Tracking Changes in RDF(S) Repositories

Atanas Kiryakov, Damyan Ognyanov

OntoText Lab, Sirma AI EOOD, 38A Chr. Botev blvd, 1000 Sofia, Bulgaria
{naso, damyan }@sirma.bg

Abstract. The real-world knowledge management applications require administrative features such as versioning, fine-grained access control, and meta-information to be supported by the back-end infrastructure. Those features together with the needs of the ontology maintenance and development process raise the issue of tracking changes in knowledge bases. Part of the research presented is as basic as defining the rules of the game, the proper formal models to build upon – what to count as a change and what to ignore, how to represent and manage the tracking information. A number of more "technical" issues such as tracking changes in imported and inferred knowledge are also discussed. The implementation is a part of the ontology middleware module developed under the On-To-Knowledge project where it is implemented as extension of the Sesame RDF(S) repository. This paper is a further development of the results reported in 7.

1. Introduction

The ontology middleware can be seen as an „administrative" software infrastructure that makes the rest of the modules in a KM toolset easier for integration in real-world applications. The central issue is to make the tools available to the society in a shape that allows easier development, management, maintenance, and use of middle-size and big knowledge bases[1]. The following basic features are considered:

- Versioning (tracking changes) of knowledge bases;
- Access control (security) system;
- Meta-information for knowledge bases.

These three aspects are tightly interrelated among each other as depicted on the following scheme.

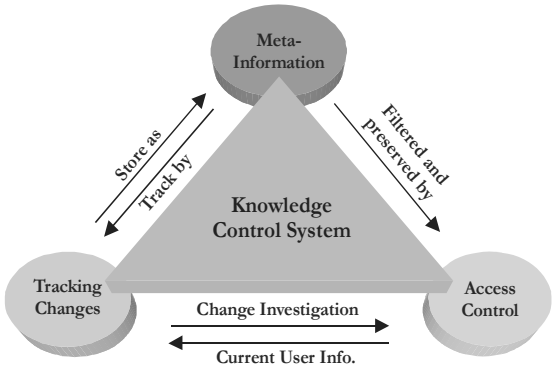


Figure 1: Knowledge Control System

[1] See the next sub-section for discussion on ontology vs. instance data vs. knowledge base.

The composition of the three functions above represents a Knowledge Control System (KCS) that provides the knowledge engineers with the same level of control and manageability of the ontology in the process of its development and maintenance as the source control systems (such as CVS) provide for the software. However, KCS is not only limited to support the knowledge engineers or developers – from the perspective of the end-user applications, KCS can be seen as equivalent to the database security, change tracking (often called cataloguing) and auditing systems. A KCS is carefully designed to support these two distinct use cases.

Further, an ontology middleware system should serve as a flexible and extendable platform for knowledge management solutions. It should provide infrastructure with the following features:

- A repository providing the basic storage services in a scalable and reliable fashion. Such example is the Sesame RDF(S) repository, 2;
- Multi-protocol client access allowing different users and applications to use the system via the most efficient transportation media;
- Knowledge control – the KCS introduced above;
- Support for plugable reasoning modules suitable for various domains and applications. This ensures that within a single enterprise or computing environment one and the same system may be used for various purposes (that require different reasoning services) so providing easy integration, interoperability between applications, knowledge maintenance and reuse.

In the rest of this introductory section we define better the scope of our research and the terminology used. Section 2 is dedicated to an overview on tracking changes in RDF(S) repositories – related work and principles. The design and implementation approach of the change tracking and versioning is presented in Section 3 and Section 4 covers the meta-information and it's tracking followed by an implementation approach and formal representation respectively in Sections 5 and 6. Future work and conclusion are presented in the last section.

The work presented here was carried as part of the On-To-Knowledge project. The design and implementation of our ontology middleware implementation is just an extension of the Sesame architecture (see 2) that already covers many of the desired features. Earlier stage of the research is presented in bigger details in 7 where the reader can find more about the Access control system (security), which is out of the scope of this paper.

1.1. Scope, Ontologies vs. Knowledge Bases

A number of justifications in the terminology are necessary. An almost trivial but still relevant question is "What the KM tools support: ontologies, data, knowledge, or knowledge bases?" Due to the lack of space we are no going in to comment this basic notions here. A simple and correct answer is "All of this". The ontology middleware module extends the Sesame RDF(S)

repository that affects the management of both ontologies and instance data in a pretty much unified fashion.

For the purpose of compliance with Sesame, here the term repository will be used to denote a compact body of knowledge that could be used, manipulated, and referred as a whole. Such may contain (or host) both ontological assertions and instance data.

2. Overview

The problem for tracking changes within a knowledge base is addressed in this section. It is important to clarify that higher-level evaluation or classification of the updates (considering, for instance, different sorts of compatibility between two states or between a new ontology and old instance data) is beyond the scope of this work. Those are studied and discussed in 3, sub-section 2.2. The tracking of the changes in the knowledge (as discussed here) provides the necessary basis for further analysis. For instance, in order to judge the compatibility between two states of an ontology a system should be able to at least retrieve the two states and/or the differences between them.

In summary, the approach taken can be shortly characterized as ”versioning of RDF on a structural level in the spirit of the software source control systems”.

2.1. Related Work

Here we will shortly review similar work, namely, several other studies related to the management of different versions of a complex objects. In general, although some of the sources discuss closely related problems and solutions, there is not one addressing ontology evolution and version management in a fashion that allows granularity down to the level of statements (or similar constructs) and capturing interactive changes in knowledge repositories such as asserting or retracting statements.

One of the studies that provide a methodological framework pretty close to the one need here is 8. The authors model a framework, which is designed to handle the identification, control, recording, and tracking of the evolution of software products, objects, structures, and their interrelationships. The paper investigates the different models and versioning strategies for large scale software projects and present a way to express the meta-information and the impact of a single change over the various components of the project in RDF(S) – in this case used just for representation of the related meta-information, the objects being tracked are from the software domain.

Database schema evolution and the tasks related to keeping schema and data consistent to each other can be recognized as very similar to ours. A detailed and pretty formal study on this problem can be found in 4 – it presents an approach allowing the different sorts of modifications of the schema to be expressed within suitable description logic. More detailed information about the reasoning and other related tasks could be found in 5.

Another study dealing with the design of a framework handling database schema versioning is presented in 1. It is similar to 4 and 5 and can be seen as a different approach of handling the changes of the evolving object and the process of class evolution.

Probably the most relevant work was done under the On-To-Knowledge project – among the reports concerning various aspects of the knowledge management, most relevant is 3, mentioned earlier in this section.

3. Versioning Model for RDF(S) Repositories

A model for tracking of changes, versioning, and meta-information for RDF(S) repositories is proposed. To make it more explicit (i) the knowledge representation paradigm supported is RDF(S) and (ii) the subject of tracking are RDF(S) repositories – independently from the fact if they contain ontologies, instance data, or both. The most important principles are presented in the next paragraphs.

VPR1: *The RDF statement is the smallest directly manageable piece of knowledge.*

Each repository, formally speaking, is a set of RDF statements (i.e. triples) – these are the smallest separately manageable pieces of knowledge. There exist arguments that the resources and the literals are the smallest entities – it is true, however they cannot be manipulated independently. It is the case that none of them can independently “live” in a repository because they always appear as a part of a triple and never independently. The moment when a resource was added to the repository may only be defined indirectly as the same as “the moment when the first triple including the resource was added”. Analogously a resource may be considered removed from a repository when the last statement containing it gets out. To summarize, there is no way to add, remove, or update (the description of) a resource without also changing some statements while the opposite does not hold. So, the resources and the literals from a representational and structural point of view are dependent on the statements.

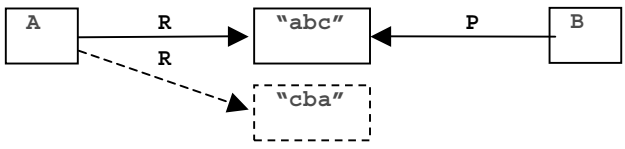
VPR2: *An RDF statement cannot be changed – it can only be added and removed.*

As far as the statements are nothing more than triples, changing one of the constituents, just converts it into another triple. It is because there is nothing else but the constituents to determine the identity of the triple, which is an abstract entity being fully defined by them. Let us take for instance the statement **ST1=<A, PR1, B>** and suppose B is a resource, i.e. an URI of resource. Then ST1 is nothing more but a triple of the URIs of A, PR1, and B – if one of those get changed it will be already pointing to a different resource that may or may not have something in common with the first one. For example, if the URI of A was **http://x.y.z/o1#A** and it get changed to **http://x.y.z/o1#C** then the statement **ST2=<C,PR1,B>** will be a completely different statement.

Further, if the resource pointed by an URI gets changed two cases could be distinguished:

- The resource is changed but its meta-description in RDF is not. Such changes are outside the scope of the problem for tracking changes in formally represented knowledge, and particularly in RDF(S) repositories.
- The description of the resource is changed –this can happen iff a statement including this resource get changed, i.e. added or removed. In such case, there is another statement affected, but the one that just bears the URI of the same resource is not.

There could be an argument, that when the object of a triple is a literal and it gets changed, this is still the same triple. However, if



there is for instance statement **<A,R,”abc”>** and it get changed to **<A,R,”cba”>**, the graph representation shows that it is just a different arc because the new literal is a new node and there could

be other statements (say, `<B,P,"abc">`) still connected to the note of the old literal. As a consequence here comes the next princple:

VPR3: *The two basic types of updates in a repository are addition and removal of a statement*

In other words, those are the events that necessarily have to be tracked by a tracking system. It is obvious that more event types such as replacement or simultaneous addition of a number of statements may also be considered as relevant for an RDF(S) repository change tracking system. However, those can all be seen as composite events that can be modeled via sequences of additions and removals. As far as there is no doubt that the solution proposed should allow for tracking of composite events (say, via post-processing of sequence of simple ones), we are not going to enumerate or specify them here.

VPR4: *Each update turns the repository into a new state*

Formally, a state of the repository is determined by the set of statements that are explicitly asserted. As far as each update is changing the set of statements, it is also turning the repository into another state. A tracking system should be able to address and manage all the states of a repository.

3.1. History and Versions

Some notes and definitions that complement the above stated principles are presented below.

History, Passing through Equivalent States

The history of changes in the repository could be defined as sequence of states, as well, as a sequence of updates, because there is always an update that turned repository from one state to the next one. It has to be mentioned that in the history, there could be a number of equivalent states. It is just a question of perspective do we consider those as one and the same state or as equivalent ones. Both perspectives bear some advantages for some applications. We accepted that there could be equivalent states in the history of a repository, but they are still managed as distinct entities. Although it is hard to provide formal justification for this decision the following arguments can be presented:

- For most of the applications it is not typical a repository to pass through equivalent states often. Although possible, accounting for this phenomenon does not obviously worth taking into account that finding (or matching) equivalent states could be a computationally very heavy task.
- It is the case that, if necessary, equivalent states could be identified and matched or combined via post-processing of a history of a repository.

Versions are labeled states of the repository

Some of the states of the repository could be pointed out as versions. Such could be any state, without any formal criteria and requirements – it completely depends on the user's or application's needs and desires. Once defined to be a version, the state becomes a first class entity for which additional knowledge could be supported as a meta-information (in the fashion described below.)

4. Meta-Information

Meta-information is supported for the following entities: resources, statements, and versions. As far as DAML+OIL ontologies are also formally encoded as resources (of type daml:Ontology) meta-information can be attached to them as well.

4.1. Model and Representation of the Meta-Information

We propose the meta-information to be modeled itself in RDF – something completely possible taking into account the unrestricted meta-modeling approach behind RDF(S). A number of objections against such approach can be given:

- It increases the number of meta-layers and so it makes the representation more abstract and hard to understand. However, adding meta-information always requires one more layer in the representation, so, making it via extension of the same primitives used for the "real data" (instead of defining some new formalization) can even be considered as a simplification.
- It makes possible confusion and may introduce technical difficulties, say, because of intensive use of heavy expressive means such as reification.

The schema proposed below handles in some degree these problems and provides number advantages:

- It is probably the most typical role of RDF to be used for encoding of meta-information
- One and the same technology can be used for viewing, editing, and management of both knowledge and meta-information. Any RDF(S) reasoners and editors will be able to handle meta-information without special support for it.
- Queries including both knowledge and meta-information will be pretty straightforward. So, lookup of knowledge according to conditions involving both meta-information and "real" knowledge is possible. Imagine a situation, when a complex ontology is being developed and there is meta-information supporting this process, say, a meta-property "Status" (with possible values "New", "Verified against the ontology", "Verified against the sample data", "Done") being attached to each class. Then a lookup of all classes that are subclasses of C and have status "New" will be just a typical query against the RDF(S) repository.
- Representing the meta-information as RDF could be done in a flexible way that allows it to be customized for the specific needs of the use case.

4.2. Tracking Changes in the Meta-Information

An important decision to be taken is whether changes in the meta-information should be tracked. The resolution proposed here is: Changes in the meta-information should be considered as regular changes of the repository, so, they turn it from one state to another. Here are few arguments backing this position:

- There are number of cases when the only result of a serious work over an ontology is just a single change in the meta-information. Let is use again the example with the "Status" meta-property for classes (described above.) The result of a complex analysis of the coherence of a class definition may result just in changing the status from "New" to one of the other values. In such case, although there is no formal change in the "real" data, something important get changed. From an ontology development and maintenance point of view it is extremely important tracking of such changes to be possible.
- If necessary, it is possible appropriate analysis to be made so that changes that affect only meta-information to be ignored. This way both behaviors can be achieved. In case of the

opposite decision (not to track changes in meta-information), no kind of analysis can reconstruct the missing information.

- An analogy with the software source control systems may also provide additional intuition about this issue. If we consider the comments in the software code as a meta-information, it becomes clear that the source control systems definitely account the changes in the meta-information as equal to the "real" changes in the code.

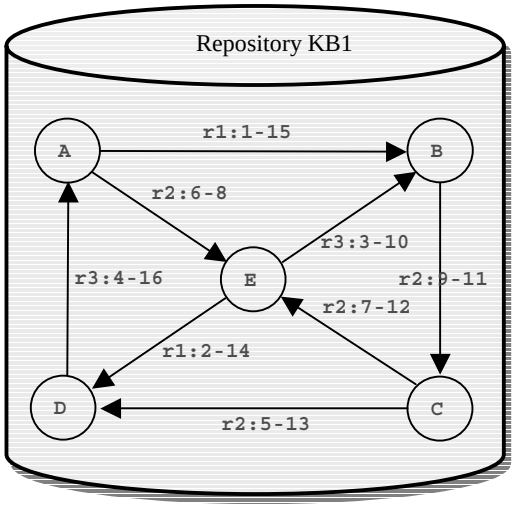
5. Implementation Approach

Let us first propose the technical schema for tracking changes in a repository. For each repository, there is an *update counter* (UC) – an integer variable that increases each time when the repository is updated, that in the basic case means when a statement get added to or deleted from the repository. Let us call each separate value of the UC *update identifier*, *UID*. Then for each statement in the repository the UIDs when it was added and removed will be known – these values determine the "lifetime" of the statement. It is also the case that each state of the repository is identified by the corresponding UID.

The UIDs that determine the "lifetime" of each statement are kept, so, for each state it is straightforward to find the set of statements that determine it – those that were "alive" at the UID of the state being examined. As far as versions are nothing more than labeled states, for each one there will be also UID that uniquely determines the version.

The approach could be demonstrated with the sample repository KB1 and its "history". The repository is represented as a graph – each edge is an RDF statement which lifetime is given separated with semicolons after the property name. The history is presented as a sequence of events in format

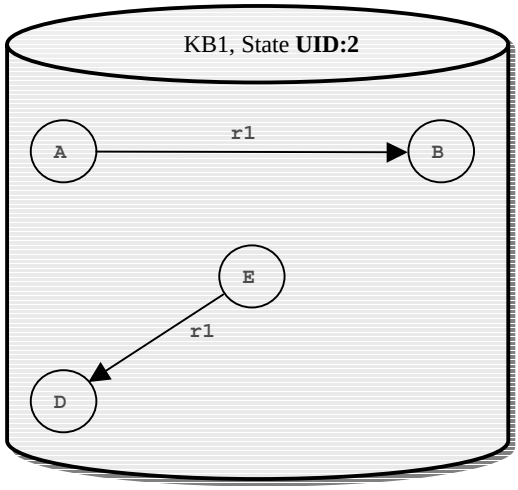
```
UID:nn {add|remove} <subj, pred, obj>
```

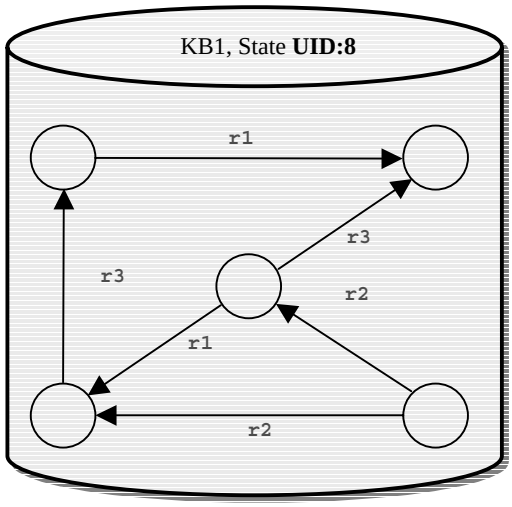


History:

```
UID:1 add <A, r1, B>
UID:2 add <E, r1, D>
UID:3 add <E, r3, B>
UID:4 add <D, r3, A>
UID:5 add <C, r2, D>
UID:6 add <A, r2, E>
UID:7 add <C, r2, E>
UID:8 remove <A, r2, E>
UID:9 add <B, r2, C>
UID:10 remove <E, r3, B>
UID:11 remove <B, r2, C>
UID:12 remove <C, r2, E>
UID:13 remove <C, r2, D>
UID:14 remove <E, r1, D>
UID:15 remove <A, r1, B>
UID:16 remove <D, r3, A>
```

Here follow two "snapshots" of the states of the repository respectively for UIDs 2 and 8





It is an interesting question how we handle in the above model, multiple additions and removals of one and the same statement, which in a sense periodically appears and disappears form the repository. We undertake the approach to consider the appearance of such statement as separate statements, because of reasons similar to those presented for the support of distinguishable equivalent states of the repository.

5.1. Batch Updates

We call *batch update* the possibility the update counter of the repository to be stopped, so not to increment its value for a number of consequent updates. This feature is important for cases when it does not make sense the individual updates to be tracked one by

one. Such example could be an assertion of a DAML+OIL element that is represented via set of RDF statements none of which can be interpreted separately.

Another example for a reasonable batch update would be an application that works with the repository in a transactional fashion – series of updates are bundled together, because according to the logic of the application they are closely related. Finally, batch updates can also be used for file imports (see subsection 5.4.).

5.2. Versioning and Meta-information for Imported Statements

New statements can appear in the repository when an external ontology is imported in the repository either by `xmlns:prefix="uri"` attribute of an XML tag in the serialized form of the ontology either by `daml:imports` statement found in the header of a DAML+OIL ontology. In each of those cases the statements imported in the repository are treated as read-only and thus the users cannot change them. All these statements will be added and removed to/from the repository simultaneously with the statement that causes their inference or import. An additional note about the imported statements related to the security: these statements should be recognized as external, and not belonging to the repository and thus we can avoid the application of the security policies to them. Meta-information may not be attached to such statements.

5.3. Versioning and Meta-information for Inferred Statements

There are cases when addition of a single statement in the repository leads to the appearance of several more statements in it. For example, the addition of the statement `ST1=<B, rdfs:subClassOf, C>` leads to the addition of two new statements `ST2=<B, rdf:type, rdfs:Class>` and `ST3=<C, rdf:type, rdfs:Class>`. This is a kind of simple inference necessary to "uncover" knowledge that is implicit but important for the consistency of the repository. There are number of such inferences implemented in Sesame down the lines of 10.

The question about the lifetime of such inferred statements is far not trivial. Obviously, they get born when inferred. In the simplest case, they should die (get removed) together with the statement that caused them to be inferred. However, imagine that after the addition of `ST1` in the repository, there was another statement added, namely `ST4=<B, rdfs:subClassOf, D`>. As far, as `ST2` is already in the repository only `ST5=<D, rdf:type, rdfs:Class>` will be inferred and added. Now, imagine `ST1` is deleted next while `ST4` remains untouched. Should we delete `ST2`? It was added together with `ST1` on one hand, but on the other it is also "supported by" `ST4`. One approach for resolving such problems is the so-called "truth maintenance systems" (TMS) – basically, for each statement information is being kept about the statements or expressions that "support" it, i.e. such that (directly) lead to its inference. Sesame currently implements such a TMS.

Suppose, there is a TMS working in Sesame (because it could be "switched off" for performance reasons), the tracking of the inferred statements is relatively easy. When the TMS "decides" that an inferred statement is not supported anymore, it will be deleted – this is the natural end of its lifetime. It will be considered as deleted during the last update in the repository, which automatically becomes a sort of batch update (if it is not already.)

This is the place to mention that the pragmatics of the remove operation in an RDF(S) repository is a bit not obvious. When a statement is removed this only means that it is not explicit any more. However, if it follows from other statements, it effectively remains in the repository and there is no way to remove it, without removing all of its "supporters". Imagine a situation when one statements was inferred on update U1, next explicitly asserted on updated U4, next deleted at U6, but still supported by other sentences until U9. Than the life time of the statement is U1-U9.

For many applications, the "explicitness" of the statements bear significant importance, so, in our implementation we keep track for this – for each state it is not only possible to uncover the statements that were "alive", but also which of them were explicit.

As with the imported statements, meta-information may not be attached to inferred statements. The security restrictions towards inferred statements can be summarized as follows:

- Inferred statements may not be directly removed;
- A user can read an inferred statement iff s/he can read one of the statements that support it.
- The rights for adding statements are irrelevant – a user may or may not be allowed to add a statement independently from the fact is it already inferred or not.

5.4. Versioning of Knowledge Represented in Files

The issues concerning the work with knowledge represented in files and its versioning can be discussed in two main topics – each of them presenting a different involvement of the content of the files.

The first one is the case when the Sesame uses a specific storage and inference layer (SAIL) to access knowledge directly from the underlying file – so files used for persistency instead of, say, relational database. In such case we cannot control the appearance and disappearance of distinct statements, which easily can happen independently from Sesame. The knowledge control system (KCS) presented here is not applicable for such SAILs.

The second case is to import knowledge into the Sesame from files – one of the typical usage scenarios. The first step than is to convert the file `F` into a set of statements `FS`, which also includes the inferred ones. Next, the appropriate changes are made in the repository within a single batch update (see subsection 5.1.) Three different modes for populating repository from files are supported:

- **Re-initializing** – the existing content of the repository is cleared and the set of statement `FS` is added. No kind of tracking or meta-information is preserved for the statements that were in the repository before the update. This is equivalent to Clear followed by Accumulative import;
- **Accumulative** – `FS` is added to the repository, it actually means that the statements from `FS` that are already in the repository are ignored (any tracking and meta-information for them remains unchanged) and the rest of the statements are added. This type of import has to be used carefully, because it may lead to inconsistency of the repository even if its previous state and the file were consistent on their own;
- **Updating** – after the import the repository contains only the statements form the file, the set `FS` (as in the re-initializing mode). The difference is that the statements from the repository that were not in `FS` are deleted but not cleared, i.e. after the update, they are still kept together with their tracking and meta-information. The statements from `FS` that were not already in the repository[2] are added.

The Updating import mode is the most comprehensive one and allows the repository to be used to track changes in a file that is being edited externally and "checked-in" periodically. This can

[2] Actually those which are not "alive".

also be used for outlining differences between versions or different ontologies represented in files.

5.5. Branching Repositories

Branching of states of repositories is possible. In this case a new repository is created and populated with a certain state of an existing one – the state we want to make branch of. When a state is getting branched, it will automatically be labeled as a version first. The appropriate meta-information indicating that this version was being used to create a separate branch of the repository into a new one will be stored.

As it can be expected, no kind of operations with the branch of the repository will affect the original one. Branches have to be used, for instance, in cases when the development of an ontology have to be split into two separate processes or threads. A typical situation when this is necessary is when an old version has to be supported (which includes making small maintenance fixes) during the development of a new version. The state, which is used in production, can be branched and the development of the new one can take place in the branch while at the same time, the old version can still be supported and updated.

5.6. Controlling the History

> *Those who control the past, control the future;*
> *Those who control the future, control the present;*
> *Those who control the present, control the past.*
> *1984, George Orwell*

The possibility for tracking changes in a repository and gathering history has an important consequence: the history has to be controlled! The most important reasons for this and the appropriate mechanisms are discussed here.

Reason 1: *The volume of the data monotonously grows.* As far as nothing is really removed from the repository (the statements are only marked as ”dead”) all the statements that were ever asserted in the repository together with their tracking and meta-information can be expected to be preserved forever. This way the volume of the data[3] monotonously grows with each update.

Reason 2: *The history may need refinement.* The automatic tracking of the changes allows a great level of control, however the fine-grained information may also be obsolete or confusing. For instance, after the end of a phase of development of an ontology, the particular updates made in the process may become unimportant – often it is the case that finally the differences with a specific previous state (say, a version) are those that count.

Therefore the following methods control over the tracking information are implemented:

- *Clear the history before certain state* – all the statements died before this state (say **`S1`**), together with any kind of tracking and meta-information about them will be permanently removed from the repository. The same applies for the labeled versions before this state. All values of the update counter form a kind of “calendar” and all changes are logged against it. There will be two options for managing the tracking information for statements that were ”born” before **`S1`** and died after it. Under the first option, they will be stated to be born at a special moment ”before the Calendar” and all calendar records before **`S1`** will also be deleted. Under the second option, the ”calendar” will be preserved, so no changes will be introduced to the tracking information for those statements that remain in the repository and the Calendar will be cleared from all the records that are not related to such statements.
- *Aggregate updates* – a number of sequential updates (say, between **`UID1`** and **`UID2`**) to be merged and made equivalent to specified one, say **`UID3`**. In this case, all references to UIDs between **`UID1`** and **`UID2`** will be replaced with **`UID3`**, which may or may not be equal to **`UID1`** or **`UID2`**.

[3] The number of the statements and resources.

6. Formal Representation of the Meta-Information

All the Knowledge Control System (KCS) related information would be represented in RDF according to a schema that could be found at: http://www.ontotext.com/otk/2002/03/kcs.rdfs. That includes tracking, versioning, and security information as well as user-defined meta-information. It is important to acknowledge that although this schema provides a well-structured conceptual view to the meta-information, its support by a repository is not be implemented directly after this schema because of obvious performance problems. So, the schema presents the way this information can be imported, exported, and accessed via RQL queries. It also facilitates good formal understanding of the supported model.

The basic idea is that all the meta-information is encoded via kind of special, easily distinguishable, properties – namely such defined as sub-properties of a **`kcs:metaInfo`**. Also, all the related classes are defined as sub-classes of **`kcs:KCSClass`**. Here follows the set of pre-defined meta-properties, mostly related to the tracking information The hierarchy of the properties is presented together with the domain and range restrictions for each property:

```
metaInfo
  trackingInfo (domain=rdfs:Statement range=)
        bornAt (domain=rdfs:Statement range=Update)
        diedAt (domain=rdfs:Statement range=Update)
     securityInfo
        lockedBy (domain=rdfs:Statement range=User)
```

The **`bornAt`** and **`diedAt`** properties define the lifetime of the statement via references to the specific updates. In similar manner we express the information associated with each particular update – the user who made it, the actual time and, etc.

Obvious extensions of the above schema are the Dublin Core primitives – there is no problem those to be declared to be sub-properties of **`metaInfo`**. The above-proposed model has number of advantages:

- Flexibility. Various types of meta-information could be defined – the appropriate schema has to be created with the only requirement the properties there to be defined as sub-properties of **`metaInfo`** and the classes as sub-classes of **`MetaInfoClass`**.
- The different meta-properties may have their appropriate domain and range restrictions.
- It is easy to “strip” or preserve the meta-info, just ignoring the statements which predicates are sub-properties of **`metaInfo`** and the resources of class **`MetaInfoClass`**.

6.1. Meta-Information for Statements

Assigning meta-information to statements is trickier than to resources at least because there are no URIs to identify each

statement. For each statement that we like to attach a meta-information we need to apply a sort of explicit- or pseudo-reification in order to associate the required meta-properties with the appropriate instance of **rdf:Statement**. A special class is defined in the KCS schema for this purpose:

```
<rdfs:Class rdf:about="&kcs;StatementMetaInfo"
    rdfs:label="StatementMetaInfo">
    <rdfs:comment>
        A common super-class for all the
        meta-information about statements …
    </rdfs:comment>
    <rdfs:subClassOf
        rdf:resource="&rdf;Statement"/>
    <rdfs:subClassOf
        rdf:resource="&kcs;MetaInfoClass" />
</rdfs:Class>
```

When we need to associate some meta-information to a statement we can instantiate **kcs:StatementMetaInfo** for that statement directly referring to its subject, predicate and object. Here is an example of such instantiation:

```
<kcs:StatementMetaInfo rdf:about="&mex;status1">
    <rdf:subject rdf:resource="&mex;Jim" />
    <rdf:predicate rdf:resource="&mex;childOf" />
    <rdf:object rdf:resource="&mex;John" />
    <rdfs:comment>Comment on statement
        (Jim,childOf,John)</rdfs:comment>
    <mex:customMetaProp>customMetaProp on
        statement (Jim,childOf,John)
    </mex:customMetaProp>
</kcs:StatementMetaInfo>
```

In this case, two pieces of meta-information are attached to the **<Jim,childOf,John>** statement. The first one is just a comment encoded using the standard **rdfs:comment** property. The second one is a sample for custom meta-property, which was defined by the user. A full working[4] example that demonstrates how meta-information for statements can be encoded is presented in: http://www.ontotext.com/otk/statement_metainfo_ex.rdf.

We can easily extract all the meta-information about specific statement using an RQL query. To do that we need the subject, predicate and object of the statement – it is sad but true, there is no other standard way to refer to or specify a triple. A sample query retrieving the meta-properties and comments about the statement **<Jim, childOf, Jonh>** from the above example may looks like:

```
select
  @metaProp, result
from
  {X : $CX } &rdfs;subject {A},
  {X : $CX } &rdfs;predicate {B},
  {X : $CX } &rdfs;object {C},
  {X : $CX } @metaProp {result}
where
  A=&mex;Jim and B=&mex;childOf and C=&mex;John
and
  ( @metaProp=&rdfs;comment or @metaProp <
    &kcs;metaInfo )
and  $CX > &rdf;Statement
```

The above is a bit simplified syntax; the following replacements should take place to make it real:

```
&rdf; -> http://www.w3.org/1999/02/22-rdf-syntax-
ns#
&rdfs; -> http://www.w3.org/2000/01/rdf-schema#
&kcs; ->
http://www.ontotext.com/otk/2002/03/kcs.rdfs#
&mex; ->
http://www.ontotext.com/otk/statement_metainfo_ex.
rdf#
```

6.2. Low-level Representation of Meta-Information for Statements

Although conceptually clear, the above model for keeping meta-information (including tracking data) for statements has at least the following problems:

- Technically speaking, it requires reification, which is the most criticized RDF(S) feature, also not supported by many tools;
- For each statement of "real" data, there should be five statements tracking it (one for each of the sub-properties of **kcs:trackingInfo** and three more that define the appropriate updates), i.e. the volume of the tracking data is five times (!) bigger than the volume of the repository without it.

In order to resolve this issues, the KCS meta-information is actually stored and queried internally using more appropriate encoding – the RDF(S) engine takes care to support some level of mimicry to preserve the abstraction of the above presented schema for the applications.

A simple database schema is presented next to provide a general idea for possible implementation of a KCS – the real implementation uses a bit more complex schema which still follows the same ideas. Suppose, an RDF(S) repository works on top of an RDBMS (which is the case with SESAME) and there is a table **Statements** with columns **Subject**, **Predicate** and **Object** each row of which represents a single statement. Few more columns could be added to this table with references to the tables with security and tracking information. This way the problems with the volume and performance will be resolved at least with respect to the most critical use cases.

[4] When the KCS schema and the example are loaded in a Sesame repository, the queries presented in this sub-section really work and can be used to extract meta-information about the statements.

Statements						
SID	Subject	Predicate	Object	Born UID	Died UID	Lock
…	…	…	…			
11	RefA	Refr2	RefE	6	8	Usr5
12	RefB	Refr3	RefE	4	10	…
13	RefA	Refr1	RefB	6	9	…
14	RefD	Refr1	RefF	7	11	…
…	…	…	…			

Updates		
UID	Time	User
…	…	…
6	…	Usr3
7	…	Usr5
8	…	Usr3
9	…	Usr6
…	…	…

The Updates table can keep the information relevant to each update: the time when it happened and the user who performed the update (this information can easily be extended on demand.) In the Statements table, for each statement, the UID when it appeared and disappeared is kept as a reference to the Updates table.

With respect to the tracking of the changes, design as the one proposed above has a number of advantages compared to a variant where the update information is kept directly in the Statements table:

- All the necessary information is kept;
- The tracking information is not messing with the "real" information about the statements, however when necessary the two tables can be easily joined;
- The most basic operations (that are expected to be performed most frequently) can be done over the Statements table without need for joining with the Updates table. Such operation is to take all the statements that were "alive" at certain state identified by UID;
- There is significant reduction of the volume of the tracking information in cases of batch updates when multiple statements are getting added or removed at once and thus refer to one and the same UID.

7. Conclusion and future work

The ontology middleware, part of which is tracking changes module presented still have to prove itself in real-world applications. At this stage it is work in progress inspired by the methodology, tools, and case studies developed under the On-To-Knowledge project.

We see two interesting areas for development. First, the tracking changes mechanism will be used under the OntoView (see 12) project to serve as a basis for development of the higher-level ontology versioning services. Second, the whole ontology middleware will be applied in language engineering domain for management of linguistic and world knowledge for the purposes of information extraction and ontology extraction application.

To achieve the first goal, we are already working for implementation of the OntoView - ambitious ontology versioning portal. On the linguistic front, the Ontology Middleware module will be integrated with GATE (see 11 and http://gate.ac.uk) – one of the most mature platforms for language engineering that already provides substantial support of ontology-based annotations and integration of lexical knowledge bases such as WordNet.

8. References

1. Boualem Benatallah, Zahir Tari. *Dealing with Version Pertinence to Design an Efficient Schema Evolution Framework.*In: Proceedings of a "International Database Engineering and Application Symposium (IDEAS'98)",pp.24-33,Cardiff, Wales, U.K. July 8-10 1998
2. Jeen Broekstra, Arjohn Kampman. *Sesame: A generic Architecture for Storing and Querying RDF and RDF Schema.*Deliverable 9, On-To-Knowledge project, October 2001. http://www.ontoknowledge.org/downl/del10.pdf
3. Ying Ding, Dieter Fensel, Michel Klein, Borys Omelayenko. *Ontology management: survey, requirements and directions.* Deliverable 4, On-To-Knowledge project, June 2001. http://www.ontoknowledge.org/downl/del4.pdf
4. Enrico Franconi, Fabio Grandi, Federica Mandreoli. *Schema Evolution and Versioning: a Logical and Computational Characterization.* In "Database schema evolution and meta-modeling" - Ninth International Workshop on Foundations of Models and Languages for Data and Objects, Schloss Dagstuhl, Germany, September 18-21, 2000. LNCS No. 2065, pp 85-99
5. Enrico Franconi, Fabio Grandi, Federica Mandreoli. *A Semantic Approach for Schema Evolution and Versioning of OODB.* Proceedings of the 2000 International Workshop on Description Logics (DL2000), Aachen, Germany, August 17 - August 19, 2000. pp 99-112
6. Atanas Kiryakov, Kiril Iv. Simov, Marin Dimitrov. *OntoMap - the Guide to the Upper-Level.*In: Proceedings of the International Semantic Web Working Symposium (SWWS), July 30 - August 1, 2001, Stanford University, California, USA.
7. Atanas Kiryakov, Kiril Iv. Simov, Damyan Ognyanov. *Ontology Middleware: Analysis and Design.* Deliverable 38, On-To-Knowledge project, March 2002.
8. Supanat Kitcharoensakkul and Vilas Wuwongse. *Towards a Unified Version Model using the Resource Description Framework (RDF).* International Journal of Software Engineering and Knowledge Engineering (IJSEKE), Vol. 11, No. 6 (December 2001)
9. W3C; Ora Lassila, Ralph R. Swick, eds. *Resource Description Framework (RDF) Model and Syntax Specification.* http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/
10. Patrick Hayes, 2001. *RDF Model Theory*. W3C Working Draft. http://www.w3.org/TR/rdf-mt/
11. Hamish Cunningham. 2001. *GATE, a General Architecture for Text Engineering.* Computing and the Humanities. (to appear).
12. Michel Klein, Atanas Kiryakov, Dieter Fensel, Damyan Ognyanov. *Finding and characterizing changes in ontologies. Proceedings of the 21st* International Conference on Conceptual Modeling – ER 2002, October 7-11, 2002, Tampere, Finland. (to appear)