Dynamic Ontology Version Control

Dan Schrimpsher
Computer Science Department
University of Alabama in Huntsville
djs0003@uah.edu

Zhiqiang Wu
Computer Science Department
University of Alabama in Huntsville
wujohn101@gmail.com

Anthony M. Orme
Computer Science Department
University of Alabama in Huntsville
aorme@cs.uah.edu

Letha Etzkorn
Computer Science Department
University of Alabama in Huntsville
letzkorn@cs.uah.edu

ABSTRACT
Ontologies are used today in many application areas. With the use of ontologies in bioinformatics, as well as their use in semantic web technologies, ontology based software has become widely used. This has led to a need for keeping track of different ontology versions [8], as the operation of software will change as the ontologies it uses change. However, existing approaches to ontology versioning have worked on static ontologies. Thus, the ontology version that a software package will use must be chosen prior to running that package. This requires substantial human oversight, and is therefore a major limitation. In this paper, we examine a dynamic approach to ontology versioning that will automatically provide the correct ontology for a software package on-the-fly. We examine a methodology that employs storing different time stamped ontologies in the same file, and we discuss how this methodology can be used on a real ontology.

Categories and Subject Descriptors
D.2.7 **[Distribution, Maintenance, and Enhancement]**: *Version control*

D.2.8 **[Metrics]**: *Complexity Measures and Performance Measures.*

H.3.5 **[Online Information Services]**: *Web-based Services.*

*I.7.1 [***Document and Text Editing***]: Version Control*

General Terms
Algorithms, Management, Measurement, Performance, Verification.

Keywords
Web Services, Ontologies, Version Control, Dynamic Configuration Control

1. INTRODUCTION
In recent years, due to the competitive advantage of Web technologies, the Web has been the most important platform to carry out business-to-business (B2B) processes. More and more organizations around the world have been migrating their business processes to the Web. Among web technologies, Web services are becoming a standard method of sharing data and functionality among loosely-coupled systems. Among the services, ontologies are supported by Web Services to improve their client interfaces . Due to the rapid change of science and industry, the content of ontology is subject to change as well. The accuracy of ontology information is critical as clients can encounter problems in cases where earlier accesses of the web service employed an "old image" of the ontology doesn't exist any more or because the web service cannot find the right version we requested.

The development of bioinformatics also led to the create of numerous ontologies with multiple versions, and the situation is very much the same as with web services; in fact, in many cases web services are used for bioinformatics[2].

Each ontology has multiple versions and a tremendous number of elements that may change frequently, either added or deleted. Therefore, identifying the right version is very critical and important. When users want to use an ontology, they not only need to identify the right version, but also need to deal with scenarios as follows: (1) identify the right version of the ontology;(2) do a comparison between a specific version and the one right before or after it to draw some conclusion;(3) get the history of a specific element; (4) for checking the effect of adding some factor, they need to find the history of some certain elements in certain period.

In this paper, we examine the use of time stamping ontology elements within a file to achieve these goals. We are particularly interesting in having a particular version of an ontology requested and received by a client at runtime. Version control has been used in the past for static ontologies [1,5,6,7, 9], but never before, to our knowledge, for on-the-fly ontology versioning. However, our approach leads to challenges such as: (1) Given a timestamp, how can we get the right version of a specific ontology efficiently? (2) Given a timestamp, can we quickly get the results of comparisons between two versions of an ontology, one comparison right before the given timestamp and another right after the timestamp? (3) Can we download multiple versions of an ontology at one time?

For the challenges addressed above, the critical issues can be concluded as two-fold: 1) quickly identifying the timestamp of each version, and 2) fetching all versions of interest at one time. To the best of our knowledge, there is no previously existing system which can meet our need specified above.

ACMSE '10, April 15-17, 2010, Oxford, MS, USA.

Another consideration is how can we efficiently store and update ontology versions on the fly. Unlike source code, where it makes sense store versions at the file level, ontologies need version of concepts. If we store off the entire ontology every time a concept is updated, it could quickly grow unmanageable.

This paper presents and discusses three methods of storing ontology data and timestamp is used to identify the version. Calculation for storage space for each method is also done.

The remainder of this paper is organized as follows: Section 2 provides the necessary background on OWL needed to understand web-based ontologies.

2. BACKGROUND

Tom Gruber has defined of an ontology as:

"When the knowledge of a domain is represented in a declarative formalism, the set of objects that can be represented is called the universe of discourse. This set of objects, and the describable relationships among them, are reflected in the representational vocabulary with which a knowledge-base program represents knowledge. Thus, in the context of AI, we can describe the ontology of a program by defining a set of representational terms. In such ontologies, definitions associate names of entities in the universe of discourse (for example, classes, relations, functions, or other objects) with human-readable text describing what the names mean, and formal axioms that constrain the interpretation and well-formed use of these terms. Formally, an ontology is the statement of a logical theory. [4]"

The Resource Description Framework, or RDF, is World Wide Web (WWW) defined method of modeling information, through a variety of syntax formats. The Web Ontology Language (OWL) is a semantic markup language for describing an ontology on the World Wide Web that extends RDF.

OWL describes concepts and their relationships through the use of classes and relations between classes. OWL contains formal semantics that allow reasoning about concepts to a limited degree.

While our discussions of methods are not limited to OWL defined ontologies, the ontologies we used as samples for testing our methods are OWL ontologies.

2.1 Ontology Versioning

In 1997, Swartout et al. recognized the need for version control in collaborative development of ontologies. They proposed an environment where modules could be checked in an out, tracking differences over time. Their focus was on the development of ontologies, rather than clients using multiple versions of the ontology dynamically [8].

In 2000, Kozaki et al. developed a ontology development tool, HOZO. This tool mapped ontologies and models to a particular author and others were only allowed read access. They claimed this allowed collaboration without explicit version control [3]. However, it limited collaboration on large ontologies by forcing a single user to work on modules.

Sunagawa et al. looked at version control between modules of a large distributed ontology. They were focused on depencendy between modules and how it was effected by changed in other modules. Version history of a module was only needed when other modules were dependent on it [7]. Again, this was part of an environment for developing application ontologies based on many modules, possibly of different versions. Client's needs were not addressed.

Khatri and Draghici looked at large gene expression ontologies and the problem of how to guarantee researchers were looking at the latest version [2]. This was mainly focused on the logistics of either pushing updates to a web server or pulling updates down to a standalone computer. This issue of how to store the versions or get previous versions was not addressed.

Kalyanpur et al. created an online ontology editor called Swoop. This editor allowed versioning of an ontology much as source code is versioned. The author could store multiple versions of the ontology locally, and choose which was available on the web [1]. While this was a step forward of actually developing a tool to handle versioning, it was limited to developers and clients were always forced to use the version pushed to the web.

Noy et al. examined multiple maintenance scenarios of ontologies and realized that some users would need to access to different versions of the code. They developed a solution using the Change and Annotation Ontology (ChAO) which is a Protégé plug-in that annotates changes made to an ontology. This allows a user to accept or reject changes made to an ontology they are working with [5]. While this is a major step forward in dynamic ontology version control, it doesn't easily support software agents working asynchronously on an ontology that is changing. Also it is not clear how much storage is needed to for ChAO information relative to the ontology.

Volkel and Groza created an RDF based versioning system called SemVersion. This allowed semantic and structural differences between versions and separated the storage of the versions from the management tool used to store them. They acknowledge that the reasoning for semantic differences may be difficult to scale to larger and larger ontologies [9].

Redmond et al. created a version management system specifically for OWL that allows concurrent, multiuser editing. This is the first study that focused on the performance of the versioning software. They looked at a heuristic approach which tried to find a near optimal number of backups to store [6]. Again, this tool is focused on supporting concurrent development on ontologies and doesn't approach the problem from the user's point of view.

3. METHODS

Ontologies, unlike source code, cannot be treated as a single file in configuration control. As we have seen, there have been new methods to implement configuration control for ontologies proposed. Research to date, however, has been focused on the developer's side. Clients who are using an ontololgy in a highly dynamic environment may wish to continue using a previously processed ontology rather than converting to a new version.

This paper will explore three methods for dynamic version control of ontologies with the goal of minimizing the impact on the user. The three methods are Whole Stamping, Range Stamping, and Implicit Stamping.

There are three considerations in comparing methods. The structure of the stamps shows how the stamp information is stored

and how it changes as attributes and classes are changed. Parent updates shows what happens to a child class or attribute when a parent is changed. Finally, deleted attributes shows how deletion is shown in the scheme.

3.1 Whole Stamping

'Whole stamping' is our term for the brute force method of dynamic ontology configuration control.. In this technique, every element (attribute, class, or instance) in the ontology is "stamped" with a list of version numbers that it is valid for.

The structure of this method is simple. Every element has a list of versions that it supports. When an element is changed a new copy is created with a new list and the old version stays with the original list. In figure 1, an element A is updated to AA. A stays in the list with its original version record and AA is added with the new version, 4. Also B is updated to be included in version 4.

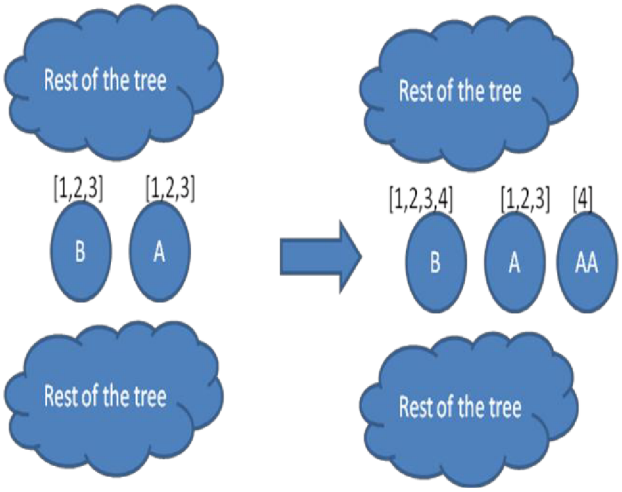

Figure 1 Example of Updating Element A with Whole Stamping

When any element is changed, all other elements in the ontology must have the new version appended to their version record to continue being valid. This is an explicit method of version control.

Deletions are shown implicitly in this method, as a deleted node simply stops updating its list for new versions. Therefore it doesn't show up in versions selected after the deletion occurred.

While this is a fairly simple method to implement, it has some space considerations that need to be discussed. We assume an ontology consist of n elements each size 1. We also assume there are m entries in each version record each of size 1. Then total size of the ontology is n*m.

As long as you either have a small ontololgy (small n) or a stable ontology (small m) the footprint of this method is acceptable. However, for large ontologies, the size complexity depends on how many updates are made to an ontology over time. As a general rule:

- m << n, size complexity is O(n)
- m ~= n size complexity is $O(n^2)$
- m >> n size complexity is O(m) (which is > $O(n^2)$)

The time to update the version record is n, since each valid element in the ontology has to be updated when a new version is created. So the update time is O(n).

To sum up this method, the ontology size is doubled with every change. This is a simple scheme but it is a waste of space if you change the ontology a lot. This technique is only useful for very stable ontologies, which very likely don't need a dynamic versioning system.

3.2 Range Stamping

In this technique we use ranges to show what time stamps a given attribute is valid for. Instead of a list, we only keep the first and last version a node is valid for. Each element then has at most two version record entries.

When an element is changed/added/deleted, all other elements in the ontology must update the max version to continue being valid. This is an explicit and compressed method of version control.

To continue our example in figure 2, a given element A is updated to AA.

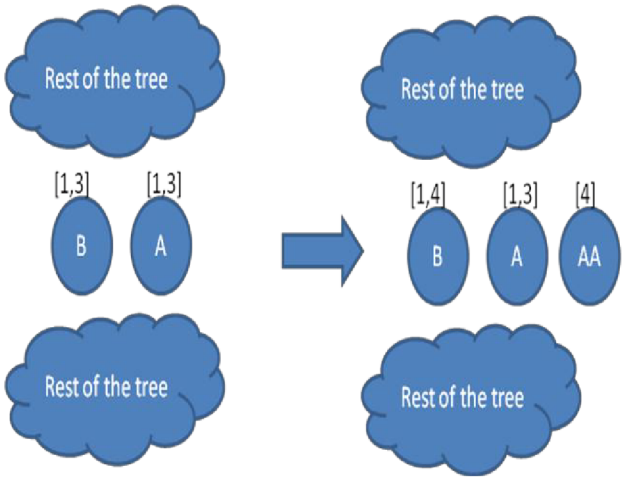

Figure 2 Example of Updating Element A with Range Stamping

Deletions are shown implicitly in this method, as a deleted node simply stops updating its range for new versions. Therefore it doesn't show up in versions selected after the deletion occurred.

This method is somewhat more complex than that of whole stamping, but it results in significant size savings. However, since all valid elements in the ontology still have to be updated during a version change, there is little savings on the update processing time over whole stamping.

As before, if we assume an ontology consist of n elements each size 1. We also know there are at most 2 entries in the version record each of size 1. Then total size of the ontology is n*2. This gives a space order of O(n), much improved over whole stamping.

The time to update the version record is also n, since each valid element in the ontology still has to be updated when a new version is created. So the update time is also O(n).

To sum up this method, while a little more complicated than whole stamping, the size of the version record doesn't significantly increase over time. Thus, much less space is wasted in this method than when using whole stamping.

3.3 Implicit Stamping

In this technique, timestamps are implicit and the use of states is used to limit the validity of the concept. Instead of a list, we only keep the first version a concept is valid for. When a element becomes invalid, due to a deletion or update, a state value of deleted is set.

When an element is changed/added/deleted, only new or deleted elements in the ontology must update the new version. This is an implicit method of version control.

To continue our example in figure 3, a given element A is updated to AA.

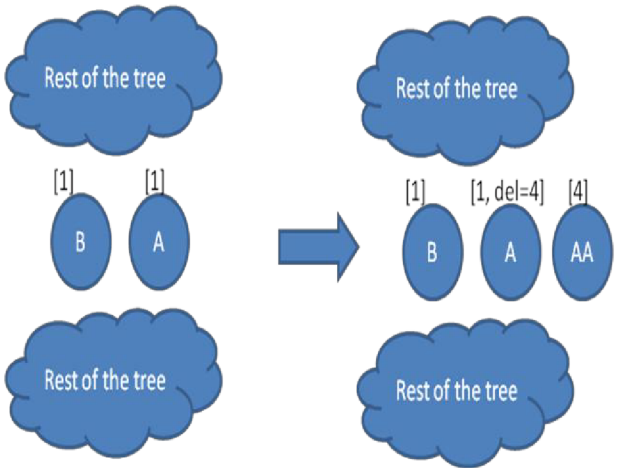

Figure 3 Example of Updating Element A with Implicit Stamping

Deletions are explicit and are done through a state attached to each version record. The state is set to *del* for the timestamp where it is removed and thus it is left out of the ontology for that version or later.

This method is perhaps the most complex of the methods we are examining in this paper, but it results in significant size savings and the time it takes to update a version is constant rather than a function of the size of the tree.

As before, if we assume an ontology consist of n elements each size 1, then we also know there are at usually 1 entries in the version record each of size 1. There are two entries when an element is deleted. Then the total size of the ontology is c*n, where 1<c<2. This gives a space order of O(n), still much improved over whole stamping.

The time to update the version record is equal to the number of elements affected. Assuming this is small relative to the ontology size, it is O(1).

To sum up this method, while a little more complicated in retrieving a version of an ontology, it doesn't change its size significantly over time and it provides constant version updates. This would be good for a large ontology that changes often.

4. RESEARCH METHOD

The goal of this study is to use minimum information to record the change happened on a specific ontology on certain period, while still allowing clients to access multiple versions of the ontology with minimum impact.

In this study, all three methods were applied to a subset of a travel agency ontology available on the internet at http://www.icsd.aegean.gr/kotis/ontologies/travelAgency.owl. A subset of the ontology was created as the base version. Then five changes, listed in table 1, were applied. For our purposes, elements include classes, properties, or instances in the ontology.

Table 1 Update to the Ontology

Version	Step	Number of Elements
0	Initial State	11
1	Add Activities sub-tree	15
2	Add Activities property to Cruise	16
3	Add Bus as a subclass of Transportation	17
4	Delete Rock Climbing class	17
5	Rename Loc to Location	18

These cover three types of changes, additions, updates, and deletions. We believe this gives a reasonable coverage of changes that may occur to an ontology.

Once the methods were applied, a number of metrics were collected. These include:

- Number of Elements (NoE)
- Size of Version Record (SVR)
- Time to Update the Ontology (TUO)
- Time to Update the Version Record (TUVR)
- Time to get Version 1 (TV1)
- Time to get Latest Version (TLV)

These metrics were then used to compare each method for storage and time efficiency.

5. RESULTS

The numbers given in this section are abstract, rather than platform specific bytes and seconds. This removes machine dependent issues and gives a more general idea of the effects of the versioning methods. Sizes are given in number of elements or number of version numbers stored. Time is given in algorithmic steps.

The first method used was whole stamping. Whole stamping is a "brute force" obvious solution to version control that we will use as a baseline to measure other techniques. The results are given in table 2.

Table 2 Metrics for Whole Stamping

Version	NoE	SVR	TUO	TUVR	TV1	TLV
0	11	11	0	0	11	11
1	15	26	4	15	12	15
2	16	42	1	16	13	16
3	17	59	1	17	14	17
4	17	75	1	16	14	17
5	18	92	1	16	15	18

The second method used was range stamping. The results are given in table 3.

Table 3 Metrics for Range Stamping

Version	NoE	SVR	TUO	TUVR	TV1	TLV
0	11	11	0	0	11	11
1	15	26	4	15	12	15
2	16	31	1	16	13	16
3	17	33	1	17	14	17
4	17	34	1	16	14	17
5	18	35	1	16	15	18

The second method used was implicit stamping. The results are given in table 4.

Table 4 Metrics for Implicit Stamping

Version	NoE	SVR	TUO	TUVR	TV1	TLV
0	11	11	0	0	11	11
1	15	15	4	4	12	15
2	16	16	1	1	13	16
3	17	17	1	1	14	17
4	17	18	1	1	14	17
5	18	20	1	2	15	18

5.1 Comparison

The first metric, NOE, is identical for each method, which is to be expected. The size of the ontology is based on the changes made, rather than the versioning method. Similarly, the metric TUO is also based on the changes rather than the versioning method, so the numbers are identical.

It should also be noted that TUO, TV1, and TLV are identical. This is a good sign, that means regardless of which method we choose, the lookup performance seen by the client will be the same.

The two interesting metrics are SoVR, which indicate the size used to store the versions, and TUVR, which indicate the steps to update the record.

5.2 Version Record Space Comparison

SoVR in whole stamping is dependent on both NOE and the number of versions. In general it can be given by:

$$SoVR_i \sim NoE_i * i$$

where i is the new version number. This means that whole stamping version record will grow very large as the number of versions increases.

In comparison, the SoVR in range stamping is only dependent on the NOE, regardless of the number of versions. Since each element can have at most 2 versions (a min and max range), it is given by:

$$SoVR_i \sim 2 * NoE_i$$

SoVR in implicit stamping is also dependent only on NOE. However, since there is only one record (except for deleted, or updated elements which have two), in general it is given by:

$$SoVR_i \sim c * NoE_i \ (where\ 1 < c < 2)$$

So implicit and range stamping are in the same order on space, but both are much smaller than whole stamping. A graphical comparison of the space requirements for version records is in figure 4.

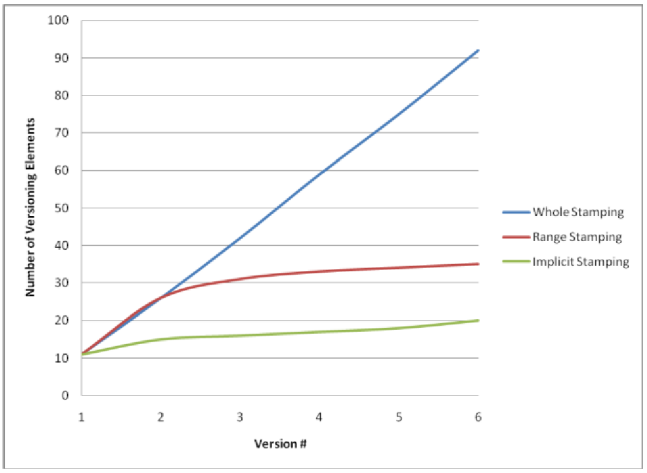

Figure 4 Space Comparison of Versioning Methods

5.3 Time to Update Version Record Comparison

In whole stamping, every time a version changes, each element in the ontology much be updated. Therefore, TUVR is dependent on NOE. In general it can be given by:

$$SoVR_i \sim NoE_i$$

Similarly, range stamping requires the max on each unchanged element's version record to be updated. So it also is dependent on NoE. In general it can be given by:

$$SoVR_i \sim NoE_i$$

This means that whole and range stamping's version record will take much longer to update a large ontology than a small one.

In implicit stamping, only the version records of elements that are added, deleted, or updated are changed. So SoVR is dependent on affected elements. In general it is given by:

$$SoVR_i \sim Changed\ Elements_i$$

So in implicit stamping, the time it takes to update the version record is determined by how many elements are changing rather than how large the ontology is. A graphical comparison of the space requirements for version records is in figure 5.

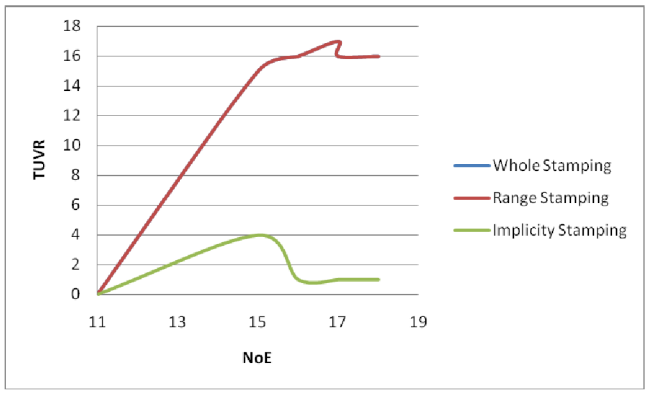

Figure 5 Time Comparison of Versioning Methods

6. CONCLUSION

Based on the results in section 5, implicit stamping is the best method for both space and time concerns. More importantly, the required space is not dependent on how many updates are made and time to update is not based on the size of the ontology. So for large, often updated ontologies it is clearly the best choice.

Implicit stamping is a more complicated scheme for versioning an ontology, so it may not always be the best choice. For small very stable ontologies, it may be a waste of time, although it space and time performance will still be marginally better than the other to methods.

This paper discussed several possible methods of storing ontology and using timestamps for version control. Ontology versions can be identified easily based on timestamp. The implicit stamping method keeps all history of the ontology. Thus implicit stamping shows great potential for use when dynamic ontologies are employed.

7. Future Research

Though we currently see time stamping methods in terms of dynamic configuration control of an ontology, future research might include creating an interface tool to apply this technique to existing ontologies, such as external concepts linked in to a time stamped ontology.

Related research ideas are merging related ontologies, creating subsets of an ontology for use with limited clients, such as mobile phones, and creating multiple subsets of an ontology for different uses based on a client's needs.

8. REFERENCES

[1] Kalyanpur, A., Parsia, B., Sirin, E., Grau, B.C., Hendler, J. 2006. Swoop: A Web Ontology Editing Browser. Journal of Web Semantics 4, 2, 144-153. DOI=10.1016/j.websem.2005.10.001.

[2] Khatri, P. and Draghici, S. 2005. Ontological Analysis of Gene Expression Data: Current Tools, limitations, and Open Problems. Bioinformatics. 21, 18, 3587-3595. DOI=10.1093/bioinformatics/bti565.

[3] Kozaki, K., Kitamura, Y., Ikeda, M., and Mizoguchi, R. 2000. Development of an Environment for Building Ontologies which is based on a Fundamental Consideration of "Relationship" and "Role." In Proceedings of The Sixth Pacific Knowledge Acquisition Workshop. Sydney, Australia.

[4] Gruber, T.R. 1993. A Translation Approach to Portable Ontology Specification. Knowledge Acquisition. 5, 2, 199-220.

[5] Noy, N., Chugh, A., Liu, W., and Musen M. 2006. A Framework for Ontology Evolution in Collaborative Environments. In Proceedings of the Semantic Web. 4273, 544-558.

[6] Redmond, T., Smith, M., Drummond, N., and Tudorache, T. 2008. Managing Change: An Ontology Version Control System. In OWL: Experiences and Directions, 5th International Workshop. Karlsruhe, Germany.

[7] Sunagawa, E., Kozaki, K., Kitamura, Y., Mizoguchi R. 2003. An Environment for Distributed Ontology Development Based on Dependency Management. In Proceedings of the 2nd International Semantic Web Conference. Florida, USA

[8] Swartout, B., Patil, R., Knight, K., and Russ, T. 1996. Toward distributed use of large-scale ontologies. In Proceedings of 10th Knowledge Acquisition for Knowledge-Based Systems Workshop. Banff, Canada.

[9] Volkel, M., and Groza, T. 2006. SemVersion: RDF-based ontology versioning system. In Proceedings of the IADIS InternationalConference WWW/Internet 2006. Murcia, Spain.