A Versioning and Evolution Framework for RDF Knowledge Bases

Sören Auer and Heinrich Herre

University of Leipzig, 04109 Leipzig, Germany
auer@informatik.uni-leipzig.de
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 compound 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 [2]) 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 [4]. 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. In areas such as software engineering, databases, and web publishing versioning and revision control mechanisms have already been developed and successfully applied. 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) standard was released as an extension to the Hyper Text Transfer Protocol (HTTP) supporting versioning and 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 control is based on unique serializations, enabled by their data models. Such unique

[1] http://www.w3.org/2001/sw/Activity

I. Virbitskaite and A. Voronkov (Eds.): PSI 2006, LNCS 4378, pp. 55–69, 2007.
© Springer-Verlag Berlin Heidelberg 2007

serializations are not available for Semantic Web knowledge bases, 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 contrast to just statements of the RDF data model.

In this paper, we present an approach for the versioning of distributed knowledge bases grounded on the RDF data model with support for 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 adhering to a certain ontology version.

This paper is structured as follows: Our 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 knowledge base (Section 2). 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, thus enabling 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 enable the evolution of ontologies with regard to higher conceptual levels than the one of statements we introduce evolution patterns (Section 6) and give examples for appropriate data migration algorithms (Section 7). We further give account of the successful implementation of the approach in Powl, summarize related work and give an outlook on planned directions for future work (Section 8).

2 Atomic Changes on RDF Graphs

To introduce our notion of atomic changes on RDF graphs we need recall some preliminary definitions from [5]. 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 Node*s, 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. $M(lit) = lit$ *for all literals* lit *which are nodes of* G*.*
3. $M(uri) = uri$ *for all URI references* 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'*.*

Based on these definitions we want to discuss the possible changes on a graph. RDF statements are in [7] identified to be the smallest manageable piece of knowledge. This view is justified by the fact that there is no way to add, remove, or update a resource or literal without changing at least one statement, whereas the opposite does not hold. We adopt this 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.

Definition 7 (Positive Atomic Change). *An 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.*

The rationale behind this definition is the aim of applying the positive atomic change C_G to the graph G. Hence, a positive atomic change on a graph G can be applied to G to yield a new graph 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 $G \stackrel{C_G}{\rightarrow} G'$. We say that C_G is applied to the graph G with result G'.*

Application of the positive atomic change C_G to G yielding G' is just identifying the union of C_G and G with G'. Of course a graph may not only be changed by adding statements leading to the notion of a negative atomic change.

Definition 9 (Negative Atomic Change). *A subgraph C_G of G is said to be a* Negative Atomic Change *on 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 $Apl^-(G, C_G) = G \backslash C_G = G'$ which is symbolized by $G \stackrel{C_G}{\rightarrow} G'$. We say that C_G 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.

3 Change Hierarchies

The evolution of a knowledge base typically results in a multitude of sequentially applied atomic changes. These 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 1 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 hierarchies of changes on a graph.

To achieve this goal first of all Atomic Changes are called *Changes of Level 0* and then changes of higher levels are defined inductively. 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

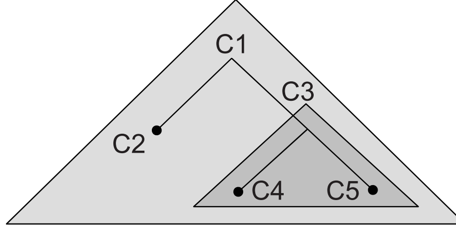

Fig. 1. Schematic visualisation of a change hierarchy. Black dots represent atomic changes and gray triangles compound changes.

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 C^A where $C \in Change(At)$, and A is an annotation object. No restriction is imposed on the annotation object A which is attached to a change. In Section 5 we present a simple ontology schema, which may be used for capturing such 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 $C \in Ch(n) \backslash Ch(n+1)$, 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. Now we would like to apply a change C of level > 0 to a graph. Since C is a sequence of changes of smaller level, these changes – being components of C – can be consecutively applied to intermediate graphs. This is demonstrated in the following for the change from Example 1.

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

$$G \stackrel{C1}{\rightarrow} G'$$

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

$$G \stackrel{C2}{\rightarrow} G^{(1)} \stackrel{C3}{\rightarrow} G'$$

And finally since $C3 = (C4, C5)$:

$$G \stackrel{C2}{\rightarrow} G^{(1)} \stackrel{C4}{\rightarrow} G^{(2)} \stackrel{C5}{\rightarrow} G'$$

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

Example 1 (Change Hierarchy). *Consider the following update of the description of a person:*

```
1  Resource changed                                     (C1)
2    Resource classified                                (C2)
3      http://auer.cx/Soeren  hasNationality  German
4    Labels changed                                     (C3)
5      Label removed                                    (C4)
6        http://auer.cx/Soeren  rdfs:label  "Soeren Auer"
7      Label added                                      (C5)
8        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 Fig. 1.

We call a change of a level $n > 1$ a *Compound Change*. As visualized in Fig. 1 it may be viewed as a tree of changes with atomic changes on its leafs. This enables the review of changes on various levels of detail (e.g. statement level, ontology level, domain level) and thus facilitates the human reviewing process.

A further advantage in addition to improved change examination 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, 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 certain 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 $(C_1, \ldots, C_n)$. If we consider the compound change from Example 1, the corresponding sequence of atomic changes will be $(C2, C4, C5)$.

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. However, this compatibility concept only deals with possible conflicts on the level of statements. In the remaining part of this section we point out directions how we can cope with incompatibilities on higher conceptual levels than the one of statements.

In [6] the impact of distinct change patterns on instance data is studied. Change patterns include all elementary operations on an OWL ontology 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 can be indicated to the user and possible solutions could be offered (cf. 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 Represention 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 `log:Change`. Statements to be added or deleted

by atomic changes are represented as reified statements and referenced by the properties `log:added` and `log:removed` from a change instance. The property `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 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`. The complete OWL ontology schema for capturing the change information is provided at `http:/powl.sf.net/logOnt`.

Table 1. Properties for representing and annotating changes

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

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 RDF statements. In this section we demonstrate how it may be used and extended to support consistent OWL ontology and instance data evolution.

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 changes operating on OWL ontologies according to specific patterns reflecting common change intentions. 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 the change is `rdf:type` and the object is `owl:ObjectProperty)`. Complementary there is a category of *object property deletions* for negative atomic changes

with that predicate and object. Such categories of changes can be described more formally and generally by our notion of Evolution Patterns.

Definition 14 (Evolution Pattern). *A positive (negative) evolution pattern is a triple* $(X, G(X), A(X))$*, where* X *is a set of variables,* $G(X)$ *is a graph pattern characterizing a positive (resp. negative) change with the variables* X *and* $A(X)$ *being an appropriate data migration algorithm.*

Graph patterns are essentially graphs where certain URI references are replaced by placeholders (i.e. variables). The precise definition is omitted here but can be found in [8]. As an example we consider the following positive atomic change of adding a cardinality restriction to the property `nationality` attached to the class `Person`:

```
1   Person  owl:subClassOf      :_1
2   :_1     rdf:type            owl:Restriction
3   :_1     owl:onProperty      nationality
4   :_1     owl:maxCardinality  2
```

The corresponding evolution pattern will be $AddMaxCardinality = (X, G(X), A(X))$ with $X = (class, property, maxCardinality)$, the graph pattern $G(X)$ will be:

```
1   ?class         owl:subClassOf      ?restriction
2   ?restriction   rdf:type            owl:Restriction
3   ?restriction   owl:onProperty      ?property
4   ?restriction   owl:maxCardinality  ?maxCardinality
```

Finally, the data migration algorithm $A(class, property, maxCardinality)$ will iterate through all instances of *class* and remove property values of *property* exceeding $maxCardinality$.

Beside facilitating the review of changes on a knowledge base the classification of changes into such evolution patterns enables the automatic migration of instance data, even in settings where instance data is distributed. General evolution patterns can be constructed out of sequences of positive and negative evolution patterns. The modification of a `owl:maxCardinality` restriction can thus be made up by sequentially applying changes belonging to the negative evolution pattern $DelMaxCardinality$ and the positive evolution pattern $AddMaxCardinality$.

In [4] a taxonomy of change patterns for OWL ontologies and their possible effects on instance data is given. However, from our point of view these change patterns will not be sufficient to capture change intentions and to enable automatic instance data migration. Intentions of changes can be made explicit by precisely describing effects on instance data, e.g. by providing instance data migration algorithms. We illustrate possible intentions for class deletions and re-classifications in the next two subsections.

Class Deletions. The deletion of some entity from an ontology corresponds to the deletion of all statements from the graph 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 rigorous, 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_R) - deleting all instances and subclasses and restricting the domain and range of directly attached properties) while others are heavily needed (see also Fig. 2):

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

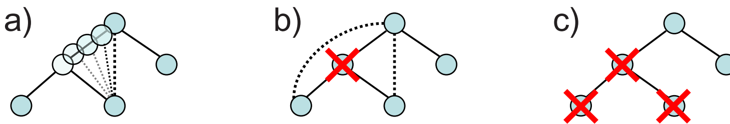

Fig. 2. Different intentions for deleting a class: a) merge with superclass, b) cut class off, c) delete subtree

As those different class deletions illustrate, different intentions to delete a class result in different combinations of data migration strategies and finally in different 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 `(Ci,rdf:type,C)`,
- if C_i is a direct subclass of C, then delete the statement `(Ci,rdfs:subClassOf,C)`, else delete all statements `(Ci,rdfs:subClassOf,Cj)` and correspondingly add `(Ci,P,Cj)`.

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).

Beside class deletions and reclassification there are other ontology evolution patterns such as:

- *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 the basic change patterns from [4] are not sufficient to capture the intentions for ontology changes. To support independently, but synchronously evolving schema and instance data, as visualized at the example of splitting a class in Fig. 3, 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 the ontology. However, it is future work to provide a complete library of ontology evolution patterns.

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 $P1$ from a class $C1$ to a class $C2$ referencing each other by a property $P2$ could be represented for example as follows:

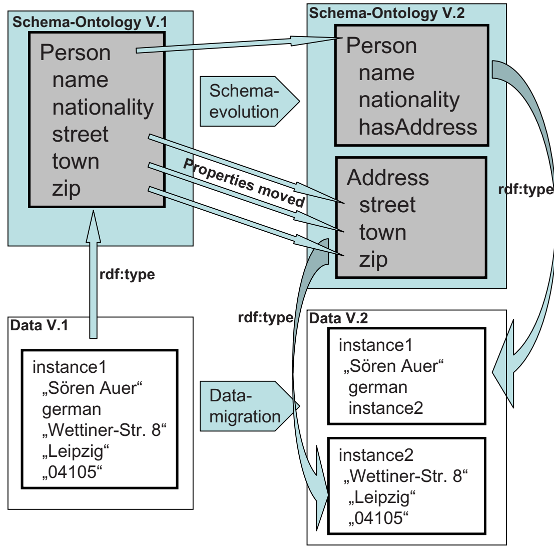

Fig. 3. Ontology evolution and instance data migration at the example of splitting a class

```
1   @prefix  rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2   @prefix  rdfs: <http://www.w3.org/2000/01/rdf-schema#
3   @prefix  log:  <http:/powl.sf.net/logOnt>
4
5   C1  rdf:type         log:PropertyMove
6   C1  log:pmProperty   P1
7   C1  log:pmFrom       C1
8   C1  log:pmTo         C2
9   C1  log:pmReference  P2
10  C1  log:removed      S1
11  C1  log:added        S2
12
13  S1  rdf:type         rdf:Statement
14  S1  rdf:subject      P1
15  S1  rdf:predicate    rdfs:domain
16  S1  rdf:object       C1
17
18  S2  rdf:type         rdf:Statement
19  S2  rdf:subject      P1
20  S2  rdf:predicate    rdfs:domain
21  S2  rdf:object       C2
```

7 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 instance data migration algorithms for evolution patterns available. In the following two subsections we give examples for data migration algorithms for the common evolution patterns `log:PropertyMove` and `log:ClassSplit`.

Moving a Property. Assuming we have a change on a graph G belonging to the evolution pattern `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:

```
1   foreach triple (?i1,rdf:type,C1) in G
2     find triple (i1,P1,?v) in G
3     foreach triple (i1,P2,?i2) in G
4       add triple (i2,P1,v) to G
5       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,R,i) to G
6   foreach moved property P
7     PropertyMove(C,C1,P)
```

First a class $C1$ is created (line 1), thereafter for every instance of C a corresponding instance of C_1 is created, whereas the relation between both is established by the property R (lines 3-5) and finally the `log:PropertyMove` data migration algorithm is used for every moved property (lines 6,7).

8 Related Work and Summary

The versioning strategy described in this paper was implemented in the web application development framework Powl [1], which provides a comprehensive web user interface for collaborative knowledge base authoring as well as an application programming interface for PHP developers. To every change on the knowledge base using Powl, an optional versioning comment can be attached describing the change for review by humans. 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, ontology, and date. Compound changes may be expanded up to the atomic change level indicating added (respectively removed) statements.

pOWL - *Semantic Web Platform*: http://bis.informatik.uni-leipzig.de/viewBIS_OW

Models | Triples | Classes | Properties | Instances | RDQL | Search | Version

RDF Versioning

Filter: model: http://bis.informatik.uni-leipzig.de/viewBIS_OWL.owl |

Search returned 7 results.[c]

S	Nr.	Date	User	Action	Rollback
☐	6764	2005/03/17 13:00:47	127.0.0.1	⊞ **Property values changed:** *rdfs:comment*	[R]
☐	6763	2005/03/17 12:59:35	127.0.0.1	⊞ **Statement added:**	[R]
☐	6762	2005/03/17 12:58:04	127.0.0.1	⊞ **Property values changed:** *swrc:fax* + http://www.informatik.uni-leipzig.de/~auer swrc:fax +49 (341) 97-32322	[R]
☐	6761	2005/03/15 11:34:56	127.0.0.1	⊞ **Property values changed:** *worksInRoom*	[R]
☐	6760	2005/03/15 11:34:43	127.0.0.1	⊞ **Property values changed:** *worksInRoom*	
☐	6759	2005/03/15 11:34:18	127.0.0.1	⊞ **Property values changed:** *worksInRoom*	[R]
☐	6758	2005/03/15 11:34:02	127.0.0.1	⊞ **Property values changed:** *worksInRoom*	

▹ **Rollback selected actions**

Fig. 4. Reviewing changes with Powl

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
– Approches which compare ontologies and compute differences or mappings between them.

Ognyanov and Kiryakov in [7] (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 [3]. 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 [6] (falling in the second category) automatic techniques based on heuristic comparisons for finding similarities and differences between versions are developed. [10] 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. Ljiljana Stojanjovic's work [9] on ontology evolution gives an overview over current developments.

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

1. Sören Auer. Powl: A web based platform for collaborative semantic web development. In Sören Auer, Chris Bizer, and Libby Miller, editors, *Proceedings of the Workshop Scripting for the Semantic Web*, number 135 in CEUR Workshop Proceedings, Heraklion, Greece, 05 2005.
2. Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic Web. *Scientific American*, 284(5):34–43, May 2001.
3. Ying Ding, Dieter Fensel, Michel Klein, and Borys Omelayenko. Ontology management: survey, requirements and directions. Technical report, IST 1999-10132 Ontoknowledge Project, Deliverable 4, 2001.
4. Michel Klein, Dieter Fensel, Atanas Kiryakov, Natasha F. Noy, and Heiner Stuckenschmidt. Wonderweb deliverable D20. versioning of distributed ontologies, December 18 2002.
5. Graham Klyne and Jeremy J. Carroll. Resource Description Framework (RDF): Concepts and abstract syntax. W3C Recommendation (http://www.w3.org/TR/rdf-concepts), 2 2004.
6. Natalya Fridman Noy, Sandhya Kunnatur, Michel C. A. Klein, and Mark A. Musen. Tracking changes during ontology evolution. In Sheila A. McIlraith, Dimitris Plexousakis, and Frank van Harmelen, editors, *Proceedings of the Third International Semantic Web Conference (ISWC 2004), Hiroshima, Japan, November 7-11, 2004*, volume 3298 of *Lecture Notes in Computer Science*, pages 259–273. Springer, 2004.
7. Damyan Ognyanov and Atanas Kiryakov. Tracking changes in RDF(S) repositories. In Asunción Gómez-Pérez and V. Richard Benjamins, editors, *EKAW*, volume 2473 of *Lecture Notes in Computer Science*, pages 373–378. Springer, 2002.
8. Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF. W3C Working Draft (http://www.w3.org/TR/rdf-sparql-query/), 2005.
9. Ljiljana Stojanovic. *Methods and Tools for Ontology Evolution*. PhD thesis, Institut für Angewandte Informatik und Formale Beschreibungsverfahren, Universität Karlsruhe (TH), 2004.
10. Gerd Stumme and Alexander Maedche. FCA-Merge: A bottom-up approach for merging ontologies. In *JCAI '01 - Proceedings of the 17th International Joint Conference on Artificial Intelligence, Seattle, USA, August, 1-6, 2001, San Francisco/CA: Morgen Kaufmann 2001/07/04*, 2001.