VERSION CONTROL FOR RDF TRIPLE STORES

Steve Cassidy and James Ballantine
Department of Computing, Macquarie Univeristy, Sydney, Australia
Steve.Cassidy@mq.edu.au

Keywords: RDF Version Control Annotation.

Abstract: RDF, the core data format for the Semantic Web, is increasingly being deployed both from automated sources and via human authoring either directly or through tools that generate RDF output. As individuals build up large amounts of RDF data and as groups begin to collaborate on authoring knowledge stores in RDF, the need for some kind of version management becomes apparent. While there are many version control systems available for program source code and even for XML data, the use of version control for RDF data is not a widely explored area. This paper examines an existing version control system for program source code, Darcs, which is grounded in a semi-formal *theory of patches*, and proposes an adaptation to directly manage versions of an RDF triple store.

1 INTRODUCTION

Semantic web applications store core data as RDF triples. In many cases, this data is authored manually in some form or other; as such, it can be seen as a form of program code and should be subject to the same kind of management. In particular keeping track of different versions of a code-base is especially important when more than one author is involved. Tools are needed to allow authors to synchronise their working copies, to exchange updates with each other, and to allow changes to be undone if necessary when problems are discovered. While it would be possible to manage RDF under an existing text-based version control system, it would be hard to guarantee consistency and correctness of a triple store managed in this way. This paper explores a novel model of version control embodied in the Darcs Roundy (2006) system to manage RDF data in the triple store.

The context of this work is a project to support Linguistic Annotation Cassidy and Harrington (2000); Bird and Liberman (2000) using RDF as the back-end data store. This application is similar in some ways to the Annotea and Vannotea projects R. Schroeter (2003) but the annotations themselves are often inter-related and represent linguistic interpretation of source data rather than comments and opinions on media. Our current work is aimed at providing a collaborative annotation facility to allow groups to develop and share annotations over the web. The provision of a version control mechanism will ensure that different copies of the same annotations can be kept synchronised where appropriate and that changes and updates to annotations can be managed. In our application, it is common for a community to settle on a 'correct' annotation through negotiation; hence the RDF store will commonly be modified over time with contributions from different parties. Even though our use of RDF isn't entirely aligned with its more general use in the Semantic Web, we feel that the need for version control for RDF is broader than just our project and that the proposals in this paper will have more general applicability.

2 BACKGROUND

A version control system is used to manage changes to a body of work, for example a set of source code documents in a project. Its major function is to record changes such that the changes can be rolled back and changes made by different authors can be

merged into a single code-base. The most common application for version control is the management of changes in program source code and here the most widely used system is CVS Cederqvist (2003) although more recently competitors such as Subversion Collins-Sussman et al. (2004) have been developed.

The majority of version control systems record different versions of documents in their store, optimising the store to make common operations inexpensive. For example, CVS stores the most recent version of each file and a set of patch files that encode the changes needed to revert to the previous version. Stepping back to the previous version is fast, but stepping back a long way requires the sequential application of patch files.

A recent development in version control has been the use of a distributed development model where multiple copies of the version control repository are maintained. In the CVS model, there is one central canonical copy of the version history that all developers use to send and receive updates. In the distributed model supported by Darcs, Arch Lord (2006) and Bitkeeper[1] among others, each developer has a copy of the version history and changes can be registered locally and passed between developers without the need for a central server.

2.1 Version Control and RDF

The On-to-Knowledge project has published a number of papers on managing changes in RDF ontologies Klein et al. (2002); Kiryakov and Ognyanov (2002). The focus of much of their work is on the description of different versions of ontologies although Kiryakov and Ognyanov (2002) note that much of their methodology applies equally to plain RDF descriptions as it does to RDF(S) schema expressed in RDF. However, their main goal is to track changes, not manipulate them and their proposal doesn't claim offer a full version control mechanism. Meta-data is associated with specific versions of an ontology allowing users to identify appropriate versions and reason about the compatibility of newer and older versions. Their papers contain some relevant work on automatically comparing ontologies to find differences.

When semantic web data is viewed as a knowledge store from which new knowledge will be deduced, a major concern is maintaining the consistency of the knowledge store. This problem has a lot in common with the version control problem in that consistency must be maintained in the face of addition and deletion of assertions. Deletion especially presents the problem of finding what deduced knowledge should be removed to keep the knowledge store consistent. Broekstra and Kampman's work on *truth maintenance* Broekstra and Kampman (2003) is concerned with tracking the deductive dependencies between statements in the RDF graph to avoid having to re-run deductive processes when statements are added or deleted from the graph.

The source or provenance of data is also a major concern when reasoning about knowledge drawn from diverse locations around the Semantic Web. A number of projects are investigating the representation of the provenance of statements within the RDF model enabling reasoning about both a fragment of data and its source. While the RDF reification mechanism provides one way of making statements about statements, as pointed out by Watkins and Nicole Watkins and Nicole (2006) reified statements cannot be used in semantic inferences. They propose the use of the Named Graph mechanism as implemented in Jena Bizer et al. (2005) to record provenance information about statements in the graph. They have used this mechanism to implement a software version control system Watkins and Nicole (2005) which uses RDF to describe changes to text based source code. An interesting result of this work is the ability to reason about the version meta-data using RDF tools to, for example, find instances where a developer reverts a change made by another developer. The use of RDF in this role adds a useful additional capability to the normal version control model.

Some recent work has directly directly addressed the problem of version management for RDF knowledge bases. The recently released IBM BOCA System[2] is an RDF repository that supports rollback of transactions in the RDF store. Very little information is available at this time about how this is implemented or the capabilities of the model used. Auer and Herre Auer and Herre (2006) present a model based on *atomic changes* to RDF triple stores which enables a kind of transaction based version management enabling changes to be rolled back if necessary. The main focus of the paper is on the application of change management to the evolution of ontologies and the authors discuss some ideas for *evolution patterns* which enable them to characterise sets of changes to an ontology as, for example, adding a new class or changing the cardinality of a property. The intention of these patterns is to provide more a useful change history to a human author than if the raw RDF changes were shown. The goals of this work are very close to that of our own project and there are many

[1] http://www.bitmover.com/

[2] http://ibm-slrp.sourceforge.net/v1/wiki/index.php/BocaUsersGuide

overlaps in the solutions we put forward. This will be discussed further at the end of this paper.

2.2 Darcs and the Theory of Patches

One issue with common version control systems such as CVS and Subversion is that they are not based on any formal model of how changes may or may not be combined with each other. This means that they are unable to convincingly assert that the result of merging a change into a working version maintains any kind of consistency constraint. One system which at least attempts to address this issue is the recently developed Darcs Roundy (2006) system.

Darcs is a distributed version control system built to manage software source code; Darcs views a version as a sequence of patches which when applied to an empty source tree, builds a particular version of the code. Also unlike other version control systems, the design of Darcs is built on a *theory of patches* – a semi-formal model of the way that individual *patches* can be manipulated to implement the required version control operations. This theory of patches is independent of the nature of a patch or change and so we can make use of it in thinking about how changes to an RDF store might be managed.

In Darcs, a *patch* is a set of changes made to a document (or collection of documents) which has been recorded by a user. A patch will tend to implement one meaningful modification to program source code, such as adding a new function or fixing a bug. The patch may include changes to multiple documents and to different parts of individual documents. A version of the document collection can be described by the sequence of patches that led up to that version; this is called the *context* of the version. A patch is said to move the working copy from one context to another.

A patch is written as an uppercase letter, optionally annotated with the left and right contexts. The sequence of patches A followed by B can be written:

$$^{o}A^{a}B^{b}$$

or more simply:

$$AB$$

the superscripts in the first notation denote named contexts: $^{o}A^{a}$ can be read as "patch A modifies the repository from context o to context a".

A Darcs patch must have the property of being invertible; that is, it must be possible to construct a second patch that will reverse the effect of a patch on the current context to get back to the previous context. The inverse of a patch is defined as the *simplest* patch that achieves this. The inverse is written as A^{-1} and by definition we know that the application of a patch followed by its inverse will bring us back to the original context:

$$^{o}A^{a}A^{-1\,o}$$

2.2.1 The Commute Operation

A fundamental operation when manipulating patches is to be able to reverse the order of two patches. As will be shown later, this forms the basis of the merge operation between two repositories and the undo operation for a patch which isn't the most recent.

Darcs defines the commute operation as the reordering of two patches. This can be written as:

$$^{o}A^{a}B^{b} \leftrightarrow {}^{o}B_1^{x}A_1^{b}$$

The commute operation is symmetrical in that both sides of the equation *have the same effect*. Note however that the individual patches on the right hand side have been modified. The new patch B_1 is defined as having the same effect as B but is modified to allow it to apply to the context o; similarly, A_1 has the same effect as A but is modified so as to apply to the context x.

In Darcs, which deals with changes to text files, the modifications needed can be quite complex since adding and deleting lines can change where subsequent changes should be applied. However, in RDF, as we will see later, no changes are ever needed when commuting patches.

In some cases the commute operation may not be defined. For example, if one patch modifies a line within a region added by the earlier patch. In this case there is said to be a *dependency* or *conflict* between the patches which prevents them from being re-ordered. The effect of a dependency will be seen later.

2.2.2 The Revert Operation

To revert the most recent patch from the current context all that is required is to remove the patch from the context (the head of the sequence). To update the working copy of the document set, the inverse of the patch can be applied to undo the changes. So, to revert the patch C from the sequence ABC, the patch C^{-1} is applied giving AB.

The ability to change the order of a sequence of patches means that it is possible to revert patches which aren't at the head of the sequence. To do so, the patch is moved to the head by a series of commute operations before the inverse patch is applied. For example, to revert A from the sequence ABC the sequence of operations is:

ABC	
B_1A_1C	Commute A and B
$B_1C_1A_2$	Commute A_1 and C
$B_1C_1A_2A_2^{-1}$	Apply the inverse patch of A_2
B_1C_1	The resulting patch sequence

This process will fail if any commute operation fails; that is if the patch being reverted has a dependency relationship with any of the subsequent patches. The process of reverting the patch is able to identify which subsequent patches are dependent and hence would have to be reverted along with the target patch to maintain consistency.

2.2.3 The Merge Operation

Two patches which apply to the same context are said to be *parallel* patches. This situation can arise when two people work from a common base each making changes to the documents. As long as the changes do not conflict with each other, it should be possible to combine them into a sequence which reflects both changes. This is the *merge* operation and it can be derived in the following manner from the inverse and commute operations.

A	Choose patch A to apply first.
AA^{-1}	Apply the inverse of A.
$AA^{-1}B$	Apply B since we are back at the initial context.
$AB_1A_1^{-1}$	Commute A^{-1} and B to put the inverse patch at the end
AB_1	Drop the final patch to give the desired sequence of patches

It should be clear that the outcome of merging A and B could be either of AB_1 or BA_1 depending on which patch was applied first and that the result of both of these should be the same. The merge operation will succeed as long as the commute operation between A^{-1} and B succeeds, that is as long as there is no dependency between these two. The resulting patch B_1 above is a modified version of B which can be applied to the context including the changes introduced by A. This modification is the same as that required for the commute operation described above.

3 VERSION CONTROL FOR RDF

In the context of an RDF triple store, there are a number of ways in which to conceive of a version control system. A primary choice is the granularity of changes that will be recorded in patches. Following Kiryakov and Ognyanov (2002) we settle on the RDF statement (triple) as the smallest directly manageable element. This implies that it is not possible to add just a resource name to a store if it doesn't take part in some RDF statement. Kiryakov and Ognyanov also state that *"An RDF statement cannot be changed – it can only be added or removed"*. This restriction follows from the observation that changing a statement $< a,b,c >$ to $< a,x,c >$ is entirely equivalent to removing the first triple and adding the second.

While the RDF statement is the smallest unit of change in the triple store, for the most part a recorded patch will consist of many additions and deletions from the store. A patch should be a *meaningful* change to the store such as the addition of a person or event; however, this is not an enforceable restriction as users of the version control system must decide what makes sense as a meaningful change. Hence, in general, a change will involve the addition or removal of a *sub-graph* from the triple store. We note here that recent work by Auer and Herre (2006) includes a very useful discussion of the role of blank nodes in recording, and more importantly in communicating, changes to a triple store. They define an *atomic graph* being one which can't be split without duplicating blank nodes and use that as the smallest unit of change in a triple store. This project has not considered issues related to blank nodes because our application doesn't make use of them, but the work of Auer and Herre is entirely compatible with the ideas expressed here as will be discussed at the end of this paper.

3.1 RDF Patches

This section defines the properties of patches used in our version control system, shows how patches are inverted, how conflicts are detected between patches and how alternate patches are computed.

A patch to an RDF store consists of a set of additions and deletions of sub-graphs. Unlike with changes to text files, no context or line number is required for each addition or deletion. The main concern when manipulating patches is to be able to invert any patch and to work out whether two patches conflict with each other.

Given these requirements, the basic form of a patch should be two sub-graphs, one to be added and the other to be deleted from the store. An example is shown in Figure 1.

While this is the basic form of a patch it may be desirable to add additional information to make identifying conflicts easier in some cases. For most uses of the version control system, modifications will be

```
add:
 :john  foaf:knows :bob .
delete:
 :john  foaf:knows :mary .
```

Figure 1: A simple RDF patch.

made by human authors; however, it may also be useful to include inferred knowledge. If this is to be done, the additions (assuming that inference does not delete knowledge) will be based on some of the existing contents of the store. These should be recorded as support for the new triples because their removal would cause a conflict with the new patch. Hence the representation of a patch is extended to include an optional dependency sub-graph taken from the working store. These dependancies must be explicitly asserted, we do not propose any mechanism for discovering dependancies from the knowledge base or any inference process. Figure 2 shows an example of such a patch.

```
add:
 :john foaf:knows :bob .
depends:
 :john foaf:works :acmecorp .
 :bob  foaf:works :acmecorp .
```

Figure 2: An example RDF patch including a dependency sub-graph.

Patch inversion can be achieved by swapping the add and delete sub-graphs in a patch fulfilling the requirement that every patch should have an inverse. This corresponds to the intuition that removing a change can be achieved by deleting anything added and adding anything deleted by the change. An interesting question is whether a dependency on the patch should remain after inversion. In the above example we infer that John knows Bob based on them both working for AcmeCorp; meaning that we should not apply this patch if the triple store does not contain the two dependent triples and that removing either dependent triple would conflict with this patch. If we were to invert this patch, so that John doesn't know Bob, we can do it irrespective of any dependency and still maintain consistency. Hence *the inversion of a patch discards any dependency sub-graph in the original*.

The Darcs commute operation is fundamental to being able to undo the effect of a historical patch and to merging changes from another repository. This operation requires that there is a test for conflict between two patches and that the alternate patches, to be applied in a different context, must be computed. These two problems are considered here.

A conflict occurs between two RDF patches if applying them in a different order would generate a possibly inconsistent state in the triple store. The easiest example is when patch A depends on a statement added by patch B: here A and B can't be re-ordered because A's dependencies won't be met in the context before applying B and so A conflicts with B. When two patches contain only add and delete sub-graphs, a conflict occurs when patch A adds a statement that is deleted by patch B or vice-versa. In this case, reordering the patches would result in deleting a non-existing statement and then adding that statement in the second patch. Hence the state of the triple store would be different if the patches were applied in a different order.

Note that it would be possible to resolve this last kind of conflict automatically. Consider the example of the patches shown in Figure 3; here the result after applying the patches in the sequence AB is that John knows James and Mary but not Bob. One way to construct a valid sequence $B'A'$ is to remove any mention of Bob from each patch. The effect of $B'A'$ would be the same as AB and consistency would be assured. The only issue with this approach is that we are potentially losing information since A and B may have come from different users who should probably be made aware of the conflict so that they can negotiate whether or not John knows James.

```
A { add:
  :john foaf:knows :bob .
  :john foaf:knows :james . }
B { add:
  :john foaf:knows :mary .
 delete:
  :john foaf:knows :bob . }
```

Figure 3: Two patches A and B which conflict due to adding and deleting a shared statement.

This brings us to the question of how to construct the alternate patch that is required when two patches are to be reversed. Recall that when a patch sequence AB is to be reversed, the patch B must be altered so that it can be applied in the context prior to A and A must be altered so that it can be applied after the modified B. In text files, these changes involve changes to line numbers in the patch. In our case, *no change is needed to patches to allow re-ordering if there is no conflict between them.*

4 IMPLEMENTATION AND EVALUATION

Our implementation goals for this project are ultimately to provide both server and desktop based triple stores and the ability to synchronise changes between different workspaces. In the first instance we have built an implementation based on the Redland Beckett (2001) system which provides a wrapper around the existing Redland Python interface to implement version control operations. This section discusses some aspects of the implementation and then provides an evaluation of the overhead cost of maintaining version control information in this way.

The first implementation decision relates to how patches will be computed from the current workspace. In text based version control the universal solution is to compute a *diff* between the current and previous versions showing which lines have been added and removed. It would be possible to use this approach for RDF and indeed we can refer to work published by Berners-Lee and Connolly (2004) on how these can be computed and stored. However, an alternative presents itself in our context which is to monitor all changes to the RDF store via the Redland interface and build the patch from the observed changes. This should be considerably more efficient when the number of statements stored becomes large.

Our approach then is to instrument the Redland Python interface to provide a wrapper around the operations that perform additions and deletions from the store to also record these changes in a patch. The wrapper also provides the version control methods `record`, to finalise a patch, `revert` to undo the effects of a patch and `merge` to merge a patch with the current working store. These will be described in more detail below.

4.1 Storing Patches

Since we have settled on a logging approach to generating patches, one possible approach is to use a tabular structure to log the add, delete and dependency triples as they are asserted in the working store. Kiryakov and Ognyanov (2002) suggest that a relational database is an appropriate store for this data as this can be stored more compactly than if it were asserted into the RDF store. A patch then would be a table of add/delete operations with their parameters (subject, predicate, object).

An alternative to this approach is to use RDF to represent the patches with each add and delete operation forming a node in a special patch store. This aligns with the work outlined by others Berners-Lee and Connolly (2004); Auer and Herre (2006) who define ontologies for RDF patches and allows for recording metadata and reasoning about the patches themselves, perhaps along the lines of the inferences suggested by Watkins and Nicole (2006). Since we need to make statements about the addition and deletion of statements some kind of reification is required. Following the earlier work, we can also include statements about the provenance of the changes and about the relationships between patches.

Our implementation makes use of the Redland Beckett (2001) *context* mechanism, which is similar to the named graph model used by Watkins and Nicole. A context in Redland can be associated with one or more assertions into the triple store. These are used to group together the changes in a given patch and since a context is identified by a resource node in the graph, can also be used to make statements about the patches themselves. Our implementation uses two separate RDF stores for the working store and the patch store although it would be only a little more complex to use one store containing many contexts for both of these. The main working store is maintained as it would normally be except that the add and delete operations are intercepted by our wrapper. The patch store contains just the assertions about patches, with each patch being in its own context. One further item of metadata must be stored in the working store to record the sequence of patches that define the current state. This information is needed when new patches are made and when patches are reverted and merged from other working stores.

Each addition or deletion from the working store is asserted as a reified statement within the current context (shown here using the TriG named graph syntax Bizer (2005)):

```
:patchA {
  :add1 rdf:type darcs:assert,
        rdf:subject :john,
        rdf:predicate foaf:knows,
        rdf:object :bob .

  :del1 rdf:type darcs:delete,
        rdf:subject :john,
        rdf:predicate foaf:knows,
        rdf:object :jane .
}
```

Here `:patchA` is a named graph containing the reified assertions. These assertions are collected in a *current patch* context until such time as the user asks to record or commit the patch. At this point the patch is closed and assertions can be made about it via the context node (in Redland, the context identifier is a first class node in the graph). For example, the au-

thor and time of the patch and any commentary can be recorded.

```
:patchA rdf:type darcs:patch,
     dc:creator <mailto:fred@bloggs.com>,
     dc:date    "2005-10-20",
     darcs:comment "Added new name" .
```

One step remains before the patch can be finalised and this relates to the possibility that the same statement might be added and deleted in the patch. If there is a pair of add and delete nodes referring to the same statement then they should be removed from the patch since the net effect will be nil on the final difference between the current and prior version. Normalisation produces a non-redundant set of changes required to create the effect of a patch. This allows conflicts to be detected without checking for redundant add/delete pairs within the patches themselves. After normalisation, the patch context can be closed and a new one opened for any future changes to be recorded.

4.2 The Revert Operation

The revert operation undoes the effect of a patch on the working store, any additions are removed and any deletions are re-added. As was shown above (Section 2.2.2), the revert operation on the most recent patch is achieved by applying the inverse patch to the working store. To revert a historical patch, the original patch must be commuted to the head of the patch history and then the inverse patch applied. Since commuting an RDF patch is a simple re-ordering the only issue in this operation is whether the patch being reverted conflicts with a patch more recent in the history. For example, if we wish to revert patch A from the sequence $ABCD$ and A asserts that *John knows Jane* while C removes that statement, then the revert cannot succeed and the user must be informed of the conflict. The user may decide to revert the patch C before reverting A to avoid this problem.

4.3 The Merge Operation

The merge operation applies changes from one working store to another. The situation arises when a branch workspace is created, for example if Steve takes a copy of James' workspace to add some material on a special topic. At the point copying the workspace, the patch history might be $ABCD$, Steve adds a number of statements and records a patch S, meanwhile James has also made some more changes and recorded a patch J. Now Steve sends James the patch S and James must merge it with his workspace $ABCDJ$. As shown above (Section 2.2.3), the merge operation requires commuting S and J to give the final sequence $ABCDJS'$ where S' is the modified version of S that applies after J. For our RDF patches $S = S'$ and so the merge can be performed simply by applying the patch S as long as there is no conflict between S and J.

4.4 Evaluation

A brief evaluation of our implementation was carried out to ensure that the overhead introduced by version control was acceptable in terms of additional space and processing time. The results showed that using the MySQL backend for the Redland store, VC is around four to eight times slower and needs from two to four times as much space as the raw RDF store. While this overhead is significant, it is still workable in the kinds of applications we envisage. There is room for optimisation here by working at a lower level in the implementation of the RDF store.

5 DISCUSSION

As RDF is used more widely it will require the same kind of tools as mainstream programming languages since developers will want to collaborate on creating and maintaining an RDF store in the same way that they currently do with source code. This paper has presented a proposal for a version control system for RDF based on a semi-formal *theory of patches* as implemented in the Darcs version control system. This provides the basic building blocks to build a distributed version control system for RDF which can support collaborative editing of RDF stores. The model is being implemented as part of an eResearch project developing tools for collaborative annotation of Linguistic data.

Auer and Herre (2006) recently described their work on version control for RDF which has many similarities with the work described here. Their focus was somewhat different to our own in that they are concerned with ontology evolution and spend time deriving evolutionary patterns which can be used to classify the changes made to the RDF store. However, they do cover a number of points that aren't considered in our work.

Auer and Herre define an *atomic graph* as the smallest sub-graph that can't be sub-divided without duplicating blank nodes; these are the smallest units of change in their system. A *change* (their name for our patch) consists one or more of these atomic graphs added and/or deleted from the triple store. Apart from the consideration of blank nodes,

this formulation is entirely compatible with our notion of RDF patches. Our system could be modified to work with atomic graphs in more general RDF vocabularies where blank nodes are an issue. The notion of *conflict* between two changes is introduced as a barrier to rolling back a change that occured some time in the past. However, this notion is only described as a relation between a change (patch) and a working store. The idea of a conflict relation between patches is not introduced and hence the manipulations that the patch theory allows are not explored in their system.

It is clear then that this work is complimentary to our own and that by borrowing the idea of the atomic graph we should be able to expand our system to deal with a broader range of RDF applications.

Our work is currently focused on the application of our VC managed triple store to the problem of Linguistic Annotation. Once we have successfully demonstrated the technology in this application we will look further into the more general problem of VC for RDF and evaluate it in the context of ontology evolution and other RDF authoring tasks.

REFERENCES

Auer, S. and Herre, H. (2006). A versioning and evolution framework for rdf knowledge bases. In *Proceedings of Ershov Memorial Conference*, Novosibirsk, Akademgorodok, Russia.

Beckett, D. (2001). The Design and Implementation of the Redland RDF Application Framework. In *Proceedings of WWW10*, Hong Kong.

Berners-Lee, T. and Connolly, D. (2004). Delta: an ontology for the distribution of differences between rdf graphs. World Wide Web. `http://www.w3.org/DesignIssues/Diff`.

Bird, S. and Liberman, M. (2000). A Formal Framework for Linguistics Annotation. *Speech Communication*.

Bizer, C. (2005). The TriG Syntax. `http://www.wiwiss.fu-berlin.de/suhl/bizer/TriG/`.

Bizer, C., Cyganiak, R., and Watkins, R. (2005). Named Graphs for Jena (NG4J) API. In *Proceedings of the Second European Semantic Web Conference.*, Greece.

Broekstra, J. and Kampman, A. (2003). Inferencing and Truth Maintenance in RDF Schema: exploring a naive practical approach. In *Workshop on Practical and Scalable Semantic Systems (PSSS)*.

Cassidy, S. and Harrington, J. (2000). Multi-level Annotation in the Emu Speech Database Management System. *Speech Communication*, 33:61–77.

Cederqvist, P. (2003). *Version Management with CVS*. Network Theory. `http://www.network-theory.co.uk/docs/cvsmanual/`.

Collins-Sussman, B., Fitzpatrick, B. W., and Pilato, C. M. (2004). *Version Control with Subversion*. O'Reilly.

Kiryakov, A. and Ognyanov, D. (2002). Tracking Changes in RDF(S) Repositories. In *EKAW '02: Proceedings of the 13th International Conference on Knowledge Engineering and Knowledge Management. Ontologies and the Semantic Web*, pages 373–378, London, UK. Springer-Verlag.

Klein, M., Fensel, D., Kiryakov, A., and Ognyanov, D. (2002). Ontology versioning and change detection on the web. In *EKAW '02: Proceedings of the 13th International Conference on Knowledge Engineering and Knowledge Management. Ontologies and the Semantic Web*, pages 197–212, London, UK. Springer-Verlag.

Lord, T. (2006). How arch works. Web page. `http://regexps.srparish.net/www/arch-tech.htm` accesed on 13 December 2006.

R. Schroeter, J. Hunter, D. K. (2003). Vannotea - a collaborative video indexing, annotation and discussion system for broadband networks. In *K-CAP 2003 Workshop on Knowledge Markup and Semantic Annotation*, Florida.

Roundy, D. (2006). The darcs revision control system. `http://abridgegame.org/darcs/`. Accessed on 2006-05-18.

Watkins, E. R. and Nicole, D. A. (2005). Version control in online software repositories. In *Proceedings of the 2005 International Conference on Software Engineering Research and Practice II*, pages 550–556.

Watkins, E. R. and Nicole, D. A. (2006). Named graphs as a mechanism for reasoning about provenance. In *Lecture Notes in Computer Science Frontiers of WWW Research and Development - APWeb 2006: 8th Asia-Pacific Web Conference*, pages 943–948, Harbin, China.