



D2.3.3.v2 SemVersion – Versioning RDF and Ontologies

Max Völkel (University of Karlsruhe)

with contributions from:

Sebastian Ryszard Kruk (DERI)

Anna V. Zhdanova (DERI)

Robert Stevens (U Manchester)

A. Artale, E. Franconi, S. Tessaris (Free University of Bolzano)

Abstract. SemVersion is a generic, extendable multi-language ontology versioning system, that can help research and industry to employ ontology based technologies in dynamic settings. This deliverable describes the requirements for and design of SemVersion.

EU-IST Network of Excellence (NoE) IST-2004-507482 KWEB
Deliverable D2.3.3.v2 (WP2.3)

Document Identifier	KWEB/2004/D2.3.3.b/v1.0
Project	KWEB EU-IST-2004-507482
Version	v1.0
Date	January 27th, 2006
State	final
Distribution	public

Knowledge Web Consortium

This document is part of a research project funded by the IST Programme of the Commission of the European Communities as project number IST-2004-507482.

University of Innsbruck (UIBK) - Coordinator

Institute of Computer Science
Technikerstrasse 13
A-6020 Innsbruck
Austria
Fax: +43(0)5125079872, Phone: +43(0)5125076485/88
Contact person: Dieter Fensel
E-mail address: dieter.fensel@uibk.ac.at

École Polytechnique Fédérale de Lausanne (EPFL)

Computer Science Department
Swiss Federal Institute of Technology
IN (Ecublens), CH-1015 Lausanne
Switzerland
Fax: +41 21 6935225, Phone: +41 21 6932738
Contact person: Boi Faltings
E-mail address: boi.faltings@epfl.ch

France Telecom (FT)

4 Rue du Clos Courtel
35512 Cesson Sévigné
France. PO Box 91226
Fax: +33 2 99124098, Phone: +33 2 99124223
Contact person : Alain Leger
E-mail address: alain.leger@rd.francetelecom.com

Freie Universität Berlin (FU Berlin)

Takustrasse 9
14195 Berlin
Germany
Fax: +49 30 83875220, Phone: +49 30 83875223
Contact person: Robert Tolksdorf
E-mail address: tolk@inf.fu-berlin.de

Free University of Bozen-Bolzano (FUB)

Piazza Domenicani 3
39100 Bolzano
Italy
Fax: +39 0471 315649, Phone: +39 0471 315642
Contact person: Enrico Franconi
E-mail address: franconi@inf.unibz.it

Institut National de Recherche en Informatique et en Automatique (INRIA)

ZIRST - 655 avenue de l'Europe -
Montbonnot Saint Martin
38334 Saint-Ismier
France
Fax: +33 4 7661 5207, Phone: +33 4 7661 5366
Contact person: Jérôme Euzenat
E-mail address: Jerome.Euzenat@inrialpes.fr

Centre for Research and Technology Hellas/ Informatics and Telematics Institute (ITI-CERTH)

1st km Thermi - Panorama road
57001 Thermi-Thessaloniki
Greece. Po Box 361
Fax: +30-2310-464164, Phone: +30-2310-464160
Contact person: Michael G. Strintzis
E-mail address: strintzi@iti.gr

Learning Lab Lower Saxony (L3S)

Expo Plaza 1
30539 Hannover
Germany
Fax: +49-511-7629779, Phone: +49-511-76219711
Contact person: Wolfgang Nejdl
E-mail address: nejdl@learninglab.de

National University of Ireland Galway (NUIG)

National University of Ireland
Science and Technology Building
University Road
Galway
Ireland
Fax: +353 91 526388, Phone: +353 87 6826940
Contact person: Christoph Bussler
E-mail address: chris.bussler@deri.ie

The Open University (OU)

Knowledge Media Institute
The Open University
Milton Keynes, MK7 6AA
United Kingdom
Fax: +44 1908 653169, Phone: +44 1908 653506
Contact person: Enrico Motta
E-mail address: e.motta@open.ac.uk

Universidad Politécnica de Madrid (UPM)

Campus de Montegancedo sn

28660 Boadilla del Monte

Spain

Fax: +34-913524819, Phone: +34-913367439

Contact person: Asunción Gómez Pérez

E-mail address: asun@fi.upm.es

University of Liverpool (UniLiv)

Chadwick Building, Peach Street

L697ZF Liverpool

United Kingdom

Fax: +44(151)7943715, Phone: +44(151)7943667

Contact person: Michael Wooldridge

E-mail address: M.J.Wooldridge@csc.liv.ac.uk

University of Sheffield (USFD)

Regent Court, 211 Portobello street

S14DP Sheffield

United Kingdom

Fax: +44 114 2221810, Phone: +44 114 2221891

Contact person: Hamish Cunningham

E-mail address: hamish@dcs.shef.ac.uk

Vrije Universiteit Amsterdam (VUA)

De Boelelaan 1081a

1081HV. Amsterdam

The Netherlands

Fax: +31842214294, Phone: +31204447731

Contact person: Frank van Harmelen

E-mail address: Frank.van.Harmelen@cs.vu.nl

University of Karlsruhe (UKARL)

Institut für Angewandte Informatik und Formale

Beschreibungsverfahren - AIFB

Universität Karlsruhe

D-76128 Karlsruhe

Germany

Fax: +49 721 6086580, Phone: +49 721 6083923

Contact person: Rudi Studer

E-mail address: studer@aifb.uni-karlsruhe.de

University of Manchester (UoM)

Room 2.32. Kilburn Building, Department of

Computer Science, University of Manchester,

Oxford Road

Manchester, M13 9PL

United Kingdom

Fax: +44 161 2756204, Phone: +44 161 2756248

Contact person: Carole Goble

E-mail address: carole@cs.man.ac.uk

University of Trento (UniTn)

Via Sommarive 14

38050 Trento

Italy

Fax: +39 0461 882093, Phone: +39 0461 881533

Contact person: Fausto Giunchiglia

E-mail address: fausto@dit.unitn.it

Vrije Universiteit Brussel (VUB)

Pleinlaan 2, Building G10

1050 Brussels

Belgium

Fax: +32 2 6293308, Phone: +32 2 6293308

Contact person: Robert Meersman

E-mail address: robert.meersman@vub.ac.be

Executive Summary

Change management for ontologies becomes a crucial aspect for any kind of ontology management environment, as engineering of ontologies often takes place in distributed settings where multiple independent users have to interact. There is also a variety of ontology languages used. Although RDF Schema and OWL are gaining more and more popularity, a lot of semantic data still resides in other formats, as it is the case in the biology domain (c.f. Sec. 2.1.3, page 8). Until now, no standard versioning system or methodology has arisen, that can provide a common way to handle versioning issues.

This deliverable describes the RDF-centric versioning approach and implementation *SemVersion*. It provides structural (purely triple based) and semantic (ontology language based, like RDFS, OWL, and OBOL) versioning. It separates language-neutral features for data management from language-specific features like semantic diffs in design and implementation. This way SemVersion offers a common approach for already widely used RDF models and a wide range of ontology languages.

Outline This deliverable describes and analyses five use cases (page 4). The main design goal of SemVersion is a separation of re-usable, language agnostic features from ontology language specific features.

In the design chapter (page 3) we identify language-neutral features like commit, branch and merge (page 17). Versioning specific metadata like last editor, time stamp of change, previous version, suggested changes etc. are also available regardless of ontology language. As ontology language dependant features we identified the diff operation. A simple, *structural diff* on the level of RDF models is always available, but the more meaningful *semantic diff* requires an inferencing engine *per ontology* language. Blank nodes make diff calculation harder. We discuss issues and solutions (page 22).

The API for dealing with versioning metadata is quite complex and cumbersome to program. Thus we developed a tool, RDFReactor¹, which *generates* a Java API from an RDF Schema.

For storage, SemVersion re-uses existing triple stores. To remain independent of a particular implementation, we developed a wrapper around common RDF management APIs: RDF₂Go (page A). Both tools are released separately and simpli-

¹<http://rdfreactor.ontoware.org>

fied implementation of SemVersion significantly; they deal with the lower-level data management parts. The SemVersion code base contains mainly code implementing versioning processes like commit, merge, branch and diff.

Contents

1	Introduction	1
1.1	Term Definitions	3
2	Requirements	4
2.1	Ontology Versioning Use Cases	4
2.1.1	Use Case 1: MarcOnt Collaborative Ontology Development . .	4
2.1.2	Use Case 2: The People’s Portal for Community Ontology Development	7
2.1.3	Use Case 3: Versioning the Gene Ontology	8
2.1.4	Use Case 4: Versioning in a Semantic Wiki	10
2.1.5	Use Case 5: Analysis of Wikipedia	11
2.2	Requirements Summary	12
3	Designing a Semantic Versioning System	13
3.1	Architecture	14
3.2	Storage	16
3.3	Generic Operations	17
3.4	Generic Metadata – SemVersion’s Data Model	18
3.5	User Management	20
3.6	API	20
4	Comparing Versions (Diff)	21
4.1	Set-Based Diff	22
4.2	Structural Diff	22
4.2.1	Computing A Diff Between Models With Blank Nodes	23
4.2.2	Blank Node Enrichment	27
4.2.3	Generating globally unique URIs	30
4.3	Semantic Diff	31
4.3.1	A Minimal Semantic Diff of RDFS Graphs	32
5	Using SemVersion	35
5.1	Walk-through	35
5.1.1	Typical Actions	37
5.1.2	Administration	38

5.1.3	Usage and Implementation Notes	39
5.1.4	SemVersion Usage Examples	39
6	Extending SemVersion	40
6.1	Package and Directory Structure	40
6.2	Extending SemVersion with RDF Schema Support	41
7	Conclusions and Outlook	42
A	RDF₂Go	46
A.1	What is RDF ₂ Go?	46
A.2	Working Example: Simple FOAF via RDF ₂ Go	48
A.3	Architecture	50
A.4	The API	51
A.4.1	Queries	54
A.5	How to get started	54
B	SemVersion Schema	55

Chapter 1

Introduction

SemVersion is a generic, extendable multi-language ontology versioning system, that can help research and industry to employ ontology based technologies in dynamic settings.

As outlined in the Knowledge Web Deliverable D2.3.1 „Specification of a methodology for syntactic and semantic versioning” [1], there is a clear need for RDF data and ontology versioning. This deliverable is a follow-up of D2.3.1, which explains the underlying concepts in detail. Here we focus on the concrete approach and implementation.

Change management for ontologies becomes a crucial aspect for any kind of ontology management environment, as engineering of ontologies often takes place in distributed settings where multiple independent users have to interact. There is also a variety of ontology languages used. Although RDF Schema and OWL are gaining more and more popularity, a lot of semantic data still resides in other formats, as it is the case in the biology domain (c.f. Sec. 2.1.3). Until now, no standard versioning system or methodology has arisen, that can provide a common way to handle versioning issues.

This deliverable describes the RDF-centric versioning approach and implementation *SemVersion*¹. It provides structural (purely triple based) and semantic (ontology language based, like RDFS, OWL and OBOL) versioning. It separates language-neutral features for data management from language-specific features like semantic diffs in design and implementation. This way SemVersion offers a common approach for already widely used RDF models and a wide range of ontology languages.

SemVersion is published as an open-source software project on the site OntoWare. The current version of the project homepage is depicted in Fig. 1.1.

Our approach is inspired by the classical CVS system [2] for version management

¹The name resembles the upcoming de-facto standard *subversion* (subversion.tigris.org) and is also a short form of „Semantic Versioning”



SemVersion

SemVersion versions your RDF data.

SeeAlso: [Ontoware Project Page](#) | [JavaDocs](#) | [RDFSchemadoc](#)

Features

- **Version your Models**
 - Support for commit and merge operations on models
- **Rich metadata support**
 - each Version has an Author, a provenance URI, a label, a URI
- **RDF commitment**
 - All data stored internally as RDF
 - Thanks to Jena it reads: RDF/XML, N3 or NT syntax

Download

SemVersion files are available at <http://ontoware.org/projects/semversion>

SemVersion builds on:

- Java 5.0
- [Jena](#): antlr.jar, commons-logging.jar, concurrent.jar, icu4j.jar, jakarta-oro-2.0.5.jar, jena.jar, xercesImpl.jar, xml-apis.jar, log4j-1.2.7.jar, [JUnit](#) (junit.jar needed only for running the test cases)
- [RDFReactor](#)
- NamedGraphs4Jena (NG4J)

License

SemVersion is released under the [GNU Lesser General Public License, Version 2.1, Feb. 1999](#).

Figure 1.1: Homepage of the SemVersion project

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

1.1 Term Definitions

RDF – **RDF** is a data model with the types **URI**, **blank node**, **plain literal**, **language tagged literal** and **data typed literal**. It consists of **triples** (also called **statements**).

model – A set of triples is called **model** (or triple set). SemVersion versions models.

ontology – An **ontology** is a model, in which semantics have been assigned to certain URIs and/or triple constructs, according to an **ontology language**.

concept – We use the term **concept** to denote things ontologies talk about: classes, properties and instances. In an RDF context, everything that is addressable by URI or by blank node is considered a **concept**.

versioned model – A model under version control is named a **versioned model**. A versioned model has a **root model**, which is a **version**. Technically, a versioned model consists of a triple set for the content plus an arbitrary number of statements *about* this model. We thus call our approach *model based versioning* in contrast to *statement based versioning*.

version – A version is a model plus versioning metadata. Versions in SemVersion never change. Instead, every operation that changes the state of a versioned model (commit, merge, ...) results in the creation of a new version.

More details about SemVersion’s conceptual data model can be found in Sec. 3.4.

Chapter 2

Requirements

In this chapter, we present use cases from within our workpackage. The industrial use cases stayed too vague to deduce a software architecture from them. We believe, that the uses cases given in the next section are diverse enough to lead to a system of general use. In Section 2.2 we sum up the requirements gathered from the use cases.

2.1 Ontology Versioning Use Cases

We gathered different requirements from Knowledge Web partners in order to create a generic design, suitable for a broad range of use cases. We tried to gather as many concrete usage requirements as possible to obtain a usable (and hence testable) design and implementation. In this section we present the different usage requirements.

For each use case, we name the stakeholder and provide a use case description, characteristics of the data set, and list derived versioning requirements.

2.1.1 Use Case 1: MarcOnt Collaborative Ontology Development

Stakeholder: Sebastian Ryszard Kruk (DERI), `sebastian.kruk@deri.org`

MarcOnt¹ is a project to create an ontology for library data exchange [3].

One of the most commonly used bibliographic description format is MARC21. Though it is capable of describing most of the features of the library resources, its semantic content is low. It means that while searching for a resource (book, paper),

¹<http://www.marcont.org/>

one has to look for particular keywords in the resource's description fields, but one cannot carry out a search by meaning or concept. Keyword search can often result in large sets of results. Also the data communication between library systems is very hard to extend (fixed structure). One of the earliest shared vocabularies is the Dublin Core Metadata standard for library resource description. Besides the fact that most of the information covered by MARC21 is lost, the full potential of the Semantic Web is not being used.

The project aims at creating the MarcOnt ontology, based on a social agreement that will combine descriptions from MARC21 together with DublinCore and makes use of the full potential of the Semantic Web technologies. This will include translations to/from other ontologies, more efficient searching for resources (i.e. users may have impact on the searching process).

The MarcOnt initiative is strongly connected to the Jerome Digital Library [4] project (e-library with semantics, formerly ElvisDL) - which implements a simple library ontology and can be used as a starting point for further work. MarcOnt also assumes that JeromeDL will be a testing platform for an experimental results from the MarcOnt initiative.

Data Set Currently there exists only one version of the MarcOnt ontology, which can be downloaded at http://www.marcont.org/index.php?option=com_content\&task=view\&id=13\&Itemid=27.

Versioning Requirements As the MarcOnt project aims to collaboratively create *the* ontology for digital libraries, they are a prototypical use case for collaborative ontology engineering in general. They intend to use the ontology at a certain point in time but continue to evolve it collaboratively at the same time. To conclude, they *need* ontology versioning.

The MarcOnt project has a clear view on the process of ontology evolution. It starts with a current main version. People can suggest (multiple, independent) changes to a version. Then the community discusses about the proposed changes and selects some. The changes are applied and a new main version is created and released. The process is illustrated in Fig. 2.1.

The ontology builder of the MarcOnt portal requires not only a GUI for building the ontology through submitting changes. It also needs the ability to:

- Manage a main trunk of the ontology (R1.1)²
- Manage versions of suggestions (R1.2)
- Generate snapshots of the main ontology with some suggestions applied (R1.3)

²Requirements are numbered by "use case number" / "." / running number

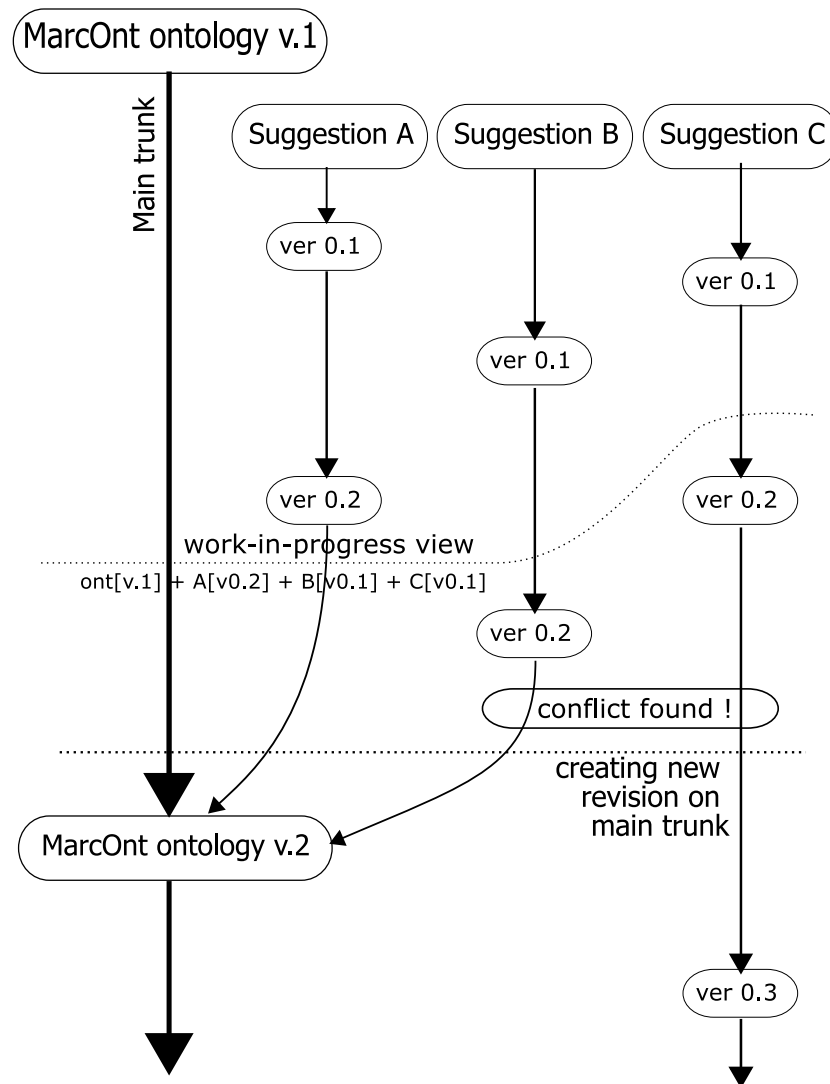


Figure 2.1: Versions and suggestions in the MarcOnt use case

- Detect and resolve conflicts (R1.4)
- Allow to add suggestions to the main trunk (R1.5)

2.1.2 Use Case 2: The People's Portal for Community Ontology Development

Stakeholder: Anna V. Zhdanova (DERI), anna.zhdanova@deri.at

People's portal [5] is an implementation of a human-Semantic Web interactive environment. The environment is named *The People's Portal* and it is implemented employing Java, Jena and Tomcat. The basic idea of the People's portal is to combine a community Semantic Web portal technology with collaborative ontology management functionalities in order to bring the Semantic Web to masses and overcome limitations of the existing community web portals.

Use cases: The People's portal environment is applied to DERI and used to produce part of the DERI web site. DERI members can login here to enter the environment. DERI web site managers can login here to manage the data in a centralized fashion.

Versioning Requirements The system uses a subset of RDF Schema. Users of the portal can introduce new classes and properties on the fly. Consensus is partly reached by usage. Properties that are often used and classes that have many instances are considered useful for the community. Hence it is necessary to ask the versioning system:

- How many instance does this class have now? Last week? Generalised: How many instances does a concept (`rdfs:Class` or `rdfs:Property`) has at a specific point in time? (R2.1)
- When has this class first been instantiated? (R2.2)
- How many properties are attached to this class? Since when? (R2.3) number of instances of class, properties NOW (specific point in time also)
- Who added this ontology item? (R2.4)
- Store new versions and return diffs between arbitrary points in time. (R2.5)
- Return predecessor of an ontology *item* (class, property) in time (R2.6)
- Support the evolution primitives: „add”, „remove” and „replace” on concept definitions. (R2.7)

- Return number of changed instance items (also properties, classes) and show which items changed. (R2.8)
- Which concepts appeared within a given time interval? (R2.9)
- Queries across change log/activity log: For each attribute, when was it instantiated and when have instances been created? (R2.10)
- What are “hot” attributes? Those instantiated or changed often recently. Which are these? (R2.11)

2.1.3 Use Case 3: Versioning the Gene Ontology

Stakeholder: Robert Stevens (U Manchester), `robert.stevens@manchester.ac.uk`

Use case description The gene ontology³ community is where collaborative ontology construction is practiced a long time comparing to other communities. The GO community showed that involvement of multiple parties is a must for a comprehensive ontology as a result. The GO community is one of the communities with the most practical experiences in collaborative ontology engineering [6], since they started already in 1999. Hence they are the ideal subject to study real-world change operations.

„The goal of the Gene Ontology (GO) consortium is to produce a controlled vocabulary that can be applied to all organisms even as knowledge of gene and protein roles in cells is accumulating and changing. GO provides three structured networks of defined terms to describe gene product attributes.”⁴

Current Gene Ontology versions are maintained by CVS repositories which handle only syntactic differences among ontologies. In other words CVS is not able to differentiate class versions for instance, being able only to differentiate text/file differences.

Data Set The Gene Ontology „per se” is not an Ontology in the formal sense, it is rather a cross-species controlled biological vocabulary as indicated above. The Gene Ontology is divided in three disjoint sub-ontologies, currently stored in big flat files or also stored in persistent repositories such as a relational database (MySQL database). The three sub-ontologies are divided into vocabularies that describe gene products in terms of: Molecular functions, associated biological processes and cellular components.

³<http://www.geneontology.org>

⁴Extracted from the OBO site <http://obo.sourceforge.net/>

The GO ontology permits to associate biological relationships among molecular functions, the involvement of molecular functions in biological processes and the occurrence of biological processes at a given time and space in cells [7]. Whereas the molecular function defines what a gene product does at the biochemical level, the biological process normally indicates a transformation process triggered or contributed by a gene product involving multiple molecular functions. Finally the cellular component indicates the cell structure a gene product is part of. The Gene Ontology contains around 19.000 concepts. The latest statistics about the GO can be found at the GO site ⁵:

Current term counts (as of June 20, 2005 at 6:00 Pacific time):

- 17946 terms, 94.2% with definitions.
- 6984 (38.9%) Molecular functions
- 9410 (52.4%) Biological processes
- 1552 (8.6%) Cellular components
- There are 998 obsolete terms not included in the above statistics (Total Terms=18944)

Further complexity assessments can be found at <http://www.fruitfly.org/~cjm/obol/doc/go-complexity.html>.

According to [8] the GO is a handcrafted ontology accepting only „is-a” and „part-of” relationships. The hierarchical organization is represented via a directed-acyclic-graph (DAG) structure similar to the representation of Web pages or hypertext systems. Members of the Consortium group contribute to updates and revisions of the GO. The Go is maintained by editors and scientific curators who notify GO users of ontology changes via email, or at the GO site by monthly reports⁶. Please note that ontology creation and annotation of GO terms in databases (association of GO terms with gene products) are two different operations. Each annotation should include its data provenance or source(a cross database reference, a literature reference, etc).

Technically, there are two different data sets, available via public CVS stores. Set I ranges from 1999 to 2001 and has a snapshot of the GO for each month in GO syntax. The second set runs from 2001 up to now and contains for each month a GO snapshot in OBO syntax. As OBO is the newer syntax, we assume the existence of a converter from GO syntax to OBO syntax available from the GO community (they must have converted their data, too). In order to use the data sets, one has to decide for an RDF-based format, as SemVersion can only handle RDF, as explained

⁵<http://www.geneontology.org/GO.downloads.shtml#ont>

⁶<http://www.geneontology.org/MonthlyReports/>

in Sec. 3.1. There are three options: (a) custom RDF, (b) OWL generated from DAG-Edit⁷ or (c) nice OWL generated by Protégé-Plugin.

Whatever choice is made, the exported data should contain the provenance information of the source file and the conversion process used. SemVersion offers ways to store such provenance information.

Versioning Requirements Essentially, here SemVersion is used for data analysis. In order to study ontology change operations, SemVersion must cope with multiple versions of the Gene Ontology (GO). The GO is authored in *Open Biology Language*⁸ (OBOL), for which usable OWL exports exist. The GO has about 19.000 concepts. Assuming about 10 statements per concept we estimate a size of roughly 190.000 statements – per version.

The researchers who study the ontology change patterns (Robert Stevens and his team) would like to use a monthly snapshot for a period of 6 years. This amounts to $6 \text{ years} \times 12 \text{ month} = 72$ versions. Thus the underlying triple store must be able to search (maybe even reason) over them.

The requirements in short form are thus

- Store 72 versions of an ontology consisting of 190000 statements (R3.1)
- Allow meta-data queries concerning individual versions and relations among them (e.g. “List all versions in which concept *X* is present, together with the last editor and a timestamp!”, “What is the last version edited by user *Y*?”) (R3.2)
- Allow data queries over the content of the 72 versions (R3.3)
- Compute a semantic diff for models in OBOL language (R3.4)
- OBOL to RDF mapping and converter (R3.5)
- A Java interface (R3.6) (this requirements stems from personal communication with Robert Stevens, who does not need a full-blown CVS-like protocol for his research)

2.1.4 Use Case 4: Versioning in a Semantic Wiki

Stakeholder: Max Völkel (U Karl), mvo@aifb.uni-karlsruhe.de

A wiki is a browser-based environment to author networked, structured notes, often in a collaborative way. The project *SemWiki*⁹ aims at creating a *semantic*

⁷<http://www.godatabase.org/dev/java/dagedit/docs/index.html>

⁸<http://obo.sourceforge.net/>

⁹<http://semwiki.ontoware.org>

wiki for personal note management. SemWiki extends the wiki syntax with means to enter statements about resources, much like in RDF. In a traditional wiki, users are accustomed to see and compare different versions of a page. In the semantic wiki „SemWiki” pages are stored as RDF resources, according to a wiki ontology.

Data Set A typical personal wiki has up to 3000 pages with approximately 10 versions per page. Each page consists roughly of 50 statements. This leads to approximately 1.5 million triples for a snapshot-based versioning system.

Versioning Requirements We can identify to levels off diff requirements. First, as the data is stored as RDFS models, the difference between two models can be computed via an RDFS diff. Second, the user would most likely prefer a diff suited for the domain of wiki pages. Such a domain-specific diff should be layered on top of SemVersion/RDFS. Conceptually this would be a post-processing of SemVersions output.

SemWiki users need ways to request a semantic diff between two page-versions. As SemWiki pages partly consist of RDF statements, which do not belong to a particular page, SemWiki needs a model-based (c.f. Sec. 3.4) versioning approach (R4.1). Sometimes users want to roll-back page changes, thus we need the ability to revert to old states (R4.2). Additionally, users want to track each statement: Who authored it, when has it been introduced, etc. (R4.3).

2.1.5 Use Case 5: Analysis of Wikipedia

Stakeholder: Denny Vrandecic (U Karl), Markus Kroetzsch (U Karl), Max Völkel (U Karl) {dvr,mkr,mvo}@aifb.uni-karlsruhe.de

An emerging research topic at AIFB is the analysis of changes in the Wikipedia¹⁰. Especially, the extension of Wikipedia with typed links, known as *Semantic Wikipedia* [9] is expected to generate a huge amount of fine-granular semantic data. Typed links allow users to give links to other pages a freely chosen type. The URL of the wiki page, the type, and the target page are interpreted as a triple. Another syntax allows users also to link to attribute values, which result in a literal object in the generated triple. This use case is mostly similar to „Versioning the Gene Ontology”, except that RDF content is already available and no converters need to be written.

Data Set The Wikipedia contains roughly 1.500.000 articles across all language versions. Each article is expected to generate 10 triples on average, resulting in a data set of 15 million triples, per version.

¹⁰<http://www.wikipedia.org>

Versioning Requirements There are no obvious requirements beyond those already mentioned in use case 3.

2.2 Requirements Summary

The uses cases described bear certain similarities. As a first observation, we can distinguish data management related requirements from ontology language specific features. As **data management requirements**, we identified:

- Store and retrieve versions in a scalable way (e.,g. **store up to 15 million triples**). This is possible with current triple stores, c. f. <http://esw.w3.org/topic/LargeTripleStores>.
- Uniquely identify versions (e. g. address versions unambiguously via URIs and user-friendly via labels)
- Manage **rich meta data per model** (e. g. provenance, author, valid time, transaction time)
- Model based versioning (and additionally concept-oriented queries for use case 2.1.2)
- **Queries across versions concerning meta data**
- Each version can have a number of attached „suggestions”; ability turn suggestions into official versions

Other requirements can not be dealt with in a language neutral way, as they are really **ontology language-dependant requirements**:

- Converter from ontology language to RDF (e. g. OBOL to RDF converter)
- **Queries across versions concerning the content**. This means **query answering over 15 million triples**.
- Ability to return **diffs between arbitrary versions**
- Ability to return **semantic diffs** for particular ontology language used (e.,g. RDF, OWL, OBOL)
- Domain-specific diff (e. g. diff between pages in a Semantic Wiki)

Chapter 3

Designing a Semantic Versioning System

We face several questions in the design of SemVersion:

Architecture. How can a single system handle many different ontology languages? In Sec. 3.1 we describe an architecture that holds up to the requirements.

Storage. Which storage strategy allows to execute queries and avoids vendor lock-in? In Sec. 3.2 we analyse the requirements and present a triple store abstraction layer as part of the solution.

Generic Operations. How are generic operations like commit, branch, and merge handled in SemVersion? What are the data formats of these operations? We describe them in Sec. 3.3.

Comparing Versions. How can changes between versions be computed? What's the right level of abstraction? The ability to handle arbitrary RDF requires to handle *blank nodes* as well, which complicates diff computation. Thus we explain all diff-related issues in its own chapter on page 21.

Generic Metadata. What metadata is needed for each ontology under version control? What for each version? Relations between versions? We describe SemVersion's data model in Sec. 3.4. The RDF Schema is given on page 55.

User Management. Collaborative authoring requires a certain support for user management. The SemVersion support is described in 3.5.

API. SemVersion allows a rich set of metadata, which is stored in RDF, described by an RDF Schema. How can such an API be kept consistent with the RDF Schema that describes it? We introduce a tool called RDFReactor in Sec. 3.6, which *generates* a Java API from an RDF Schema.

3.1 Architecture

Note: Different from the design described in [1], we now aim at creating a pure Java library. This allows us to concentrate on the core functionality, algorithms and data structures; additional HTTP-based web services will be added as required.

We distinguish four kinds of data, to be managed in SemVersion:

Versioned model – the *content* of a model, representing an ontology in a given ontology language

Versioning metadata per model – metadata managed by SemVersion, described by an RDF Schema, and exposed via a Java API, e.g. `getLastMainBranchVersion()`, `getLastEditedTime()`

User-supplied metadata per model – arbitrary metadata

Management data – relations between versions; user management; projects and other global data

We generalise from the use cases: A versioning system has generally two main parts. One deals with *general data management* issues, the other part with *versioning specific functionality* such as calculating the difference between two versions.

The key to separate the two is the intuition that *RDF represents the structural core of ontology languages*. By creating a generic RDF data management system, we can implement the data management parts once and re-use them for all ontology languages. A more detailed discussion can be found in the Knowledge Web Deliverable D2.3.1 [1]. In SemVersion, we use RDF to represent each of these types of data. In the next paragraph we give our reasons to use RDF for storing content of ontologies.

RDF triples as the structural core of ontology languages The most elementary modelling primitive that is needed to model a shared conceptualisation of some domain is a way to denote entities and to unambiguously reference them. For this purpose RDF uses URIs, identifiers for resources, that are supposed to be globally unique. Every ontology language needs to provide means to denote entities. For global systems the identifier should be globally unique. Having entities, that can be referenced, the next step is to describe relations between them. As relations are semantic core elements, they should also be unambiguously addressable. Properties in RDF can be seen as binary relations. This is the very basic type of relations between two entities. More complex types of relations can be modelled by defining a special vocabulary for this purpose on top of RDF, like it has been done in OWL.

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

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

Layered Architecture The high-level architecture of the implementation consists of several layers (c.f. Fig. 3.1). Each layer depends on the layer below. To users, only the SemVersion API and the RDF₂Go API (c.f. page 46) are exposed.

- SemVersion API – A facade API for everything from user management, to committing versions, creating projects, branching and merging the version tree, retrieving or setting metadata, asking queries, to manipulating models. Described in greater detail in Chapter 5.
- SemVersion internal data structures (like Branch, Root, User, VersionedModel, Session and Diff)– these are rich in functionality, but too complex as an external API (mostly generated with RDFReactor).
- RDFReactor – generated glue-code to bridge from object-oriented Java to triple-centric RDF
- RDF₂Go – a wrapper over triple (and quad) stores, prevents vendor lock-in.
- Existing quad store implementations (NG4J, YARS).

All versioned content is stored as separate RDF models. At startup time, a SemVersion server loads its root data model from the same RDF store, but caches this one in memory. The root model contains only information about projects, versioned models, their versions, branch labels, last edited time and other metadata about models. User-defined metadata is stored as separate RDF models in the RDF store. Diffs are calculated on-the-fly in the SemVersion server, but could be cached.

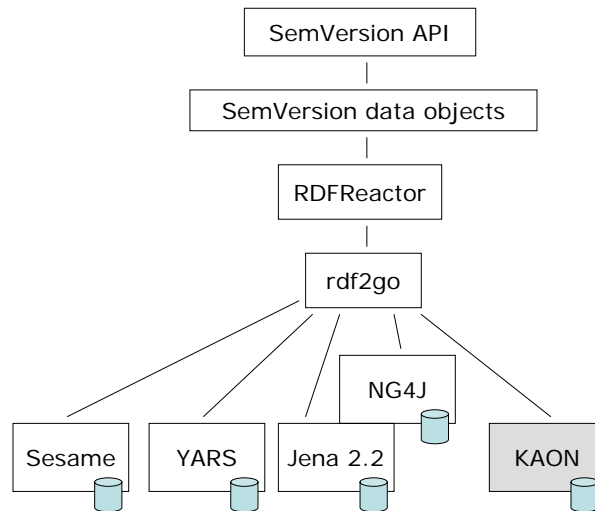


Figure 3.1: The Layered Architecture of SemVersion

3.2 Storage

The requirements for SemVersion’s storage are:

- Query answering across multiple versions (1)
- Diffs between arbitrary versions (2)
- No vendor lock-in (3)

To be meet requirements (1) and (2), we decided to store all RDF models materialised, e.g. not as diffs. Instead, diffs are computed on demand. More about diffs can be found in Chapter 4.

SemVersion utilized RDF₂Go, a sub-project of SemVersion. RDF₂Go¹ is a wrapper over RDF triple and quad stores. It abstracts away the triple store implementation and gives the user a simple Java-centric API for model changes. We describe RDF₂Go in depth in the appendix (page 46).

In [1] we suggested reification for data storage, which would make models four times as large. Facing the large volume of the Gene Ontology data (see 2.1.3), we need more powerful storage solutions than for the other use cases. To avoid reification, we now use native quad stores, which provide a context URI for each triple [15]. We use the context URI to address models more efficiently. RDF₂Go has the ability to access parts of a quad model at runtime as a triple model. This simplifies RDF handling in SemVersion significantly.

¹<http://rdf2go.ontoware.org>

Implementation

The storage layer access is implemented in the class `TripleStore` which offers means to get models. The `TripleStore` uses a `ContextModel` for its persistence. The identification URI for a model is used as the context URI in the quad model. All models are only proxies for the `ContextModel`. Currently the data is stored as an XML file in *TriX* [16] syntax. We plan to migrate to *Sesame 2.0* as soon as it is released.

3.3 Generic Operations

Most versioning operations can be classified as pure data management operations, agnostic with respect to the versioned data semantics.

In a **commit** operation, a user commits a model as an update to a former model. Each new version is simply stored *as is* – no incremental storage via diffs is used. This guarantees that the retrieval will give the user exactly back what she checked in (syntactic versioning). More importantly, this allows to compute diffs between arbitrary versions and *execute queries across versions*. Each new model is sent to the RDF store with a locally generated URI, which is globally unique. Relations to other models are recorded in the main model.

More sophisticated storage mechanisms could be developed, but the real challenge in ontology versioning is not *storage space*, but efficient query answering for large amounts of (quad) statements.

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

For merge, we distinguish a merge between two arbitrary versions and the merging of two branches. It is possible to merge arbitrary versions, not only those at the end of a branch. A merge of version *A* and version *B* is the set union of the triple sets. If one version is the predecessor of the other one, duplicate blank nodes have to be detected (via blank node enrichment, see also Sec. 4.2.2 and removed).

Merging two branches is different. First we look at the branch point *c*, which is defined as the most recent common version of the two branches. Such a version always exists, as branches can only be created by committing a version to an existing version. We also take two versions from the different branches, in most cases the most recent ones, and call them *a* and *b*. Consider the example version tree given in Fig. 3.2. Here $c = A, a = A'', b = B''$. In order to merge *b* back into *a* we compute the $diff(c, b)$ and apply it to *a*.

3.4 Generic Metadata – SemVersion’s Data Model

The general idea is to re-use data management functionality across ontology languages. The *relations between different versions* of an RDF model or ontology are the same, regardless of the semantics used.

Data *management* deals with storage and retrieval of chunks of data. In our case, the smallest unit of data we *store and retrieve* is a model (also called ‘triple set’). A model is a set of RDF triples. A versioned model consists of a triple set for the content plus an arbitrary number of statements *about* this model. We thus call this *model based versioning* in contrast to *statement based versioning*. Note that we need the more fine-granular structure of RDF to compute diffs.

SemVersion’s data model (depicted in Fig. 3.3) was mostly derived from the requirements of the MarcOnt use case (Sec. 2.1.1) and fulfills also the other use cases needs. But note that the statement-oriented versioning requirements from use case 2 seem difficult to integrate.

repository SemVersion has a **repository** of **projects**.

project They can be created, listed and removed from the repository. A project can hold a number of **versioned models**, i.e. in a project several different and independent ontologies can be developed and versioned.

versioned model A versioned model is the container for a single RDF model or ontology under version control. A versioned model has a **root version** and also knows all other **versions** that are direct or indirect descendants of the root version. Versioned models are quite an important concept and give the user the ability to retrieve the right version by e.g. listing all branches or simple getting the most current „main” branch version.

version A **version** is the most central concept. It is a model decorated with all kinds of metadata. A version knows where it comes from (it parents), has a branch, a label and optionally even a comment and a provenance URI. The user can commit a **model** as the successor of a version; create a new version

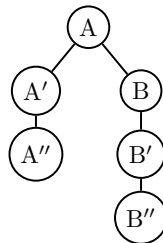


Figure 3.2: A sample version tree

by merging two existing models or commit a diff. Committing diffs is useful, if the models become really large and change only little – a use case we are likely to experience in the Gene Ontology scenario.

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

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

Users can store *arbitrary RDF encoded metadata* objects for each project, versioned model and most important for each version. This data is stored in the RDF storage layer and linked by RDF statements to the versioning artefact it belongs to. Metadata models are also URI-addressable. This metadata strategy enables a

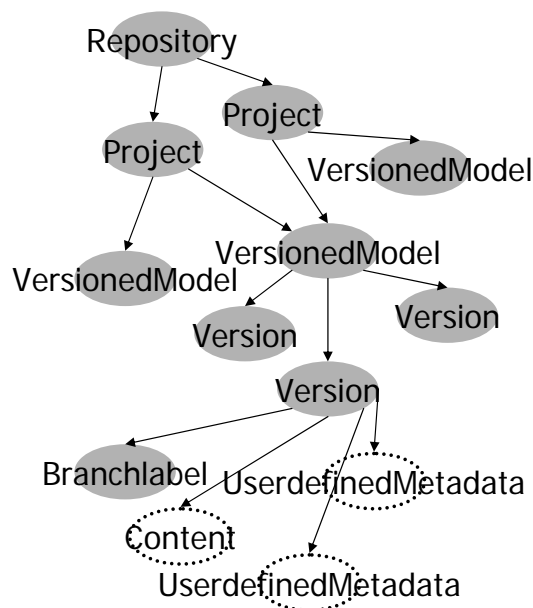


Figure 3.3: Data Model for RDF Versioning

good re-use of the SemVersion system, as e. g. the evolution log of an ontology engineering tool could be assigned to a version with this mechanism.

3.5 User Management

The SemVersion API has means to create users and set login names and passwords for them. The user management is not optimized for security, but against accidental misbehavior. All user data is stored in plain RDF in the same backend as all other data. It's thus the task of SemVersion clients to ensure that no direct access to the underlying RDF store is possible – if strong security is needed at all.

3.6 API

The metadata model of SemVersion is rather expressive. Creating a Java API suitable for adding, retrieving and changing all metadata requires a great deal of glue code to translate `get`-methods in RDF queries, and `add`/`set`-methods in statement manipulation operations. The same holds for SemVersions internal data management (relations between versions, user management, adding projects, setting the parents of a version or storing the branch label).

Luckily, this error-prone glue-code is generated from an RDF Schema by RDFReactor. RDFReactor² [17] is released as an independent open-source project. Parallel to the generated Java API, direct access to the stored RDF data is always possible via RDF₂Go. RDFReactor builds on RDF₂Go. Until now, RDFReactor has been downloaded over 140 times.

²<http://RDFReactor.ontoware.org>

Chapter 4

Comparing Versions (Diff)

In this chapter we describe in detail all concepts and issues related to diff calculation and its role in SemVersion. The biggest challenge is correct handling of blank nodes, as they cannot be referenced across models.

A semantic versioning system needs the ability to *compute* a *diff*, which tells the user what (conceptually) has changed between two versions. Compare this to the situation in source code versioning, where changes are reported on the syntactical level. If a colleague used a different source code formatting tool, a syntactic diff will report many changes, while semantically nothing has changed in the program code. Users are thus not really interested in a line-by-line diff, but rather in the *semantic diff*.

Diffs serve two purposes: First, SemVersion allows to *compute* (structural and semantic) diffs between two arbitrary chosen models, to *inform* the user about changes. This allows collaborative ontology engineering. Second, diffs can be used in an update command to *apply changes* to a remotely stored model. When dealing with very large models, it might not be feasible (nor efficient) to transfer the complete model, if only a small fraction has changed.

We distinguish three levels of diffs, which are described in detail in the next sections:

- Set-based Diff which is easy to compute, but may result in large diffs.
- Structural Diff which takes blank node semantics into account.
- Semantic Diff which respects the ontology language semantics. This is the most concise diff.

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

version A: a rdfs:type c b rdfs:type c c rdfs:subClassOf d	version B: a rdfs:type d b rdfs:type d c rdfs:subClassOf d e rdfs:type d
added $a(A, B)$: a rdfs:type d b rdfs:type d e rdfs:type d	removed $r(A, B)$: a rdfs:type c b rdfs:type c

Figure 4.1: Example for a Set-Based Diff

- $a(A, B) = B - A = B \setminus (A \cap B)$ and
- $r(A, B) = A - B = A \setminus (A \cap B)$.

Note: We represent a diff as two independent RDF models: One for added statements ($a(A, B)$), one for removed statements ($r(A, B)$).

4.1 Set-Based Diff

For versioning, the **set-based diff** is simply the set-theoretic difference of two RDF triple sets. Libraries such as Jena have built-in functions to compute this set-difference. An example for a set-based diff can be found in Fig. 4.1. Note that the set-based *does not* uses any inferencing.

Such diffs can be computed by simple set arithmetics for triple sets that contain only URIs and literals, as shown in [11].

In the Example 4.2, we have a model consisting of two statements, both involving the same blank node.¹ The exact same model is later stored as an update. As blank nodes have no identity across model boundaries, a set-based RDF diff concludes that all triples have been removed from A , while a similar amount of triples has been added to B .

4.2 Structural Diff

Without the presence of blank nodes, the set-based diff is the same as the structural diff. With blank nodes, the set-based diff considers all blank nodes to be different and reports all statements involving blank nodes both as added and as removed.

¹Note that we label blank nodes with $_1, _2 \dots$ in our examples.

version A: _:1 :hasName "Max" _:1 :hasPhone "123"	version B: _:3 :hasName "Max" _:3 :hasPhone "123"
added $a(A, B)$: _:6 :hasName "Max" _:6 :hasPhone "123"	removed $r(A, B)$: _:7 :hasName "Max" _:7 :hasPhone "123"

Figure 4.2: A naive set-based “structural” RDF Diff

Unfortunately, *blank nodes*² are used in practice. They are e. g. used in OWL for property restrictions and in FOAF to denote persons. Blank nodes cause some problems in computing the structural diff, as we have no knowledge about the relation (equal or not?) between two blank nodes from different models.

Note: Applying a diff faces the same blank node identification problems as for computing a diff. The RDF semantics [13] dictate to treat blank nodes across models as different, as long as they are not making exactly the same statements. If a model contains ten statements about a blank node and we add one statement to it, a naive diff thus concludes that all ten statements have been removed and eleven other statements have been added.

In the remainder of this section, we analyse cases in which we can conclude blank node equality in a versioning context. In Section 4.10, we explain a way to track the identity of blank nodes across versions.

4.2.1 Computing A Diff Between Models With Blank Nodes

In Example 4.2 it seems obvious to come up with a better diff. In fact, both models encode exactly the same semantic knowledge: There exists a thing which has a name (Max) and a phone number (123). So it’s safe to treat the blank nodes as equal across versions. On the other hand, such equality is not always safe to deduce. Consider the Example 4.3. Here it would be illegal to consider `_:1` and `_:3` to be the

²Sometimes also called „bnodes” – but they have nothing to do with b-trees.

version A: _:1 :hasName "Max"	version B: _:3 :hasPhone "123"
added $a(A, B)$: _:5 :hasPhone "123"	removed $r(A, B)$: _:2 :hasName "Max"

Figure 4.3: Blank nodes can not be considered equal

same node. Doing so would say: I know that this it is the same object, that had the name “Max” in the last version and now has the phone number “123”. It might as well mean: The object with the name “Max” has been replace by an object with the phone number “123”. Now we look at different cases of diffs, trying to unify blank nodes across versions, where appropriate. We can distinguish five cases of blank node differences across models:

Case 1: Total Similarity.

The blank node appears in exactly the same statements in both versions. Then it is semantically safe to assume blank node identity (c.f. Ex. 4.4).

Case 2: Single Monotonous Extension.

If only additional statements have been added (the previous version’s statements are all entailed by this versions statements), one can also conclude blank node equality without problems (c.f. Ex. 4.5).

Case 3: Multiple Monotonous Extension.

If the blank node has been extended monotonously but in multiple and different ways (c.f. Example 4.6), what does this mean? Are the blank node identifiers „_:1” and „_:3” referring to the same object? As we cannot know this, we must treat the blank nodes as different. However, the blank node in model A could either be identified with _:3 or _:5. We discuss this in Section 4.3.1.

Case 4: Extension and Reduction.

Consider the case, where some properties have been removed and others have been added. Here again, no equality of the blank nodes should be assumed. Example 4.7 shows such a case.

Case 5: Multiple Extensions and Reductions

The original blank node could appear only in reduced and extended models. Here again, no equality should be assumed. Note that this is the most general case, which also handles the case of blank nodes completely being dropped from the model. An example is given in 4.8

version A: _:1 :hasName "Max" _:1 :hasPhone "123"	version B: _:3 :hasName "Max" _:3 :hasPhone "123"
added $a(A, B)$: \emptyset	removed $r(A, B)$: \emptyset

Figure 4.4: Case 1: Total Similarity

version A: _:1 :hasName "Max" _:1 :hasPhone "123"	version B: _:3 :hasName "Max" _:3 :hasPhone "123" _:6 :hasAge "26"
added $a(A, B)$: _:7 :hasAge "26"	removed $r(A, B)$: \emptyset

Figure 4.5: Case 2: Single Monotonous Extension

version A: _:1 :hasName "Max"	version B: _:3 :hasName "Max" _:3 :hasPhone "123" _:5 :hasName "Max" _:5 :hasPhone "456"
added $a(A, B)$: _:6 :hasName "Max" _:6 :hasPhone "123" _:2 :hasName "Max" _:2 :hasPhone "456"	removed $r(A, B)$: _:4 :hasName "Max"

Figure 4.6: Case 3: Multiple Monotonous Extension

version A: _:1 :hasName "Max" _:1 :hasPhone "123"	version B: _:8 :hasName "Max" _:8 :hasAge "28"
added $a(A, B)$: _:5 :hasName "Max" _:5 :hasPhone "456"	removed $r(A, B)$: _:3 :hasName "Max" _:3 :hasPhone "123"

Figure 4.7: Case 4: Extension and Reduction

version A: _:1 :hasName "Max" _:1 :hasPhone "123"	version B: _:8 :hasName "Max" _:8 :hasAge "28" _:6 :hasPhone "123" _:6 :hasFax "4711"
added $a(A, B)$: _:5 :hasName "Max" _:5 :hasPhone "456"	removed $r(A, B)$: _:8 :hasName "Max" _:8 :hasAge "28" _:6 :hasPhone "123" _:6 :hasFax "4711"

Figure 4.8: Case 5: Multiple Extensions and Reductions

4.2.2 Blank Node Enrichment

As a solution, we invented the concept of *blank node enrichment*.

In order to identify blank nodes across models, we must assign a unique identifier to them. In order not to break existing semantic interpretations of the model, we cannot simply replace blank nodes with URI nodes. Instead, we add additional statements to the model that attach URIs as inverse functional properties to blank nodes, as illustrated in Example 4.10.

Integration in SemVersion Every model stored in SemVersion is *blank node enriched* at insertion time. If a user retrieves this model later, he will additionally find triples attaching random URIs to blank nodes. During his local ontology change operations, three cases can happen with respect to blank nodes:

Blank node deletion. A blank node is deleted. Either the ontology editor also deletes *all* statements about this blank node, or leaves the blank node enrichment statement intact. In either case, when the user commits his model back as a new version, SemVersion can deduce that this node has been deleted.

Blank node changes. Some (but not all) properties involving the blank node are added or removed. Upon commit, this blank node can be linked to its ancestor and thus a correct diff can be computed upon request.

Blank node creation. The user introduces a new blank node. This can happen directly or indirectly, e.g. in OWL when a user adds a new constraint.

Later, when the user commits such an enriched model back to SemVersion, it is simply stored, enriching only the not-yet-annotated blank nodes. Even later, a diff between two versions might be requested. Now, SemVersion first computes the structural diff (c.f. Fig. 4.11). This diff is then minimized by removing blank nodes which occur both add the added and removed side, but have the same `bne:hasID` ID (c.f. Fig. 4.12). The algorithm for blank node enrichment is given in Fig. 4.10.

Most RDF processing tools will leave the extra triples intact. In the MarcOnt scenario (c.f. Sec. 2.1.1), a dedicated ontology builder is used, so this constraint can be enforced. In SemVersion, the content of every version is blank node enriched before it is stored in the RDF storage layer.

To sum up, the idea is to attach artificial inverse functional properties to every blank node. This changes nothing to the RDF semantics but helps to identify equal blank nodes across models. The generation of unique identifiers (we use URIs) is described in the next section.

Model A:

```
_:3 rdf:type foaf:Person
_:5 foaf:name "Max"
```

Model A' (blank node enriched):

```
_:3 rdf:type foaf:Person
_:3 bne:hasID rnd:1130937311/416
_:5 foaf:name "Max"
_:5 bne:hasID rnd:1130937738/417
```

Figure 4.9: Blank Node Enrichment

```
Map<BlankNode, URI> knownBnodes = INITIALIZE EMPTY;
for each Statement s in Model m:
  if (s.subject is a BlankNode)
    // make sure subsequent bnodes get the same URI
    URI u = knownBnodes.get( s.subject )
    if (u = null ) // this bnode has not been seen before
      u = randomUniqueURI();
    knownBnodes.put( s.subject, u );
    // write bnode enrichment into model
    model.addStatement( s.subject, bne:hasID, u );
  if (s.object is a BlankNode)
    URI u = knownBnodes.get( s.object )
    if (u == null ) // this bnode has not been seen before
      u = randomUniqueURI();
    knownBnodes.put( s.object, u );
    model.addStatement( s.object, bne:hasID, u );
```

Figure 4.10: Blank Node Enrichment Algorithm

version A: _:3 rdf:type foaf:Person _:3 bne:hasID rnd:416 _:5 foaf:name "Max" _:5 bne:hasID rnd:417	version B: _:4 rdf:type foaf:Person _:4 bne:hasID rnd:416 _:2 foaf:name "Max" _:2 bne:hasID rnd:417 _:2 foaf:phone "123" _:7 foaf:name "Max" _:7 foaf:phone "456" _:7 bne:hasID rnd:426
added $a(A, B)$: _:8 foaf:name "Max" _:8 bne:hasID rnd:417 _:8 foaf:phone "123" _:1 foaf:name "Max" _:1 foaf:phone "456"	removed $r(A, B)$: _:6 foaf:name "Max" _:6 bne:hasID rnd:417

Figure 4.11: Simple Structural Diff between blank node enriched models

version A: _:3 rdf:type foaf:Person _:3 bne:hasID rnd:416 _:5 foaf:name "Max" _:5 bne:hasID rnd:417	version B: _:4 rdf:type foaf:Person _:4 bne:hasID rnd:416 _:2 foaf:name "Max" _:2 bne:hasID rnd:417 _:2 foaf:phone "123" _:7 foaf:name "Max" _:7 foaf:phone "456" _:7 bne:hasID rnd:426
added $a(A, B)$: _:8 foaf:name "Max" _:8 bne:hasID rnd:417 _:8 foaf:phone "123" _:1 foaf:name "Max" _:1 foaf:phone "456"	removed $r(A, B)$: _:6 foaf:name "Max" _:6 bne:hasID rnd:417

Figure 4.12: *Smart* Structural Diff between blank node enriched models

4.2.3 Generating globally unique URIs

The Blank Node Enrichment relies on a way to produce an arbitrary number of globally unique URIs.

The strategy for generating globally unique URIs in SemVersion is as follows: (i) The first part of the URI is the URL the SemVersion server is running at. This reduces the problem of generating globally unique URIs to generating locally unique URIs, assuming that the same SemVersion server URL will not be used for different SemVersion server ever. To soften this constraint, (ii) the current system time for the server, measured in milliseconds since 1970 is also made a part of the generated URL. Thus the problem is reduced to maintain an accurate server clock and never issue the same URI again in a given period of time (server clock may be off for minutes, but not months). To issue different URIs at all times, (iii) an internal counter is added to the URI string. The URI generator cannot guarantee uniqueness, but the likelihood for the same URI being generated twice is really low.

Estimating the chance for generating the same URI twice is easy. The same URI is generated with a likelihood $P_{error} = P_{servername} * P_{time} * P_{counter}$, with the following definitions:

$P_{servername}$ the likelihood of using the same base URL on two servers. The base URL of a SemVersion server is the URL under which SemVersion runs. It has to be configured in a configuration file and thus can be guaranteed to be unique.

P_{time} the likelihood of the server clock reporting the same time twice. We assume this happens only for one millisecond per year.

$P_{counter}$ the likelihood of the server processor malfunctioning. We estimate this to be really low (roughly 1 : 1.000.000).

An example for such a generated URI is `http://semversion.example.com/1130937311/416` which can be represented locally as `rnd:1130937311/416`, assuming correct name space definitions.

4.3 Semantic Diff

The semantic difference has to take the semantics of the used ontology language into account. It is not possible to write a generic, semantics-agnostic algorithm for this.

An intuitive way to understand the concept of a semantic diff goes like this: Let's assume we use RDF Schema as our ontology language, and have two versions (A and B) of an RDFS ontology. Now, in order to compute the semantic RDFS diff, we use RDF Schema entailment on model A and infer all triples we can ($Inf(A)$). Then we do the same for model B ($Inf(B)$). Now we calculate a structural (syntactical, set-based) diff on $Inf(A)$ and $Inf(B)$. Fig. 4.13 illustrates the semantic diff under RDF Schema entailment semantics.

Note: This is not the same as the structural diff between model A and B. If the structural diff of two models is empty, then the semantic diff must also be empty. The inverse is not necessarily true: There might be two different RDF models which encode the same semantic model, resulting in an empty semantic diff, but a non-empty structural diff.

More formally, to calculate the semantic diff d_l under the semantic of an ontology language l , a system has to know the semantics of that specific language l . The *semantic closure* $s_l(M)$ of a model M is then defined as the set of all statements that can be concluded from the statements in M under the semantics of the RDF-based ontology language l . The *semantic diff* of two models A and B is $d_l(A, B) \rightarrow \langle a_l(A, B), r_l(A, B) \rangle$ with $a_l(A, B) = s_l(B) \setminus (s_l(A) \cap s_l(B))$ and $r_l(A, B) = s_l(A) \setminus (s_l(A) \cap s_l(B))$.

Briefly, a way to compute a semantic diff is to materialize the complete entailment (transitive closure) and then perform a structural diff. For RDF Schema, the calculation of the transitive closure can be re-used from the Jena framework. However, in certain cases this might not be feasible, e.g. some ontology languages infer infinite amounts of triples from finite models. In these cases, the calculation of a semantic diff can be accomplished by a language specific reasoner or by a language specific set of rules. These rules can be formulated in a language like TRIPLE as demonstrated in [12]. Initially, SemVersion provides support for RDF Schema only.

version A:	version B:
a rdfs:type c	a rdfs:type d
b rdfs:type c	b rdfs:type d
c rdfs:subClassOf d	c rdfs:subClassOf d
	e rdfs:type d
added $a(A, B)$:	removed $r(A, B)$:
e rdfs:type d	

Figure 4.13: Example for a Semantic Diff under RDFS semantics

4.3.1 A Minimal Semantic Diff of RDFS Graphs

(Contribution from Free University of Bolzano)

In the following we consider RDFS as the ontology language.

The scenario is the one where an ontology engineer generates different versions of an ontology while those different versions need to be stored and retrieved. The problem we are interested in is to find the minimal information that must be stored together with the original ontology to reconstruct the new version. In this scenario, the notion of semantic diff must be useful both to enlighten the differences between two different versions (hereafter called the source and the target ontology), and as the minimal piece of information that we need to store together with the source ontology to obtain the target ontology.

Note: SemVersion currently stores full versions and generates diffs only on demand.

The main idea is to start from a simple structural diff between the two versions and then minimizing the result of the structural diff by exploiting the semantics of the RDFS ontology language. In particular, considering the structural diff as a pair of sets, $a(A, B)$, $r(A, B)$, we should exploit the semantic of RDFS and in particular the notion of entailment between two RDFS graphs (as defined in [13]) to eliminate:

- Triples from $a(A, B)$ which are entailed by the source (we call the resulting set the semantic add, in symbols SE_{Add});
- Triples from $r(A, B)$ which are entailed by the target (we call the resulting set the semantic remove, in symbols SE_{Rem}).

To formally define the notion of semantic diff it is crucial to find a way to eliminate triples as described above in such a way that:

1. Both sets SE_{Add} and SE_{Rem} are minimal;
2. The target ontology can be reconstructed from such semantic diff, i. e.:

$$O_t \approx O_s \cup SE_{Add} - SE_{Rem}.$$

Two relevant remarks follows that clarify the importance of having a semantic diff and the difficulties related with its definition due to the presence of blank nodes and the removal of information that is implicit in the target graph.

Remark 1. Due to the presence of blank nodes the semantic diff is not unique. For example, let us consider Example 4.14, with $O_s = A$ and $O_t = B$.

We can consider two different minimal sets as semantic add depending whether we identify $_ : z$ to $_ : x$ and $_ : z$ to $_ : y$, respectively:

version A: a :R _:x _:x rdf:type C a :R _:y _:y rdf:type E	version B: a :R _:x _:x rdf:type C a :R _:y _:y rdf:type E a :R _:z _:z rdf:type C _:z rdf:type E
added $a(A, B)$: a :R _:z _:z rdf:type C _:z rdf:type E	removed $r(A, B)$:

Figure 4.14: Structural Diff

- If we identify the blank node _:z with _:x, thus
 $SE_{Add} = \{(_:x \text{ rdf:type } E)\}$
- If we identify the blank node _:z with _:y, thus
 $SE_{Add} = \{(_:y \text{ rdf:type } C)\}$

Furthermore, if the source and target ontologies do not contain the triple ($_:y \text{ rdf:type } E$) then the second mapping (i.e. $_:z$ to $_:y$) does not provide a valid semantic diff since it does not minimize SE_{Add} .

Remark 2. The notion of semantic remove can be counterintuitive due to triples that are implied by the target ontology. Let us consider the following versioning case given in Example 4.15. From the transitivity of subclass relations: O_t entails (A

version A: A rdfs:subClassOf B A rdfs:subClassOf C	version B: A rdfs:subClassOf B B rdfs:subClassOf C
added $a(A, B)$: B rdfs:subClassOf C	removed $r(A, B)$: A rdfs:subClassOf C

Figure 4.15: Structural Diff

rdfs:subClassOf C), thus: $SE_{Rem} = \{\}$. Thus, if we reconstruct the target ontology by using the semantic diff we obtain:

$$O'_t = \{ (A \text{ rdfs:subClassOf } B) (A \text{ rdfs:subClassOf } C) (B \text{ rdfs:subClassOf } C) \}$$

i.e. the resulting graph contains the redundant triple (A rdfs:subClassOf C). In order to remove this triple we need to process furthermore the target graph that removes redundant information. Note that the semantic diff as defined in [14] has the same problem. We envisage various alternative solutions.

- To adopt the semantic diff as defined here in conjunction with a graph minimization procedure that removes redundant information (this further step is not for free).
- To avoid minimization over the remove part, leaving just the syntactic remove.
- To resort to a new notion of update where removing means asserting the negation of the triples in the structural remove. This also implies that we should further modify the target graph to obtain a consistent target ontology. This last solution does not verify anymore the equivalence: $O_t \approx O_s \cup SE_{Add} - SE_{Rem}$.

To actually compute the semantic diff a simple case is the one where the target ontology does not introduce new blank nodes. In this case we can show that the semantic diff can be obtained by simply checking entailment between source and target ontologies and single triples belonging to $a(O_s, O_t)$ and $r(O_s, O_t)$, respectively.

In this case we can show that the semantic diff can be obtained by simply checking entailment between source and target ontologies and single triples belonging to $a(O_s, O_t)$ and $r(O_s, O_t)$, i.e.:

1. $SE_{Add}(O_s, O_t) = a(O_s, O_t) - \{t \in a(O_s, O_t) | O_s \text{entails } t\}$
2. $SE_{Rem}(O_s, O_t) = r(O_s, O_t) - \{t \in r(O_s, O_t) | O_t \text{entails } t\}$

Example. The case of ground ontologies, as showed in Fig. 4.1, can be solved applying the above mentioned minimization rules. Thus, the semantic diff is:

$$\begin{aligned}
 SE_{Add}(O_s, O_t) &= a(O_s, O_t) - \{(ardfs : typed), (brdfs : typed)\} \\
 &= \{(e \text{ rdfs:type } d)\} \\
 SE_{Rem}(O_s, O_t) &= r(O_s, O_t) - \{\} \\
 &= \{(a \text{ rdfs:type } c), (b \text{ rdfs:type } c)\}.
 \end{aligned}$$

version A: a rdfs:type c b rdfs:type c c rdfs:subClassOf d	version B: a rdfs:type d b rdfs:type d c rdfs:subClassOf d
added $a(A, B)$: a rdfs:type d b rdfs:type d e rdfs:type d	removed $r(A, B)$: a rdfs:type c b rdfs:type c

Figure 4.16: Structural Diff

Chapter 5

Using SemVersion

In this chapter we describe how developers can actually use SemVersion to solve their versioning tasks. In the next chapter, we explain how SemVersion can be extended to handle other ontology languages.

How to get started

First, download SemVersion from <http://semversion.ontoware.org>. Second, configure the file `semversion.properties` to point to a storage directory on your hard drive. Third, write your Java application, using the SemVersion API.

5.1 Walk-through

Probably the best way to explain how to use the Java library SemVersion is to show real source code. In this chapter we get practical and show a variety of commented (!) source code fragments. A possible typical work session with SemVersion involves the following steps:

- Start SemVersion
- Log in with username and password and obtain a *session*.
- Get, list or create a *VersionedModel*.
- Get most recent or list all *Version* objects.
 - Read or update a versions metadata
 - Commit a new model as a child-version
- Request a diff between two *Version* objects

- Query the SemVersion repository

- End session

5.1.1 Typical Actions

Commit initial model

```
// log in
Session userSession = semVersion.login("tom", "password");

// get VersionedModel by label
VersionedModel vm = userSession.getVersionedModel("Gene Ontology");

// obtain an empty Model
Model myFirstModel = userSession.getModel();

// manipulate the model
URI tool = URIUtils.createURI("http://example.com/#Tool");
myFirstModel.addStatement(
    URIUtils.createURI("http://semversion.ontoware.org"),
    RDF.type,
    tool);

// commit as first version
vm.commitRoot(myFirstModel, "version1");

// log out
userSession.close();
```

Commit a suggestion to a version

```
// log in
userSession = semVersion.login("joe", "password");

// get versionedmodel by name and fetch root
Version root = userSession.getVersionedModel("Gene Ontology").getRoot();

// get a copy of the content
Model rootModel = root.getContent();

// manipulate the copy
rootModel.addStatement(
    URIUtils.createURI("http://semversion.ontoware.org"),
    RDFS.label,
    "rdf-based versioning tool");
```

```
// commit the new version as a suggestion to the root versions
root.commit(rootModel, "version2", true);
```

Calculating Diffs

```
Version previousMainVersion = recentMainVersion.getFirstParent();

// get real model content of both versions (actually a copy of it)
Model recentMainModel = recentMainVersion.getContent();
Model previousMainModel = previousMainVersion.getContent();

// calculate diff between the models
Diff diff = semVersion.getSemanticDiff(recentMainModel,
                                       previousMainModel);

// Print out the number of added and removed Statements
System.out.println("Added: " + diff.getAdded().size());
System.out.println("Removed: " + diff.getRemoved().size());
```

5.1.2 Administration

Creating a user

```
// prepare server and create users
SemVersion semVersion = new SemVersion();
semVersion.createUser("admin", "password");
semVersion.createUser("tom", "password");
semVersion.createUser("joe", "password");
```

Create a versioned model Here an administrator creates a „VersionedModel” for the Gene Ontology.

```
// prepare versioned model
Session adminSession = semVersion.login("admin", "password");
adminSession.createVersionedModel(
    URIUtils.createURI("vm://1"),    // URI
    "Gene Ontology",                 // label
    new Date(),                       // valid from now on
    ValidTime.NOW);                  // valid forever
adminSession.close();
```

5.1.3 Usage and Implementation Notes

`URIUtils` is a simple helper class that creates a URI without declaring the throwing of an exception. If the string is not a valid URI, the method `createURI` will throw an (undeclared) `RuntimeException`.

Blank node enrichment is integrated into the model layer. `SemVersion` wraps all `rdf2go.Model` instances in a `SessionModel`. These perform blank node enrichment automatically. Note that currently the blank node identifiers are *not* used in diff calculation (yet)

5.1.4 SemVersion Usage Examples

How can `SemVersion` be used to solve the problems outlined in the requirements chapter 2? We present versioning for the `MarcOnt` scenario and briefly explain what can be done with the Gene Ontology.

Versioning for MarcOnt `SemVersion` can manage different branches of versions. Suggestions to the main branch are modelled as different branches, which can evolve separately. Snapshots of the main ontology with suggestions applied are created realised by merging the different branches and showing the user the merged version. Mappings between different versions can be stored as metadata of the version for which the backward-mapping is required. As every version can be identified by an URL, it is easy to discuss them, e. g. reference them in a forum. As URLs are also URIs one can also express arbitrary statements about them in RDF.

Versioning the Gene Ontology This will be the most exciting part of `SemVersion`'s near future. Until now, the exact queries to ask are not known and even the data set is not prepared. `SemVersion` now has a solid data storage, which will hopefully enable us to study the Gene Ontology as we wish.

Chapter 6

Extending SemVersion

In this section we explain how SemVersion can be used to build an ontology versioning system for a particular ontology language.

- The first step to take is to choose an **RDF encoding** for the ontology language. This should be possible for all ontology languages. In fact, for many languages an RDF encoding is already specified (e.g. OWL, OBOL and Topic Maps).
- We can **reuse** the complete version data management infrastructure of SemVersion, that includes managing projects, versioned models, versions and meta-data for each of these concepts. Some basic versioning functions can also be used out-of-the box such as retrieve, commit and branch. To conclude, no further effort is needed to reuse this part of the software. However, if the user needs a Java API for domain specific meta data, such an API can easily be generated from an RDF Schema with RDFReactor (c.f. 3.6).
- The only ontology language specific function of SemVersion is the **semantic diff**.

6.1 Package and Directory Structure

The packages of SemVersion in `/src` are

org.ontoware.semversion public API. This package contains the main facade class `SemVersion` from which the most important class `Session` can be obtained. Additionally, this package has exceptions (`BranchlabelAlreadyUsedException`, `CommitConflictException`, `LoginException`), time related helpers (`TransactionTime`, `ValidTime`), data containers (`Diff`, `Change`) and core API parts (`Version` and `VersionedModel` which both inherit from `VersionedItem`).

org.ontoware.semversion.dev is used only at development time to generate the RDF access classes.

org.ontoware.semversion.example contains a simple usage example for SemVersion.

org.ontoware.semversion.impl contains generated (and tweaked) RDF data accessors and manually coded implementations of the public API.

In `/test-src` you find JUnit¹ test cases for automated testing. The test cases also serve as a very precise documentation for the API.

6.2 Extending SemVersion with RDF Schema Support

RDF encoding RDF Schema naturally uses an RDF encoding, so this step requires no additional work. The same would be true for OWL.

Semantic Diff We must create an algorithm which computes the semantic diff of two data sets under RDFS semantics.

Implementing a new semantic diff in SemVersion requires developers to implement only one method in the class `org.ontoware.semversion.SemanticDiffEngine`. Currently, an implementation for RDFS is available, relying on Jena's inferencing abilities.

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

¹<http://www.junit.org>

Chapter 7

Conclusions and Outlook

In this deliverable, we described the design and implementation of the RDF-based versioning system *SemVersion*, which is released under LGPL as an open source project¹. The design of SemVersion emphasises a separation of re-usable, language agnostic features from ontology language specific features.

Five versioning uses cases (page 4) are described and requirements are deduced. Storing versions is delegated to existing RDF stores. This allows to pose queries across all stored versions. SemVersion allows for a rich set of metadata per model, e.g. provenance, author, valid time, transaction time, last editor, time stamp of last change, previous version.

SemVersion supports collaborative ontology development with the built-in ability to manage *suggestions* for each version. These suggestions can be turned into official versions.

Different from classical source code versioning, versioning ontologies requires the computation of a *semantic diff*, indicating changes on a conceptual, not a syntactic level. Blank nodes make diff computation harder. A solution, *blank node enrichment*, is presented. Diffs can be returned between arbitrary versions. We supply initial support for semantic diffs in the RDF Schema ontology language.

The API for dealing with versioning metadata is quite complex and cumbersome to program. Thus we developed a tool, RDFReactor², which *generates* a Java API from an RDF Schema.

For storage, SemVersion re-uses existing triple stores. To remain independent of a particular implementation, we developed a wrapper around common RDF management APIs: RDF₂Go (page A). Both tools are released separately and simplified implementation of SemVersion significantly; they deal with the lower-level data management parts. The SemVersion code base contains mainly code implementing

¹<http://semversion.ontoware.org>

²<http://rdfreactor.ontoware.org>

versioning processes like commit, merge, branch and diff.

Outlook In the future, we assume specialised RDF stores become more mature and allow more efficient query answering over large quad models. SemVersion is ready to change, as all RDF calls are channelled through RDF₂Go.

Additionally, we expect further extensions of the metadata managed for each version. If this is required, we have to manually extend the RDF Schema and then generate an enhanced API with RDFReactor.

Bibliography

- [1] Winkler, W., Völkel, M., Sure, Y., Schickel-Zuber, V., Binder, W., Tzouvaras, V., Ponte, D., Zini, C., Cruciani, M., Bonifacio, M., Kruk, S.R., Synak, M.: D2.3.1 specification of a methodology for syntactic and semantic versioning. Technical report, Knowledge Weg (NoE) (2004)
- [2] Berliner, B.: CVS II: Parallelizing software development. In: Proceedings of the USENIX Winter 1990 Technical Conference, Berkeley, CA, USENIX Association (1990) 341–352
- [3] Kruk, S.R.: Marcont initiative. Technical report, DERI.Galway, Ireland, <http://www.marcont.org/> (2004) Bibliographic description and related tools utilising Semantic Web technologies.
- [4] Kruk, S.R., Synak, M.: Jeromedl - e-library with semantics. Technical report, DERI.NUIG - Ireland; Gdansk University of Technology - Poland, <http://www.jeromedl.org/> (2004)
- [5] Zhdanova, A., Krummenacher, R., Henke, J., Fensel, D.: Community-driven ontology management: Deri case study. In: Proc of the IEEE/WIC/ACM International Conference on Web Intelligence, Compiègne, France, IEEE Computer Society Press (2005)
- [6] Bada, M., Stevens, R., Goble, C.A., Gil, Y., Ashburner, M., Blake, J.A., Cherry, J.M., Harris, M.A., Lewis, S.: A short study on the success of the gene ontology. *J. of Web Sem* 1(2) (2004) 235–240
- [7] Ashburner, M., Ball, C.A., Blake, J.A., Butler, H., Cherry, J.M., Corradi, J., Dolinski, K., Eppig, J.T., Harris, M., Hill, D.P., Lewis, S., Marshall, B., Mungall, C., Reiser, L., Rhee, S., Richardson, J.E., Richter, J., Ringwald, M., Rubin, G.M., Sherlock, G., Yoon, J.: Creating the gene ontology resource: design and implementation. *Genome Research* **11** (2001) 1425–1433
- [8] Stevens, R., Wroe, C., Lord, P., Goble, C. In: *Ontologies in bioinformatics*. Springer (2003) 635–657

- [9] Völkel, M., Krötzsch, M., Vrandečić, D., Haller, H.: Semantic wikipedia. In: Proceedings of the 15th international conference on World Wide Web, WWW 2006, Edinburgh, Scotland, May 23-26, 2006. (2006)
- [10] Harth, A., Decker, S.: Yet Another RDF Store: Perfect Index Structures for Storing Semantic Web Data With Contexts. Submitted (2005)
- [11] Kiryakov, A., Simov, K., Ognyanov, D.: Ontology middleware: Analysis and design. Technical report, IST Project IST-1999-10132 On-To-Knowledge (2002)
- [12] Sintek, M., Decker, S.: Triple - a query, inference, and transformation language for the semantic web. In: ISWC '02: Proceedings of the First International Semantic Web Conference on The Semantic Web, London, UK, Springer-Verlag (2002) 364–378
- [13] Hayes, P.: Rdf semantics. Recommendation, W3C (2004)
- [14] Völkel, M., Enguix, C.F., Kruk, S.R., Zhdanova, A.V., Stevens, R., Sure, Y.: Semversion - versioning rdf and ontologies. KnowledgeWeb Deliverable D2.3.3.v1, Institute AIFB, University of Karlsruhe (2005)
- [15] Carroll, J.J., Bizer, C., Hayes, P., Stickler, P.: Named graphs, provenance and trust. Technical report, HP (2004)
- [16] Carroll, J., Stickler, P.: Trix: Rdf triples in xml. Technical report, HP, Nokia (2004)
- [17] Völkel, M., Sure, Y.: Rdfreactor - from ontologies to programmatic data access. Poster and Demo at International Semantic Web Conference (ISWC) 2005, Galway, Ireland (2005)

Appendix A

RDF₂Go

As RDF₂Go is exposed to SemVersion users as the API to manipulate RDF models, we describe it at some depth here.

RDF₂Go¹ is a wrapper over RDF triple and quad stores. It provides a software layer to connect a Java application with some of the most popular Java frameworks for the Resource Description Framework (RDF)². The basic idea is depicted in Figure A.1.

The current version 1.0 offers support for the triple store Jena³ 2.2 and the quad stores YARS⁴ ref. 1217 [10], NG4J⁵ V0.4 (which builds on Jena), and Sesame⁶.

A.1 What is RDF₂Go?

RDF₂Go is a lightweight adapter framework between existing RDF triple and quad stores and Java Applications. While there are many implementations of the Resource Description Framework in Java, each of them has its pros and cons and it's difficult to choose the right one for your purposes among them. Using RDF₂Go it's easy to change the underlying triple or quad store without major effects for your application. Java applications may use the RDF₂Go API to remove compile-time and run-time dependencies on any particular RDF implementation.

Note: Currently, RDF₂Go has not achieved this goal, as the instantiation and configuration of triple stores has still to be done manually. Full compile-time in-

¹<http://rdf2go.ontoware.org>

²<http://www.w3.org/RDF/>

³<http://jena.sourceforge.net>

⁴<http://sw.deri.org/wiki/YARS>

⁵<http://www.wiwiss.fu-berlin.de/suhl/bizer/ng4j/>

⁶<http://www.openrdf.org>

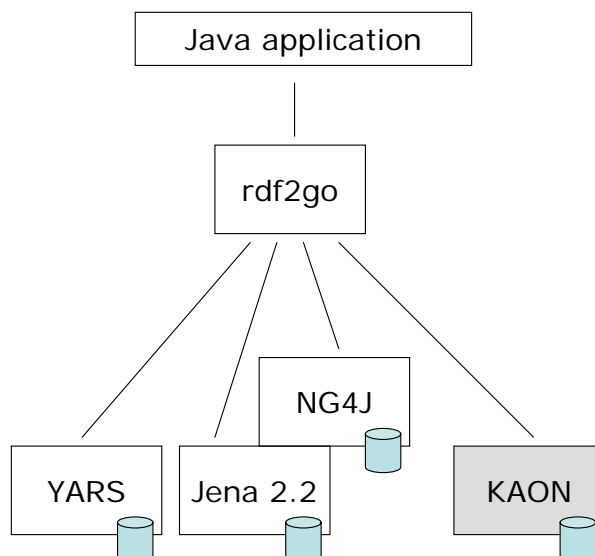


Figure A.1: RDF₂Go and existing RDF triple (and quad) stores

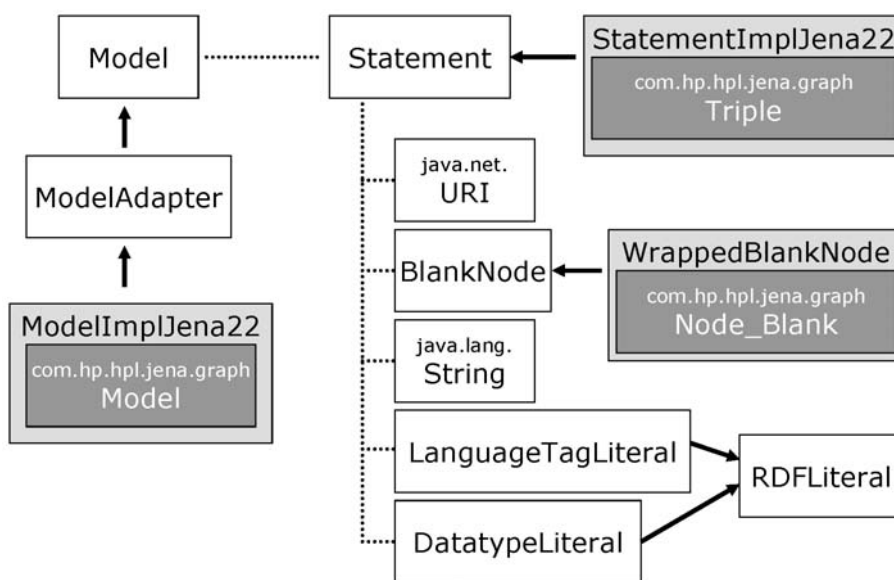


Figure A.2: RDF₂Go Type System and the Jena adapter classes as an example

dependence can only be achieved via reflection. Then configuration would have to be done through config files. This is not easy, as the set up of triple stores is very heterogenous, e. g. some need a data base connection, other a storage directory, and so on.

A similar project has been created by the Apache Software Foundation and is known as „jakarta commons logging⁷”.

RDF₂Go is so easy to use, that it might even be used in courses like „Semantic Web for Java Developers”. It’s main goals though, have been flexibility and ease-of-use.

A.2 Working Example: Simple FOAF via RDF₂Go

Imagine you want to write your own FOAF⁸ file using RDF₂Go. Here we provide a simple example how to do this.

Because all RDF frameworks use different configuration settings when constructing a model, it’s necessary for RDF₂Go to use different constructors. In each of the `impl`-Packages for the RDF stores you can find a `Model` implementation.

When we want to instantiate a new RDF₂Go model using Jena as the underlying RDF framework, we have to start with the following line of code. Right now we don’t want any inferencing, so we put it off.

```
// set up underlying triple store
com.hp.hpl.jena.rdf.model.Model jenaModelPlain =
    ModelFactory.createDefaultModel();
com.hp.hpl.jena.rdf.model.Model jenaModelRDFS =
    ModelFactory.createRDFSModel(jenaModelPlain);

// wrap model as RDF2Go model
Model model = new ModelImplJena22(jenaModelRDFS);
```

We want to state something about persons and relationships between them using the FOAF vocabulary. The next step is creating some URIs from this vocabulary, so building statements with them later is much easier.

```
URI foafName      = URIUtils.createURI("http://xmlns.com/foaf/0.1/name");
URI foafPerson    = URIUtils.createURI("http://xmlns.com/foaf/0.1/Person");
URI foafTitle     = URIUtils.createURI("http://xmlns.com/foaf/0.1/title");
URI foafKnows     = URIUtils.createURI("http://xmlns.com/foaf/0.1/knows");
URI foafHomepage  = URIUtils.createURI("http://xmlns.com/foaf/0.1/homepage");
```

⁷<http://jakarta.apache.org/commons/logging/>

⁸<http://www.foaf-project.org/>

With those URIs we now can start to state something about a person. While we don't have an URI for a person, we use a blank node (this is always done this way in FOAF).

```
BlankNode werner = model.getNewBlankNode();
```

First we say some things about Werner - where his homepage can be found, what his full name is and that he is a foaf:Person.

```
// N-TRIPLES Syntax:
//   _:blankNodeWerner
//       <http://xmlns.com/foaf/0.1/homepage>
//       <http://www.blue-agents.com> .
model.addStatement(
    werner,
    foafHomepage,
    URIUtils.createURI("http://www.blue-agents.com"));

model.addStatement(werner, foafTitle, "Mr");
model.addStatement(werner, foafName, "Werner Thiemann");
model.addStatement(werner, RDF.type, foafPerson);
```

Then we do similar things with Max.

```
BlankNode max = model.getNewBlankNode();
model.addStatement(max, RDF.type, foafPerson);
model.addStatement(max, foafName, "Max Völkel");
model.addStatement(max, RDFS.seeAlso,
    URIUtils.createURI("http://www.xam.de/foaf.rdf.xml"));
```

Now that we have introduced two persons, we can state that they know each other. We do this by using the foaf:knows property.

```
model.addStatement(werner, foafKnows, max);
```

While we added a lot of statement to the model by now, we don't know yet how we get this information back from the model. Let's assume, we want to list all instances of foaf:Person. Therefore we use a wildcard for the subject (i.e. Variable.ANY).

```
Iterator<Statement> it =
    model.getStatement(Variable.ANY, RDF.type, foafPerson);
```


We get back an `Iterator` over `Statements`. Finally we iterate over the query results and print the persons URI (i.e. a blank node) and the name of the person to standard out. To do this we have to do another query. This time we want any object for the found person that have the property `foaf:name`. Subject, predicate and object of a statement provided by the iterator can be accessed via get-methods.

```
while (it.hasNext()) {
    Object person = it.next().getSubject();
    System.out.println(person + " is a person");

    // get foaf:name
    Iterator<Statement> it2 = model.getStatement(person, foafName,
        Variable.ANY);
    while (it2.hasNext()) {
        System.out.println(person + " has the foaf:name "
            + it2.next().getObject());
    }
}
```

The full example can also be found in the package `org.ontoware.rdf2go.example`.

A.3 Architecture

Figure A.1 shows how RDF₂Go interacts with common Semantic Frameworks and RDF stores like Yars, Jena , NG4J or KAON⁹. Those underlying frameworks become transparent for the application, which only communicates with the interfaces provided by RDF₂Go.

The main package `org.ontoware.rdf2go` provides all those interfaces the application developer might need to manipulate and query RDF data.

The `org.ontoware.rdf2go.impl` provides implementations of RDF specifics, which don't exist in Java and are independent of the underlying framework. Classes herein also implement an adapter between `ContextModel` and `Model` and vice versa and also cope with URI handling. The methods of those classes simplify the implementations of the adapters for the underlying frameworks

All other packages implement specific adapter classes to communicate with the underlying RDF store.

While providing adapters for the most widely used triple and quad stores, any contributors may find it easy to write implementations for the RDF framework of their choice due to the simple API.

⁹<http://kaon.semanticweb.org/>

As RDF₂Go strives to be a unifying API, it cannot make reasonable assumptions about the nature of the underlying API's exceptions. Right now best way to deal with these exceptions seems to be to throw them as a generic type `Exception` to the RDF₂Go user. Hence most methods have a `throws java.lang.Exception` clause. In future versions RDF₂Go might have its own exception hierarchy which catches the exceptions of the underlying layer and return its own exception objects instead.

A.4 The API

`Model` and `ContextModel` are the main interfaces.

`Model` represent an RDF triple model while a `ContextModel` represents a quad model, also sometimes called *Named Graphs* or *Triples with Context* (hence the name).

The query results have to be in a form which easily can be accessed by Java applications. Therefore the classes `Statement` and `ContextStatement` provide a simple access to the RDF statements.

Model and ContextModel

We already saw how to create a model in the working example. For the sake of completeness we list all existing Model creation possibilities here.

```
//Jena without inferencing
Model model = new ModelImplJena22(false);
//Jena with RDFS inferencing
Model model = new ModelImplJena22(true);
//Yars
Model model = new ModelImplYars();
//NG4J
Model model = new ModelImplNG4J();
```

The methods can be divided into the following groups:

Model Manipulation

- `addStatement(Object subject, URI predicate, Object object)`
- `addStatement(Object subject, URI predicate, String literal, String language-Tag)`
- `addStatement(Object subject, URI predicate, String literal, URI datatype-URI)`

- `addAll(Model other)`
- `removeStatement(Object subject, URI predicate, Object object)`
- `removeStatement(Object subject, URI predicate, String literal, String languageTag)`
- `removeStatement(Object subject, URI predicate, String literal, URI datatype-URI)`
- `removeStatement(Statement statement)`
- `getNewBlankNode()`

The add and remove statements are straightforward. `getNewBlankNode()` provides an overall class for blank node treatment, because the RDF stores cope differently with those.

Model Querying and Existence Checks

- `query(String queryString)`
- `getStatement(Object subject, Object predicate, Object object)`
- `getStatement(Object subject, Object predicate, Object literalValue, Object literalAttribute)`
- `getStatements()`
- `contains(Object subject, Object predicate, Object object)`
- `contains(Statement statement)`

The query and `getStatement` concepts are explained in depth in A.4.1). The `contains`-methods provide a simple way to check if a statement exists in the current model.

Debugging Support RDF₂Go uses Apache Jakarta Commons Logging¹⁰. The underlying logging implementation used is log4j¹¹. The configuration for the logging can be found in `/src/log4j.properties`.

Additionally the following methods are provided by the `Model`.

- `size()`

¹⁰<http://jakarta.apache.org/commons/logging/>

¹¹<http://logging.apache.org/log4j/docs/>

- `dump()`
- `getUnderlyingModelImplementation()`

While `size()` gives you an idea how many statements can be found in a model, the `dump()` method prints the whole content to the logger instance of the implementation. For some special purposes it might be necessary to get the model of the underlying layer. This is provided by `getUnderlyingModelImplementation()`.

Another rather unintuitive feature at first sight is the ability to store object references at runtime in the model. They enable the model to act as a central facade for all kinds of usage. This feature was introduced to be used in RDFReactor.

- `setProperty(Uri propertyUri, Object value)`
- `getProperty(Uri propertyUri)`

Type System One of the central aspects of RDF₂Go is to map the RDF type system to the Java type system. The mapping is simple and should be intuitive for regular Java developers. It goes as follows:

URI is mapped to `java.net.Uri`.

Plain Literal is mapped to `java.lang.String`.

Literal with a Language Tag is mapped to `org.ontoware.rdf2go.LanguageTagLiteral` which has two methods: `public String getValue()` and `public String getLanguageTag()`.

Literal with a Datatype URI is mapped to `org.ontoware.rdf2go.DatatypeLiteral` which also has two methods: `public String getValue()` and `public Uri getDatatype()`.

Blank Node has only the semantics of being either the same (equal) or not the same as another blank node. In RDF₂Go this is mapped to the marker interface `org.ontoware.rdf2go.BlankNode`. The `equals`-method should work correctly.

Variables are used only in queries. RDF₂Go maps wildcards which can be used in triple (or quad) search patterns to instances of `org.ontoware.rdf2go.Variable`. As there exists only one wildcard there is only one instance: `org.ontoware.rdf2go.Variable.ANY`.

Figure A.2 shows the type system and the RDF₂Go adapter classes for Jena.

A.4.1 Queries

RDF₂Go offers two ways to query a model. Both return an `Iterator<Statement>`. This is a new feature of Java 5.0 called „generics”. It basically ensures that each object returned by `next()` is of type `Statement`. Iterators as query result have the advantage of low memory consumption¹².

Queries can be plain text, which is interpreted by the underlying triple or quad store implementation. This offers flexibility, until a standard query language for RDF emerges.

The second query option has less expressivity but clearly defined semantics. It uses only triple (or quad) search patterns. A search pattern has for each section of a triple – namely subject, predicate and object (and context) – either a concrete value or a wildcard. The iterator returns all elements which fulfill the given search pattern.

A.5 How to get started

RDF₂Go is simple to install and simple to use.

RDF₂Go can be **downloaded** from ontoware.org¹³. Right now it comes in 4 flavors. Pure, with Yars, with Jena or with NG4J. There is also a developers CVS, which can be found at ontoware.org¹⁴

RDF₂Go is released under the **GNU Lesser General Public License (LGPL)**, Version 2.1, Feb. 1999. We reserve the right to release RDF₂Go in parallel under different licenses.

For **support**, please feel free to post to the forum at ontoware.org¹⁵ – we will respond quickly and your feedback is very welcome.

¹²This idea was provided by Andreas Harth

¹³http://ontoware.org/frs/?group_id=37

¹⁴http://ontoware.org/scm/?group_id=37

¹⁵http://ontoware.org/forum/forum.php?forum_id=143

Appendix B

SemVersion Schema

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix : <http://semversion.ontoware.org/ns/2005#>.
@prefix reactor: <http://rdfreactor.ontoware.org/2005/04#>.
@prefix xsd: <http://www.w3c.org/2001/XMLSchema#>.

:Root a rdfs:Class.

:hasVersionedModel a rdf:Property
; rdfs:domain :Root
; rdfs:range :VersionedModel
.

:VersionedItem a rdfs:Class
; rdfs:comment "valid time, transaction time and core metadata"
.

:Diff a rdfs:Class
; rdfs:comment "contains added and removed statements,
                each modeled as 'model'"
.

:hasAdded a rdf:Property
; rdfs:domain : Diff
; rdfs:range :Model
.

:hasRemoved a rdf:Property
```

```
; rdfs:domain : Diff
; rdfs:range :Model
.

:hasTag a rdf:Property
; rdfs:domain :VersionedItem
; rdfs:range rdfs:Literal
; a reactor:SingleValueProperty
.

:hasAuthor a rdf:Property
; rdfs:domain :VersionedItem
; rdfs:range :User
; a reactor:SingleValueProperty
.

:hasCreationTime a rdf:Property
; rdfs:domain :VersionedItem
; rdfs:range xsd:long
; a reactor:SingleValueProperty
.

:hasDeletionTime a rdf:Property
; rdfs:domain :VersionedItem
; rdfs:range xsd:long
; a reactor:SingleValueProperty
.

:hasValidTimeStart a rdf:Property
; rdfs:domain :VersionedItem
; rdfs:range xsd:long
; a reactor:SingleValueProperty
.

:hasValidTimeEnd a rdf:Property
; rdfs:domain :VersionedItem
; rdfs:range xsd:long
; a reactor:SingleValueProperty
.

:hasUserdefinedMetadata a rdf:Property
; rdfs:domain :VersionedItem
; rdfs:range :TripleSet
```

```
; a reactor:SingleValueProperty
.

:hasProvenance a rdf:Property
; rdfs:domain :VersionedItem
; rdfs:range xsd:anyURI
; a reactor:SingleValueProperty
.

:VersionedModel rdfs:subClassOf :VersionedItem
; rdfs:comment "A tree of Versions"
.

:hasRoot a rdf:Property
; rdfs:comment "has only one root"
; rdfs:domain :VersionedModel
; rdfs:range :Version
; a reactor:SingleValueProperty
.

:hasVersion a rdf:Property
; rdfs:comment "exhaustive list of all versions"
; rdfs:domain :VersionedModel
; rdfs:range :Version
.

:Version rdfs:subClassOf :VersionedItem
; rdfs:comment "A set of triples (content TripleSet)
               with a set of metadata"
.

:hasContent a rdf:Property
; rdfs:domain :Version
; rdfs:range :TripleSet
; a reactor:SingleValueProperty
.

:hasFirstParent a rdf:Property
; rdfs:comment "A version has two parents only when merged"
; rdfs:domain :Version
; rdfs:range :Version
; a reactor:SingleValueProperty
```



```
# ; reactor:inverseOf :hasChildren
.

:hasChild a rdf:Property
; rdfs:comment "inverse of firstParent and secondParent,
               relation maintained manually"
; rdfs:domain :Version
; rdfs:range :Version
.

:hasProvenance a rdf:Property
; rdfs:domain :Version
; rdfs:range rdf:Resource
; a reactor:SingleValueProperty
.

:hasSecondParent a rdf:Property
; rdfs:domain :Version
; rdfs:range :Version
; a reactor:SingleValueProperty
# ; reactor:inverseOf :hasChildren
.

:hasContainer a rdf:Property
; rdfs:domain :Version
; rdfs:range :VersionedModel
; a reactor:SingleValueProperty
.

:Model a rdfs:Class
; rdfs:comment "models the actual triples in the model.
               Probably using rdf2go.ontoware.org"
.

:User a rdfs:Class
# ; rdfs:subClassOf foaf:Person
; rdfs:comment "@todo"
.

:hasName a rdf:Property
; rdfs:domain :User
```

```
; rdfs:range rdfs:Literal
; a reactor:SingleValueProperty
.
```

```
:hasPassword a rdf:Property
; rdfs:domain :User
; rdfs:range rdfs:Literal
; a reactor:SingleValueProperty
.
```

```
:branchLabel a rdf:Property
; rdfs:domain :Version
; rdfs:range rdfs:Literal
.
```