Abstract. Knowledge domains and their semantic representations via ontologies are typically subject to change in practical applications. Additionally, engineering of ontologies often takes place in distributed settings where multiple independent users interact. Therefore, change management for ontologies becomes a crucial aspect for any kind of ontology management environment. This paper introduces a new RDF-centric versioning approach and an implementation called SemVersion. SemVersion provides structural and semantic versioning for RDF models and RDF-based ontology languages like RDFS. The requirements for our system are derived from a practical scenario in the librarian domain, i.e. the MarcOnt scenario.

1 Introduction

For many practical applications, ontologies (cf. [?]) can not be seen as static entities, they rather change over time. Support for change management is crucial to support uncontrolled, decentralized and distributed engineering of ontologies. First approaches have been described in [?],[?]. But, there is no one yet that functions as a standard versioning system for ontologies like CVS does in the field of software development.

This paper introduces a new RDF-based versioning approach and describes the new versioning system SemVersion that provides versioning for RDF models and RDF-based ontology languages like RDFS, OWL flavors or TRIPLE[?]. We present a working methodology accompanied by its implementation in the system SemVersion. The methodology and the system provide a well-defined core functionality for ontology versioning. We captured the requirements for the methodology and our system in a real-life scenario from the librarian domain.

Our approach is inspired by the classical CVS system for version management of textual documents (e.g. Java code). Core element of our approach is the separation of language-specific features (the diff) from general features (such as structural diff, branch and merge, management of projects and metadata).
A speciality of RDF is the usage of so-called blank nodes. As part of our approach we present a method for blank node enrichment which is required for the versioning of such blank nodes.

The paper is structured as follows. In Section 2 we present a practical scenario, i.e. the MarcOnt scenario. From this scenario, which is about providing a collaboration-based ontology for digital libraries, we derive requirements for versioning of ontologies and in particular for our system SemVersion.

We then describe our framework for RDF-based versioning of ontologies. Firstly, we motivate a layered approach which follows the layers of the Semantic Web architecture in Section 3. Secondly, in Section 4, we show our approach for versioning of RDF data. Thirdly, in Section 5, we argue that our approach can be reused for the versioning of RDF-based ontologies. In Section 6 we present the architecture of our system SemVersion and discuss further implementation issues. Before we conclude, we depict related work in Section 7.

2 Scenario and Requirements

With the use of computers librarian systems became popular. They are used to catalogue and search books and other resources stored in libraries all over the World. Today, the Internet is a place where more and more documents of all kinds are published. They are often organized within Digital Libraries. Searching them often produces poor results and is too slow, because traditional systems are ineffective when adapted to suit the needs of a Digital Library.

The MarcOnt Initiative\(^4\) is an attempt to address some of these problems by creating an extendable and versatile description system for librarian purposes which will take advantage of the Semantic Web. One result of the MarcOnt Initiative will be a standard ontology for digital libraries that incorporates existing standards like MARC21\(^5\) and Dublin Core\(^6\). The ontology will be easy to merge with other bibliographic ontologies or to translate to them. It will also support searching\(^7\) and browsing (cf. \([?]\)) using ontology concepts and semantic user profiles e. g. to deduce personal preferences.

To achieve these goals, there is a need to include a wide community of librarians, computer analysts, software developers, programmers etc. in the development process. Only a community effort can create a recognized standard, as standards are at the end based on social agreement. Such a community needs the appropriate collaborative ontology engineering tools.

The core feature of the MarcOnt Initiative collaboration portal is an integrated ontology builder. Concurrent versions of the MarcOnt ontology are built out of suggestions proposed by community members (see Fig. 1). These suggestions themselves can evolve over time. After a voting process, some are applied and a new ontology version is created.

\(^4\) http://www.marcont.org/
\(^5\) http://www.loc.gov/marc/
\(^6\) http://dublincore.org/
\(^7\) e.g. http://www.jeromedl.org/
As soon as organizations start to use the current ontology version, the problem arises, how to develop the ontology further while retaining interoperability between systems based on different versions of the MarcOnt ontology. There are two possible solutions: Either by allowing only monotonic extensions or by developing accompanying mapping/translation rules between versions. To be able to do so, the ontology builder must be capable of presenting differences between various versions.

The ontology builder of the MarcOnt portal requires not only a GUI for building the ontology through submitting changes. It also needs the ability to: manage a main trunk of the ontology (M1); manage versions of suggestions (M2); generate snapshots of the main ontology with some suggestions applied (M3); detect and resolve conflicts (M4); add suggestions to the main trunk (M5) and attach mapping/translation rules (M6).

We generalized these requirements in order to: (i) support the CVS core functions, (ii) create a system which can be easily integrated into existing ontology engineering environments such as the ontology builder. Thus our requirements are:

R1 Basic management functions for projects
R2 Retrieve and commit versions, either as full ontologies or as diffs
R3 Branch and merge operations
R4 Flexible annotation of versioning artefacts with arbitrary metadata such as “top voted”, “released” or “temporal view”
3 A layered approach for versioning various ontology languages

On one hand our goal is to support as many ontology languages as possible with our versioning system. On the other hand we want to provide this support as specific as possible. That means SemVersion should provide general versioning features independent of the ontology language used. At the same time, it should be easy to integrate functions that are specific for a particular ontology language, like calculating semantic diffs.

To achieve this, we chose an RDF-based, layered approach. In this section we describe how our approach follows the Semantic Web architecture, why RDF is particularly suitable as a basis layer, and explain the roles of the RDF versioning layer and the ontology versioning layer in our approach.

3.1 Following the layered architecture of Semantic Web languages

For the Semantic Web, a layered architecture has been suggested [?]. These layers define different aspects of the Semantic Web like syntax (Unicode, XML, URI), data model (RDF, RDF-Schema) and semantics (ontology vocabulary, logic, proof).

In the Semantic Web architecture RDF plays a key role as a universal data model that is generic enough to encode all the ontology languages needed for representing richer semantics. [?] and [?] demonstrate how new ontology languages can be created on top of RDF and RDF-Schema and how this was done to create the DAML+OIL ontology language which evolved into the Web Ontology Language (OWL) [?]. OWL comes in three different flavors (Lite, DL, and Full) and even more OWL flavors are suggested such as DLP [?]. But there are also Horn logic based languages that are encoded in RDF like TRIPLE, a rule-based language proposed in [?]. Since there are more and more RDF-based languages, a versioning system that is based on RDF, seems to be a good basis for reaching our goal to support as many languages as possible as specific as possible. The next subsection describes, why RDF is particularly suitable as data model for ontology languages.
3.2 RDF as structural core of ontology languages

The most elementary thing that is needed to model a shared conceptualization of some domain is a way to denote entities and to unambiguously reference them (Figure 2). For this purpose RDF uses URIs, identifiers for resources, that are supposed to be globally unique. Compared to logical modelling this roughly corresponds to having constants, predicate and function symbols in a logical theory. Every ontology language needs to provide means to denote entities. For global systems the identifier should be globally unique. Having entities, that can be referenced, the next step is to describe relations between them. As relations are semantic core elements, they should also be unambiguously addressable. Properties in RDF can be seen as binary relations. This is the very basic type of relations between two entities. More complex types of relations can be modelled by defining a special vocabulary for this purpose on top of RDF, like it has been done in OWL. Figure 2 illustrates that denoting and stating relations between entities are the second core element in semantic models.

The two core elements for semantic modelling, mechanisms to identify entities and to identify and state relationships between them, are provided by RDF. Ontology languages that build upon RDF use these mechanisms and define the semantics of certain relationships, entities, and combinations of relationships and entities. So RDF provides the structure in which the semantic primitives of the ontology languages are embedded. That means we can distinguish three layers here: syntactic layer (e.g. XML), structural layer (RDF), semantic layer (ontology languages) (see Fig. 3).

The various ontology languages differ in their vocabulary, their logical foundations, and epistemological elements, but they have in common that they describe structures of entities and their relations. Therefore RDF is the largest common denominator of all ontology languages. RDF is not only a way to encode the ontology languages or just an arbitrary data model, but it is a structured data model that matches exactly the structure of ontology languages.

3.3 RDF-layered ontology versioning methodology

Derived from the considerations above we propose a layered RDF-based ontology versioning methodology.

As the base layer, we suggest an RDF versioning layer that manages versions of RDF models and their metadata. The metadata itself consist again of one or more RDF models. This leads to a very generic versioning system, where client system can store their version-related metadata. By systematically storing and allowing access to versions of RDF models and their metadata the RDF versioning system provides the complete data management that every versioning
The system can also calculate structural diffs between two RDF models. This is the set-based difference between the two triple sets of the models. The RDF layer is thus responsible for

- versioning data management,
- structural diff, and
- storage layer access.

The RDF versioning layer is a full, usable RDF versioning system on its own. It can also be used to version RDF-encoded ontology languages as well, by creating an ontology versioning layer on top of it. This ontology version layer should provide the following additional language specific versioning functions:

- Semantic diffs that take the semantics of the specific language into account.
- Merging with semantic conflict detection as language specific conflicts can not be detected at the RDF level.

4 Versioning on the RDF layer

Commit and branch are pure data management operations. Diff, merge and conflict detection depend on the used data model. As merge and conflict detection depend on diff calculation, the diff operation is really at the heart of a versioning system. Versioning on the RDF data layer consists of the core functions download, diff, commit, branch and merge, which will be described first. We then explain additional management functions necessary for a full versioning system.

The most well-known versioning system in the developer community is probably CVS [?]. We explain the SemVersion terminology by referencing the CVS terms.

download via HTTP GET in SemVersion is equivalent to a CVS checkout. It is the same as update as SemVersion has no locking mechanism.

commit can either be done by committing RDF models or by committing RDF diffs. If the RDF models in question are too large to be send over the network, committing diffs is required. In this case one needs a way to address blank nodes unambiguously, to be able to update them correctly as well. We present our blank node enrichment approach. If a user commits a new version of an ontology, it is considered an update to an existing version.

diff in SemVersion returns a triple-set-based difference between two models.

This is a structural diff in contrast to a semantic diff, which takes also semantically inferred triples into account.

branch and merge work the same as in CVS.

In an ontology life cycle, one can distinguish the engineering phase and the usage phase. The difference between the two phases is: Once an ontology is used, that specific version should not change without notice. In our model, we don’t distinguish between these two phases. Instead every ontology version is accessible unchanged via HTTP for an unlimited period of time. To enable automatic
updatable versions we introduce „magic URLs” that refer to the most recent version of a given ontology in a given branch.

In an ontology versioning context retrieval will be the most often accessed operation, as e. g. RDFS or OWL ontologies are referred to by their URLs. This implies that released versions of an ontology have to be downloadable via HTTP.

4.1 RDF specific function

RDF Diff The most important function of versioning systems is the computation of a „diff", the difference between two arbitrary versions. It allows users to see and discuss about changes between versions and decide about their acceptance. In the next paragraphs we will look at ways to calculate and represent structural diffs.

The diff function $d(A,B) \rightarrow \langle a(A,B), r(A,B) \rangle$ is a non commutative function from two triple sets $(A, B)$ to two triple sets of added $(a(A,B))$ and removed $(r(A,B))$ statements, with $a(A,B) = B - A = B \setminus (A \cap B)$ and $r(A,B) = A - B = A \setminus (A \cap B)$.

Such diffs can be computed by simple set arithmetics for triple sets that contain only URIs and literals, as shown in [?]. Unfortunately, blank nodes make the diff more complicated. If a user commits a new model and later requests a diff, the system cannot tell whether two blank nodes are equal or different. They have by definition no globally unique identifier. Literals are no problem, as they may only occur in the object of a statement. As shown in the example in Fig. 4, there are always two possible diffs. The conservative diff assumes that blank nodes between two models are never equal, following the RDF semantics for blank nodes. This can be semantically wrong, if a blank node in one model represents the same resource as a blank node in another model. This is especially true for two versions of the same model. Thus a better diff is needed, but only possible, if blank nodes can be identified unambiguously.

We propose to overcome this problem by uniquely identifying blank nodes. This can be done by using a technique we call blank node enrichment. Blank

Fig. 4. Diff Example
node enrichment creates an „enriched model” from a normal model by introducing a new property bne:id. The value of this identifier property plays the role of an inverse functional property like in OWL. We chose to use a globally unique URI which can be created by a generator as described in Section 6. Blank nodes should only have one such property value assigned. This unique URI makes blank nodes globally addressable, while they remain formally blank nodes in the RDF model. All existing RDF semantics are still valid.

Most RDF processing tools will leave this information intact. In the MarcOnt scenario, a dedicated ontology builder is used, so this constraint can be enforced. In SemVersion, the content of every version is blank node enriched before it is stored in the RDF storage layer.

If a user edits an ontology locally and deletes all statements involving a particular blank node, SemVersion can deduce that it has been removed from the model. If a user locally creates new blank nodes, the tool she uses might not use blank node enrichment. SemVersion then finds no reference to existing nodes and correctly identifies the added blank nodes. If a user commits a plain model without identifier properties for some blank nodes, SemVersion falls back to a conservative diff for the non-annotated blank nodes and assumes they are new.

If a user commits a diff as an update to a previous version, this diff should also be blank node enriched, in order to work properly.

For representing RDF Diffs it might be desirable to have a single RDF model containing the whole diff information. There is no standardized way to express sets of triples within a single RDF model. The most common approach – addressing triple sets by the URL of the document containing the RDF triples – has no defined semantics. Delta[?] proposes „quoted graphs”, Named Graphs[?] proposes a new XML-based encoding called „TriX”[?]. Both proposals work outside the RDF model. We chose the following RDF-friendly approach: A triple is made addressable by

```turtle
@prefix rdfs: – as usual
@prefix ts: – the triple set ontology
@prefix u: – namespace for world-wide unique, generated URIs on this SemVersion server
@prefix bne: – bnode-enrichment ontology
@prefix : – a versioned ontology namespace
@prefix ov: – SemVersion data model

ov:added a ts:TripleSet ; #added
  ts:member u:322169000832000001.
  u:322169000832000001
    rdfs:subject <anon-32>;
    rdfs:predicate :hasAuthor;
    rdfs:object "James Hender".
  u:322169000832000001
    rdfs:subject <anon-32>;
    rdfs:predicate :hasAuthor;
    rdfs:object "Allison Druin".

ov:removed a ts:TripleSet ; #removed
  ts:member u:322169000832000002;
  u:322169000832000002
    rdfs:subject <anon-32>;
    rdfs:predicate :hasAuthor;
    rdfs:object "Jim Hender".

# bnode enrichment
<anon-32> bne:id u:322169000832000003 .
```

Fig. 5. Example for RDF diff encoding as RDF model using blank node enrichment

8 with @prefix bne: http://SemVersion.ontoware.org/bnode-enrichment#
reification, sets of triples are represented as rdfs:Bags. This leads to a trivial triple set ontology\(^9\). A full RDF diff contains a triple set of added and a triple set of removed statements. Additionally the blank node enrichment statements have to be added, as shown in picture 5.

In the SemVersion API we additionally provide a diff represented as multiple, URI-addressable RDF models.

**Branch and Merge** Branch and merge operations allow ontology engineers to follow multiple development paths in parallel. A branch operation works like a commit, but the new version is considered to be in a new branch, marked by a different branch label.

For merge we distinguish a merge between two arbitrary versions and the merging of two branches. It is possible to merge arbitrary versions, no only those at the end of a branch. A merge of version A and version B is simply the set union of the triple sets.

Merging two branches is different. First we look at the branch point c, which is defined as the most recent common version of the two branches. Such a version always exists, as branches can only be created by committing a version to an existing version. We also take two versions from the different branches, in most cases the most recent ones, and call them a and b. Consider the example version tree given in Fig. 6. Here \( c = A, a = A'', b = B'' \). In order to merge b back into a we compute the \( diff(c, b) \) and apply it to a.

**Conflict Detection** RDF models themselves are never in a conflict state. But a diff between two models can indicate a conflict on the ontology layer. SemVersion uses a simple conflict detection heuristic, that detects if a diff adds statements about a resource that was present in c, but has been removed on its way to a. This means, the URI of a resource was used in triples from c, but no triple in a contains this URI.

### 4.2 Management Aspects Of Versioning

SemVersion uses the data model depicted in Fig. 7. We first briefly explain the definition and basic operations of each concept:

- **repository** is the root data element for an SemVersion server. Operations: add/remove/list projects, create/delete versioned model;
- **project** is a set of models. Operations: add/remove/list versioned models;
- **versioned model** represents the concept of a thing, that has different versions and branches. Operations: list versions, commit RDF model or RDF diff, commit RDF or RDF diff as branch, list version tree, get current version of branch;

\(^9\) Available at http://SemVersion.ontoware.org/2004/12/tripleset
version has predecessors, that are the versions it was created from. It has two predecessors, if it was created as the result of a merge operation. Each version also has a time stamp when it was created and a branch label. Operations: get branch label, add/get user-defined metadata, list history of versions this version was created from, get RDF content (as a set of triples), merge with version, merge with version as branch

branch is an abstract concept. Branches are labelled by URIs.

Typically a new user starts by creating a new project and then adds a RDF model to it. This model is then treated as the first version of a „versioned model“. The initial RDF model was probably created on the users desktop with third-party ontology engineering tools.

A versioned model consists of different versions that have attributes and relations. Common attributes are time stamp, branch label, status of acceptance. Predecessor relationships indicate the history path. This meta-information about versions can be managed independent of the versioned artefacts themselves. Thus this management layer can be designed very flexible and reusable. As every version can be identified via an URI, one can make arbitrary statements in RDF about them. The concepts of branches, acceptance status and version dependencies can then be represented easily in RDF. SemVersion uses this distinction of stored RDF models and statements about them. Realized as statements about versions is e. g. the concept of ontology engineering projects. Such projects are simple sets of versioned models and give the user a better ability to manage the different ontologies in progress.

Users can store arbitrary RDF encoded metadata objects for each project, versioned model and most important for each version. This data is stored in the RDF storage layer and linked by RDF statements to the versioning artefact it belongs to. Metadata models are also URI-addressable. This metadata strategy enables a good re-use of the SemVersion system, as e. g. the evolution log of an ontology engineering tool could be assigned to a version with this mechanism.

5 Using SemVersion for Ontology Versioning

In this section we explain how SemVersion can be used to build an ontology versioning system for a particular RDF-based ontology language. We can reuse the complete version data management infrastructure of SemVersion, that includes managing projects, versioned models, versions and metadata for each of these
concepts. Some basic versioning functions can also be used out-of-the box such as retrieve, commit and branch.

The following language specific versioning functions need to be implemented additionally:

- Semantic diffs - the basic structural diff provided by the RDF versioning system is not identical with the semantic diff. To calculate the semantic diff $d_l$ a system has to know the semantics of the specific language $l$. The semantic closure $s_l(M)$ of a model $M$ is the set of all statements that can be concluded from the statements in $M$ under the semantics of the RDF-based ontology language $l$. The semantic diff of two models $A$ and $B$ is $d_l(A, B) \rightarrow \langle a_l(A, B), r_l(A, B) \rangle$ with $a_l(A, B) = s_l(B) \setminus (s_l(A) \cap s_l(B))$ and $r_l(A, B) = s_l(A) \setminus s_l(A) \cap s_l(B))$. The calculation of a semantic diff can be accomplished by a language specific reasoner or by a language specific set of rules. These rules can be formulated in a language like TRIPLE as demonstrated in [?].

- Semantic conflict detection - the ontology language semantics determine what should be considered as a conflict e. g. if the result of a merge is an inconsistent ontology.

Additionally, the specific ontology versioning system might want to have a special diff encoding. Our system can be adopted by providing mapping rules between the RDF diff and the specific language encoded diff.

Further a specific versioning system can use the 'user defined metadata' functionality of SemVersion for storing specific metadata like access rights, degree of agreement, mappings between versions etc.

5.1 Examples

In order to version RDFS in a reasonable way, one needs a semantic diff between two RDF Schemas\textsuperscript{10}. Besides that, the SemVersion functions should be sufficient.

As a second example, we show briefly how SemVersion can be used to realize the MarcOnt scenario as stated in Section 2. SemVersion can manage different branches of versions (M1). Suggestions to the main branch are modelled as different branches, which can evolve separately (M2, M5). Snapshots of the main ontology with suggestions applied are created realized by merging the different branches and showing the user the merged version. As explained on page 9, some conflicts can be detected on the RDF layer (M4). For other conflicts, an ontology language specific semantic diff is required. Mappings between different versions can be stored as metadata of the version for which the backward-mapping is required (M6). As every version can be identified by an URL, it is easy to discuss about them, e. g. link to them in a forum. As URLs are also URIs one can also express arbitrary statements about them in RDF.

\textsuperscript{10} With $srdfs$ as defined in http://www.w3.org/TR/rdf-mt/#RDFSRules
6 Implementation

6.1 Architecture

The overall system architecture is depicted in Figure 8. SemVersion is designed as an HTTP-accessible server that handles the versioning business logic. Storage is delegated to an RDF storage layer. In this context, an RDF store does not deal with the semantics for RDF vocabularies like RDFS or OWL.

At startup time an SemVersion server loads its root data model from a configured RDF store and caches it in memory. In periodic intervals and at shutdown time the root model is written back to the RDF store. The root model contains information about projects, versioned models, their versions and other metadata. User-defined metadata is stored as separate RDF models in the RDF store. Only time stamps and branch labels are stored directly in the SemVersion root model. This reduces the SemVersion data layer to a clean layer with statements about versioning artefacts. Diffs are calculated on-the-fly in the SemVersion server, but could be cached.

6.2 Storage Layer Access

An ontology versioning system should scale in many dimensions. It should allow a large number and size of ontologies. This implies a scalable storage architecture. If the ontologies become large, it is undesirable to download them first and query or manipulate them locally. Thus a remote querying and manipulation facility is needed. There already exist scalable RDF stores with remote query and update functionality. SemVersion adds a small but smart layer of version management on top of such RDF stores. They are connected to SemVersion via the RDF Storage Protocol (RSP) over HTTP, which resembles the Java Map interface. The two basic methods are RDF/XML get(URI u) and put( RDF/XML model, URI u). The RSP is defined as a simple protocol over HTTP, inspired by the REST [?] architectural style. The method 'get' is e. g. mapped to HTTP GET http://url-of-rs-store/uri=<u>. The method 'put' is realized as HTTP POST. The architecture allows to use multiple, distributed RDF stores in parallel which are integrated via tiny RSP adapters.
6.3 Handling Commits

The new version will simply be stored – this guarantees that the retrieval will give the user back what she checked in. More sophisticated storage mechanisms could be developed, but the real challenge in ontology versioning is not storage space but the management of the distributed engineering processes within a heterogenous tool environment. The new model is send to the RDF store with a locally generated URI, which is globally unique.

6.4 Generating an infinite number of globally unique URIs

The strategy for generating globally unique URIs is as follows: (i) The first part of the URI is the URL the SemVersion server is running at. This reduces the problem of generating globally unique URIs to generating locally unique URIs, assuming that the same SemVersion server URL will not be used for different SemVersion server ever. To soften this constraint, (ii) the current system time for the server, measured in milliseconds is also made a part of the generated URL. Thus the problem is reduced to maintain an accurate server clock and never issue the same URI again in a given period of time (server clock may be off for minutes, but not months). To issue different URIs at all times, (iii) an internal counter is added to the URI string. The URI generator cannot guarantee uniqueness, but the likelihood for the same URI being generated twice is really low.

6.5 Application Programming Interface

The general trade-off between the power of a strongly typed, object-oriented API and the flexibility of having direct access to the underlying data exists as well in the RDF and Java world. The open-source project RDFReactor\textsuperscript{11}, which generates data manipulation classes from an RDF Schema\textsuperscript{12}, is used to give the user an object-oriented access for many common functions like adding projects, setting the parents of a version or storing the branch label. Alternatively the root RDF model can be used directly.

The SemVersionInterface (OVI) has two implementations. One is the real implementation, that calls directly the SemVersion core. The second implementations tunnels all calls over the Ontology Versioning Protocol (OVP), which exposes the OVI as an HTTP-based protocol. For convenience SemVersion comes with an OVP Client and OVP Server to bridge between OVP and Java. This architecture allows clients to work remotely over OVP or – via the OVP Client – with the OVI. Local applications can use the OVI directly.

\textsuperscript{11} \url{http://RDFReactor.OntoWare.org}
\textsuperscript{12} \url{http://SemVersion.ontoware.org/2004/12/datamodel}
7 Related Work

The very popular concurrent version system (CVS\textsuperscript{13}) initially was a collection of scripts to simplify the handling of the revision control system (RCS). RCS operates in a file-centric way by using a “lock-modify-unlock”-style. However, CVS works on the syntactical level, not on the conceptual. I.e., it is not capable of versioning objects and in particular not capable of versioning ontological entities and their complex structure. The underlying diff operation is capable of showing the syntactical differences between two files (based on the differences of text lines).

Following terminology from the database community (cf. [?]) we mainly distinguish between „ontology versioning” and „ontology evolution”. The difference between schema evolution and ontology evolution is shown in [?]. Ontology versioning is accommodated when an ontology management system allows for handling of ontology changes by creating and managing different versions of it. Ontology evolution is accommodated when an ontology management system facilitates the modification of an ontology by preserving its consistency.

Whereas we deal with ontology versioning, [?] is mostly concerned with ontology evolution. There, an ontology evolution process is defined and implemented in the KAON system which (i) enables the handling of continuously occurring ontology changes; (ii) ensures the consistency of the underlying ontology and all dependent artifacts (such as instances); (iii) supports the user to manage changes more easily; and (iv) offers advice to the user for continual ontology re-engineering.

A first survey on causes and consequences of changes in an ontology have been described in [?], followed by an implementation for ontology versioning (cf. [?]) that is based on the comparison of two ontology versions in order to detect changes. Even though there are many ways to transfer an ontology into a new version, this system generates only one solution based on the set of heuristics. It mainly addresses OWL and OKBC as representation languages and does not deal with RDF/S specific problems as we do.

Other related fields include schema evolution in databases (cf. e.g. [?], [?] etc.) and evolution of XML documents (cf. e.g. [?], [?]). Along with a critical discussion of the relationships between ontology evolution and these approaches one can also find a quite extensive amount of additional related work in [?].

8 Conclusion

We presented a methodology for RDF-based versioning that separates the management aspects from the versioning core functions. We presented blank node enrichment as a technique to identify blank nodes in models and diffs. We also explained how generic RDF versioning is beneficial for ontology language specific versioning. Finally we sketched our implementation of the system.

\textsuperscript{13} Available freely for download at \url{http://www.cvshome.org/}