,C) in G
3. create new instance identifier i
4. add triple (i,rdf:type,C1) to G
5. add triple (i1,P1,i) to G
6. foreach moved property P
7. PropertyMove(C,C1,P)

First a class $C_1$ is created (line 1), thereafter for every instance of $C$ a corresponding instance of $C_1$ is produced (lines 2-5) and then the log:PropertyMove data migration algorithm is used for every moved property.

### 6.2 Detecting Evolution Patterns

Even if the ontology development environment does not support the annotation of changes with corresponding evolution patterns in many cases they might be detected out of a sequence of atomic changes applied to a graph. Again we would like to illustrate this for the evolution patterns log:PropertyMove and log:ClassSplit.

**Detecting Property Moves** Let $G$ be a graph containing classes $C_1$ and $C_2$ and properties $P_1$ and $P_2$, where $C_1$ belongs to the domain of $P_1$ and $P_2$ establishes a relation between $C_1$ and $C_2$. A change on $M$ consisting of:

- an change removing $C_1$ from the domain of $P_1$
- an change adding $C_2$ to the domain of $P_1$

qualifies to be a evolutionary change belonging to the evolution pattern log:PropertyMove moving directly attached properties from $C_1$ to $C_2$. 
Detecting Class Splits Let $M$ be a model containing a class $C$ with a set of attached properties. A change on $M$ consisting of:

- an change adding a class $C_1$
- an change adding a property $P$ with domain $C$ and range $C_1$
- a sequence of changes qualifying to belong to the log:PropertyMove evolution pattern moving directly attached properties from $C$ to $C_1$

qualifies to be a evolutionary change belonging to the evolution pattern log:ClassSplit. Note that it is made use of the property move detection.

For comparing versions of ontologies where no change tracking information is available, a sequence of atomic changes transforming one ontology into the other may be computed. A direction for further research then is to sort and group those atomic changes in such a way that algorithms detecting evolution patterns as presented lead to successful results. Of course a complete reconstruction of the ontology evolution might not be possible, if versions differ to much.

Figure 3. Reviewing changes with Powl

7 Related Work and Summary

The versioning strategy described in this article was implemented in the web application development framework Powl\(^2\) [1], which provides a comprehensive

\(^2\) http://powl.sf.net
web user interface for collaborative knowledge base editing as well as an application programming interface for PHP developers. To every change on the knowledge base using Powl, an optional versioning comment describing the change for human consumption may be attached. The user interface of Powl’s versioning module then enables users to review changes chronologically, their compatibility with the current version is indicated and distinct changes may be rolled back. Changes may be filtered according to user, model, and date. Compound changes may be expanded up to the atomic change level indicating added (respectively removed) statements.

Other approaches targeting to support ontology evolution and versioning can be roughly divided into two categories:

– Approaches which are aware of the trace of changes which result in a new version and
– Approaches which compare ontologies and compute differences or mappings between them.

Ognyanov and Kiryakov in [11] (falling in the first category) define a formal model for tracking changes in graph-based data models. Higher-level evaluation or classification of the updates are beyond the scope of their work. Those are studied and discussed in depth, for example, in [5]. Our contribution here is a way to easily relate low-level changes on the statement level to higher-level changes on the level of complex operations. In [10] (falling in the second category) automatic techniques based on heuristic comparisons for finding similarities and differences between versions are developed. [13] develop a merging method for ontologies following a bottom-up approach which offers a structural description of the merging process. These approaches are complementary to the presented one, since they are applicable even if ontology editors or storage systems do not support a finely grained change tracking.

We presented a method for specifying complex changes by means of less complex changes and finally atomic changes on a graph. This method is especially suited to be implemented in ontology editors and storage systems. In a dynamic distributed environment sets of changes may then independently spread out from the originating ontologies. A user of some ontology may decide for every single change whether he accepts it or not. Assistance for this decision is provided by the compatibility concept between an ontology and a change. Annotation of changes on OWL ontologies with corresponding ontology evolution patterns further enables automatic data migration of independently stored instance data agreeing on the changed ontology. In this context the development of an exhaustive library of ontology evolution patterns with appropriate data migration algorithms is planned.

References


