D2.3.3.v1 SemVersion – Versioning RDF and Ontologies

Max Völkel (University of Karlsruhe)

with contributions from:
Carlos F. Enguix (National University of Ireland, Galway, Ireland)
Sebastian Ryszard Kruk (DERI)
Anna V. Zhdanova (DERI)
Robert Stevens (U Manchester)
York Sure (AIFB)

Abstract.
EU-IST Network of Excellence (NoE) IST-2004-507482 KWEB
Deliverable D2.3.3.v1 (WP2.3)

This papers describes the requirements for a semantic versioning system. The design, implementation and usage of SemVersion are described.

Document Identifier	KWEB/2004/D2.3.3.a/v1.0
Project	KWEB EU-IST-2004-507482
Version	v1.0
Date	June 6th, 2005
State	final
Distribution	internal

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 Polythechnique 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 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 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 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 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

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 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

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. 1.2.3). 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*. 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.

The requirements for our system are derived from a set of practical scenarios, which are documented in detail in this deliverable.

The project experienced a shift in requirements, when Robert Stevens from University of Manchester joined the group in May 2005. WP 2.3 decided to tackle the problem of versioning the Gene Ontology.

In [1] we suggested reification for data storage. As we now face the large volume of the Gene Ontology data (see 1.2.3), we need more powerful storage solutions than for the other use cases. Addressing triple sets (models) is another challenge. In [1] we argued to use reification, which would make models four times as large. To avoid this, we now use native quad stores, which provide a context URI for each triple. We use the context URI to address models more efficiently.

A sub-project, Rdf2Go, has been created to deal with various model abstractions and serves as a unifying triple (and quad) store entry point. Rdf2Go is described in Chapter 2.

A second sub-project of SemVersion, RdfReactor, facilitates the usage of RDF Schema based data in Java significantly. It's latest version is based on Rdf2Go. In fact, RDFReactor has been designed for SemVersion in the first place. RDFReactor is described in Sec. 1.5.4.

Contents

Chapter 1

SemVersion – An RDF Versioning System

1.1 Introduction

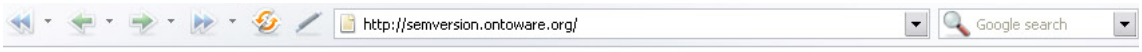
As outlined in the Knowledge Web Deliverable D2.3.1 „Specification of a methodology for syntactic and semantic versioning" [1], there is a clear need for RDF data and ontology versioning. This deliverable is a follow-up of D2.3.1, which explains the underlying concepts in detail. Here we focus on the 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. 1.2.3). 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.

[1]The 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.ja needed only for runnign the test cases)
- RDFReactor
- NamedGraphs4Jena (NG4J)

License

SemVersion is released under the GNU Lesser General Public License, Version 2.1, Feb. 1999.

AIFB

Figure 1.1: Homepage of the SemVersion project

Our approach is inspired by the classical CVS system for version management of textual documents (e.g. Java code). Core element of our approach is the separation of language-specific features (the *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.

1.1.1 Term Definitions

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**). A set of triples is called **model** (or triple set). An **ontology** is a model, in which semantics have been assigned to certain URIs and/or triple constructs, according to an **ontology language**. 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**.

SemVersion versions models. A model under version control is named a **versioned model**. A versioned model has a **root model**, which is a **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. 1.3.2.

1.2 Requirements for an ontology versioning system

We gathered different requirements from Knowledge Web partners in order to create a more general design. We tried to gather as 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 derived versioning requirements.

1.2.1 Use Case 1: MarcOnt Collaborative Ontology Development

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

The MarcOnt[2] scenario served as the first source of inspiration for SemVersion. MarcOnt is a project to create an ontology for library data exchange.

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, one has to look for particular keywords in the resource's description fields, but one cannot carry out a search be meaning or concept. This can often result in large sets of results. Also the data communication between library systems is very hard to extend. On 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 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 assumed 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 The MarcOnt project has a clear view on the process of ontology evolution. It starts with a current main version. Now people can suggest (multiple, independent) changes. Then the community discusses about the proposed changes and selects some. The changes are applied and a new main version is created. The process is illustrated in Fig. 1.2.

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)[3]
- Manage versions of suggestions (R1.2)
- Generate snapshots of the main ontology with some suggestions applied (R1.3)

[2] `http://www.marcont.org/`

[3] Requirements are numbered by "use case number" / "." / running number

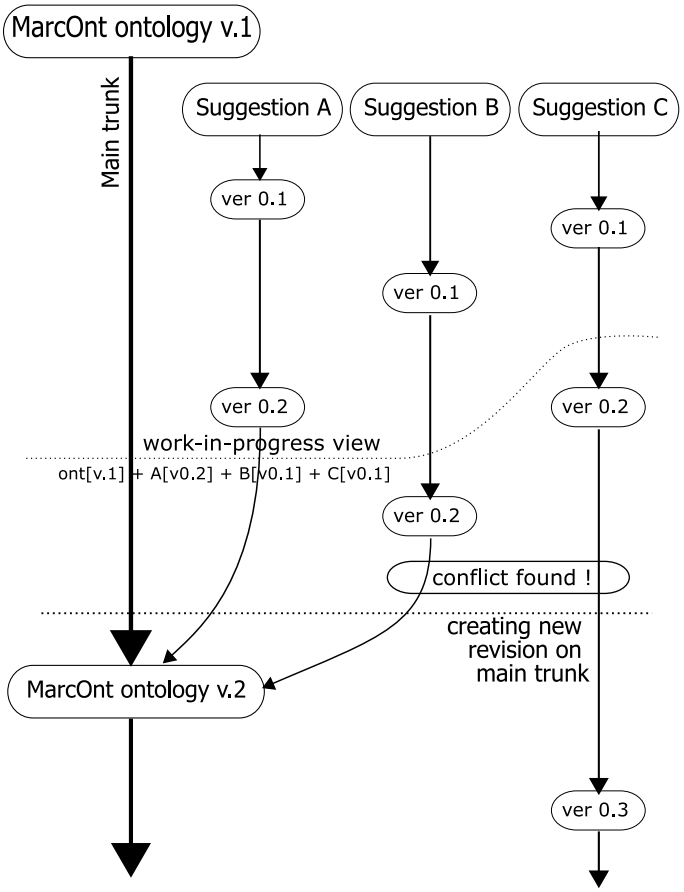


Figure 1.2: Versions and suggestions in the MarcOnt use case

- Detect and resolve conflicts (R1.4)
- Add suggestions to the main trunk (R1.5)
- Attach mapping/translation rules (R1.6)
- Be able to check out arbitrary versions by HTTP GET with a specific URL (R1.7)

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

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

People's portal [2] 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 marry 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)

1.2.3 Use Case 3: Versioning the Gene Ontology

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

Background An important step was the phone conference on 12.07.2005, in which common goals were identified[4]. Robert Stevens from Manchester University has become an active member of the work package. Robert is a biologist who is also a doctor in Computer Science. Robert is a Bioinformatics Lecturer in the BioHealth Informatics Group at the University of Manchester. He has around 80 publications in international conferences, workshops, journals and so on. He was involved in the TAMBIS project for transparent access and integration of biological databases. Now one of his main interests is in the definition of formal biological ontologies. He is involved in the transformation of the Gene Ontology controlled vocabulary into a description-logics OWL based ontology. He is interested in contributing to the development of an ontology-based versioning system to the Gene Ontology which is part of the Open Biological Ontologies. Also he want's to study how conceptualisations change over time, hence the need for data analysis.

Use case description The gene ontology[5] 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 far ahead of other communities constructing ontologies [3]. 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

[4] `http://sw.deri.org/wiki/KnowledgeWeb/WP23/MeetingAgenda12July2005`
[5] `http://www.geneontology.org`

protein roles in cells is accumulating and changing. GO provides three structured networks of defined terms to describe gene product attributes."[6]

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.

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*[7] (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 100.000 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 years × 12 month = 72 versions. Thus the underlying triple store must be able to handle up to 7 million triples and search (maybe even reason) over them.

The requirements in short form are thus

- Store up to 7 million triples (R3.1)
- Allow meta-data queries over the 72 versions (R3.2)
- Allow data queries over all versions (7 million triples) (R3.3)
- OBOL semantic diff (R3.4)
- OBOL to RDF converter (R3.5)
- A Java interface (R3.6)

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 previously 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.

The GO ontology permits to associate biological relationships among molecular functions, the involvement of molecular functions in biological processes and the

[6]Extracted from the OBO site `http://obo.sourceforge.net/`

[7]`http://obo.sourceforge.net/`

occurrence of biological processes at a given time and space in cells [4]. 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 20.000 concepts which are convertible to OWL. The latest statistics about the GO could be found at the GO site [8]:

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 [5] 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 reports[9]. 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. In order to use the data sets, one has to decide for a format. There are three options: (a) RDF, (b) OWL generated from DAG-Edit[10] or (c) nice OWL generated by Protégé-Plugin. Whatever choice is made, the exported data should contain the provenance

[8] `http://www.geneontology.org/GO.downloads.shtml#ont`
[9] `http://www.geneontology.org/MonthlyReports/`
[10] `http://www.godatabase.org/dev/java/dagedit/docs/index.html`

information of the source file and the conversion process used. SemVersion offers ways to store such provenance information.

1.2.4 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*[11] aims at creating a *semantic* 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"[12] pages are just a special kind of resource and some attached properties. Hence, a semantic diff has to be calculated „by hand".

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 SemWiki users need ways to request a semantic diff between two page-versions. As pages partly consist of „background statements", which do not belong to a particular page, SemWiki needs a model-based 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).

1.2.5 Use Case 5: Analysis of Wikipedia

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

An emerging research topic at AIFB is the analysis of changes in the Wikipedia[13]. This use case is mostly similar to „Versioning the Gene Ontology".

Data Set The Wikipedia contains roughly 1.500.000 articles across all language versions.

[11] http://semwiki.ontoware.org
[12] http://semwiki.ontoware.org
[13] http://www.wikipedia.org

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

1.2.6 Requirements Summary

We can distinguish rather data management related requirements and rather ontology language specific features.

Data Management Requirements

- Store and retrieve versions; **store up to 7 million triples**
- Retrieve versions via HTTP or Java function calls; address versions unambiguously via URIs and user-friendly via labels
- **Rich meta data per model**/statement: provenance, author, valid time, transaction time
- Model based versioning and additionally concept-oriented queries
- **Queries across versions concerning meta data**
- Each version can have a number of attached „suggestions"; ability turn suggestions into official versions

Ontology Language Requirements

- **Queries across versions concerning the content**
- return **diffs between arbitrary versions**
- OBOL semantic diff
- OBOL to RDF converter
- RDFS semantic diff
- OWL semantic diff
- Semantic Wiki semantic diff
- Conflict detection in OWL

1.3 Data Management Design

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. We first present the data management parts and then the ontology specific versioning functions.

The data management parts can be used no matter which ontology language is used – as long as the data model is encoded as RDF. RDF encoding of data is crucial in order to have a significant re-use of software across ontology languages. We now present some arguments for this claim. A more detailed discussion can be found in the Knowledge Web Deliverable D2.3.1 [1].

1.3.1 RDF 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).

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.

1.3.2 Version Data Management

The general idea is the re-use of 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*.

SemVersion's data model (depicted in Fig. 1.3) was basically derived from the requirements of the MarcOnt use case (Sec. 1.2.1) and fulfills also the other use cases needs. Only the rather statement-oriented versioning requirements from the use case 2 remains difficult to integrate.

SemVersion has a **repository** of **projects**. They can be created, listed and removed from the repository. A project can hold a number of **versioned models**. 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.

A **version** is the most central concept. It is a model decorated with all kinds of metadata. A version knows where it cames 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 by merging two existing models or commit a diff. Committing diffs is useful, if the models become really large and change only litte – 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

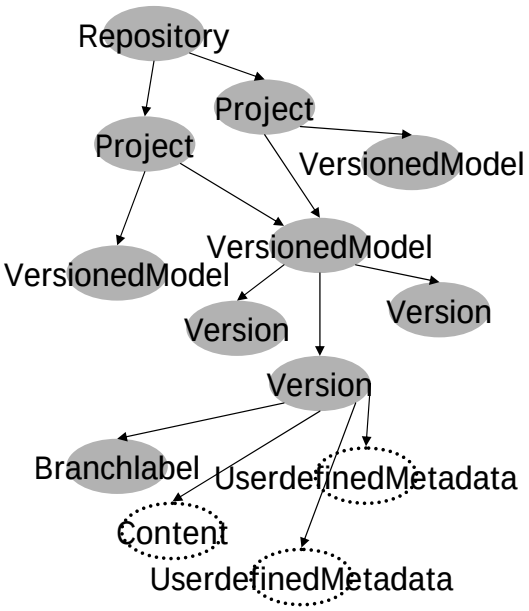


Figure 1.3: Data Model for RDF Versioning

engineering projects. Such projects are simple sets of versioned models and give the user a better ability to manage the different ontologies in progress.

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

1.4 Versioning Functionality Design

Versioning functionality deals with ontology language specific functionality such as the structural diff (ignoring semantics) and the semantic diff (depends on ontology language; uses structural diff). We also talk about blank nodes, which make the issue of comparing versions harder, but not impossible.

1.4.1 Structural Diff

Although the structural diff is the same for all ontology languages, we describe it here for sake of consistency. The structural diff is simply the set-theoretic difference of two RDF triple sets. Libraries such as Jena have built-in functions to compute

version A:	**version B:**

`a rdfs:type c`
`b rdfs:type c`
`c rdfs:subClassOf d`
	`a rdfs:type d`
	`b rdfs:type d`
	`c rdfs:subClassOf d`
	`e rdfs:type d`

added:	**removed:**

`a rdfs:type d`
`b rdfs:type d`
`e rdfs:type d`
	`a rdfs:type c`
	`b rdfs:type c`

Figure 1.4: Example for a Structural Diff

this set-difference. RDF2GO also offers a native implementation. An example for a structural diff can be found in Fig. 1.4.

The **diff function** $d(A, B) \rightarrow \langle a(A, B), r(A, B)\rangle$ is a non commutative function from two triple sets (A, B) to two triple sets of added ($a(A, B)$) and removed ($r(A, B)$) statements, with $a(A, B) = B - A = B \backslash (A \bigcap B)$ and $r(A, B) = A - B = A \backslash (A \bigcap B)$. Such diffs can be computed by simple set arithmetics for triple sets that contain only URIs and literals, as shown in [6]. Blank nodes cause some problems here, as discussed in Sec. 1.4.3.

1.4.2 Semantic Diff

The semantic difference has to take the semantics of the used ontology language into account. It is therefore not possible to write a generic algorithm for this. An intuitive way to understand the concept of a sematic diff goes like this: Let's assume we have RDF Schema as our ontology language. Further we have two models A and B, which express two versions of an RDF Schema. Now, in order to compute the semantic 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 diff on $Inf(A)$ and $Inf(B)$. This not the same as the structural diff between model A and B. Fig. 1.5 illustrates the semantic diff under RDF Schema entailment semantics.

A possible way to compute a semantic diff is thus 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 doable, especially when the models grow really large. 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 [7]. Initially we provide support for RDF Schema. An extension to OBOL is planned.

If the structural diff of two models is empty, then the semantic diff must also

version A:	**version B:**

`a rdfs:type c`
`b rdfs:type c`
`c rdfs:subClassOf d`
	`a rdfs:type d`
	`b rdfs:type d`
	`c rdfs:subClassOf d`
	`e rdfs:type d`

added:	**removed:**
`e rdfs:type d`	

Figure 1.5: Example for a Semantic Diff under RDFS semantics

be empty. The inverse is not necessarily true: There might be two different RDF models which encode the same semantic model.

1.4.3 Blank Nodes and the Diff

Blank nodes[14] cause some problems in computing the structural diff, as we have no knowledge about the relation (equal or not?) between two blank nodes form different models. The RDF semantics dictate to treat them as different. In a versioning context, this leads to the unwanted fact that the diff between a model and itself is not empty, if it contains blank nodes.

As a work-around we invented the concept of *blank node enrichment*, which attaches artificial inverse functional properties to every blank node. This changes nothing to the RDF semantics but helps to identify equal blank nodes across models.

Most RDF processing tools will leave this information intact. In the MarcOnt scenario (c. f. Sec. 1.2.1), 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.

However, if no blank node enrichment is present, we still have to offer a good versioning system. We can distinguish five cases of blank node differences across models:

- The blank node appears in exactly the same statements in both versions. Then it is semantically safe to assume blank node identity.
- 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.
- If the blank node has been extended monotously but in different ways, what does this mean? Are the blank node identifiers „_:1" and „_:3" referring to the same object? Has „Max" know two phone numbers or are there two Maxes

[14] Sometimes also called „bnodes" – but they have nothing to do with b-trees.

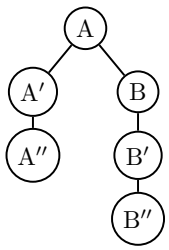


Figure 1.6: A sample version tree

know? In general, it seems better to treat the blank nodes as different in this example. Example:

version A:	**version B:**

`_:1 :hasName "Max"`	`_:3 :hasName "Max"`
	`_:3 :hasPhone "123"`
	`_:5 :hasName "Max"`
	`_:5 :hasPhone "456"`

added:	**removed:**

`_:3 :hasName "Max"`
`_:3 :hasPhone "123"`
`_:5 :hasName "Max"`
`_:5 :hasPhone "456"`
	`_:1 :hasName "Max"`

- Almost as ambiguous is the case when some properties have been removes and some have been added. Here again, no equality should be assumed.

- The original blank node could appear only in reduced and extended models. Here again, no equality should be assumed.

Unfortunately, blank nodes *are* used in practice. They are used in OWL for property restrictions and in FOAF to denote persons.

1.4.4 Branch and Merge

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

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

Merging two branches is different. First we look at the branch point c, which is defined as the most recent common version of the two branches. Such a version always exists, as branches can only be created by committing a version to an existing version. We also take two versions from the different branches, in most cases the most recent ones, and call them a and b. Consider the example version tree given in Fig. 1.6. Here $c = A, a = A'', b = B''$. In order to merge b back into a we compute the $diff(c, b)$ and apply it to a.

1.4.5 Conflict Detection

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

1.4.6 Query Language Extension

WP 2.3 proposed an extension to RDQL and SPARQL to enable the querying of versions through bi-temporal database features such as valid-time and transaction-time and context information. The proposal was merely practical and did not include the intended semantics associated to the query language extensions, which in fact triggered some debate in the room. Our viewpoint was from a database perspective considering that both RDQL and SPARQL are SQL-like query languages for semistructured/graph-based data. The point to be discussed is that SPARQL could simulate or include features available in SQL3 such as nested/correlated queries with the required closure of query results, include bi-temporal data such as found in temporal SQL, and possibly include procedural capabilities such as calls to external functions, use of surrogate methods, use of path expressions as in object oriented or graph databases and so on.

1.5 Implementation

Note: Different from the approach described in [1], we now aim at creating a pure Java library. As the java library is the core of such a versioning concentrating out efforts on the core gives us the opportunity for quicker feedback. Additional HTTP-based web services will be added as required.

The high-level architecture of the implementation consists of several layers (c. f. Fig. 1.7). Each layer depends on the layer below. To users only the SemVersion API and the RDF2Go API are exposed.

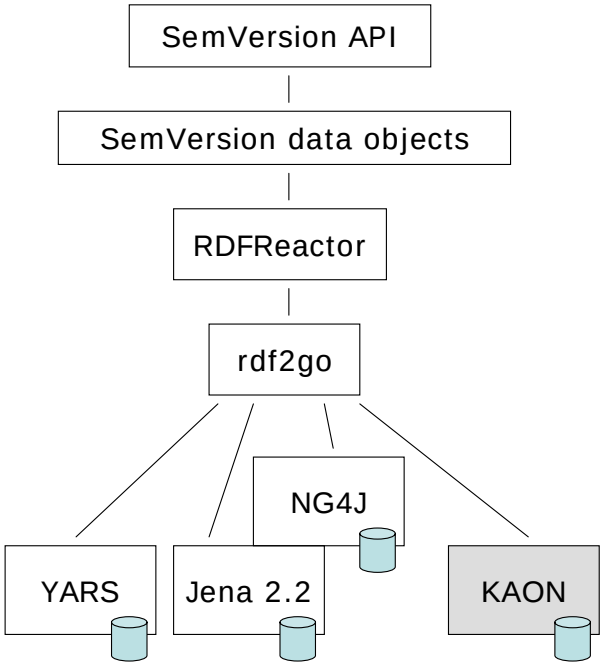


Figure 1.7: The Layered Architecture of SemVersion

- SemVersion API – which is described in Chapter 3
- SemVersion internal data structures – these are rich in functionality, but too complex as an external API
- RDFReactor – a framework for domain-specific object-oriented RDF access in Java
- Rdf2Go – an abstraction over triple (and quad) stores
- YARS – a scalable quad store

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

1.5.1 Storage Layer Access

An ontology versioning system should scale in many dimensions. It should allow a large number and size of ontologies. This implies a scalable storage architecture. If

the ontologies become large, it is undesirable to download them first and query or manipulate them locally. There already exist scalable RDF stores with remote query and update functionality. SemVersion utilized Rdf2Go, a sub-project of SemVersion which is described on page 22. It abstracts away the triple store implementation and gives the user a simple Java-centric API for model changes. The storage layer access is implemented in the class `TripleStore` which offers means to get models. The TripleStore uses a ContextModel for it's 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, SemVersion uses YARS [8] to store it's data.

1.5.2 Handling Commits

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

1.5.3 Generating globally unique URIs

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

1.5.4 RDFReactor

The general trade-off between the power of a strongly typed, object-oriented API and the flexibility of having direct access to the underlying data exists as well in the RDF and Java world. The open-source project RDFReactor[15], which generates data manipulation classes from an RDF Schema[16], is used to give the user an object-

[15] `http://RDFReactor.ontoware.org`
[16] `http://SemVersion.ontoware.org/2004/12/datamodel`

oriented access for many common functions like adding projects, setting the parents of a version or storing the branch label. Parallel access to the stored RDF data is always possible.

RDFReactor builds on RDF2GO. A longer paper about RDFReactor can be found at `http://xam.de/2005/05/rdfreactor.pdf`.

Until now, RDFReactor has been downloaded over 70 times.

Chapter 2

RDF2Go

This chapter exclusively describes RDF2Go. RDF2Go[1] is an abstraction 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 2.1.

The current version 1.0 offers support for the triple store Jena [3] 2.2 and the quad stores YARS [4] ref. 1217 and NG4J [5] V0.4 (which builds on Jena).

2.1 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 it's 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.

A similar project has been created by the Apache Software Foundation and is known as „ jakarta commons logging[6]".

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.

[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://jakarta.apache.org/commons/logging/

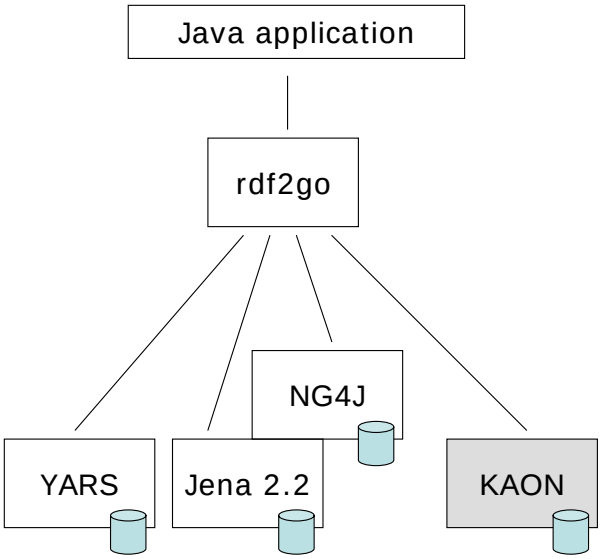


Figure 2.1: RDF2GO and existing RDF triple (and quad) stores

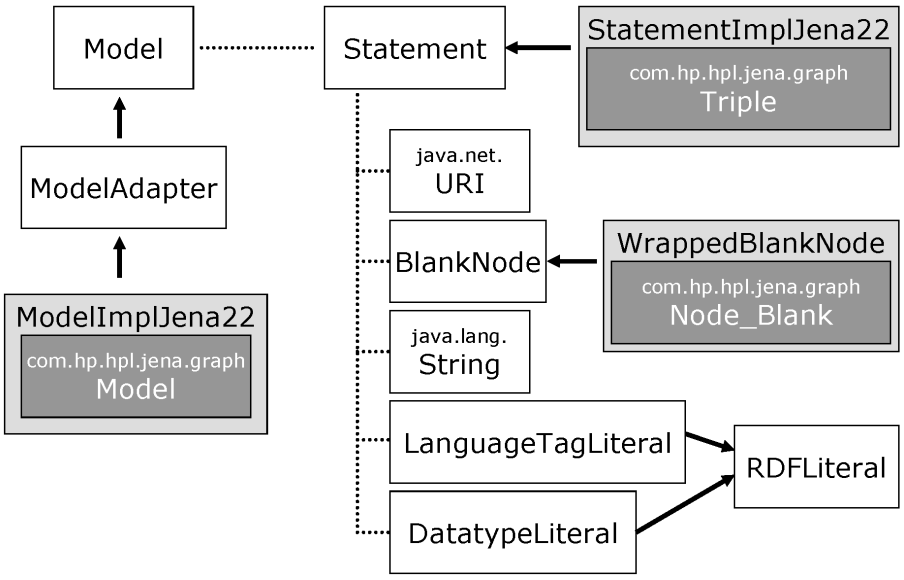


Figure 2.2: RDF2GO Type System and the Jena adapter classes as an example

2.2 Working Example: Simple FOAF via RDF2Go

Imagine you want to write your own FOAF[7] 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.

```
// no inferencing
Model model = new ModelImplJena22(false);
```

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");
```

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"));
```

[7] http://www.foaf-project.org/

```
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 yet don't know how we get this information back from the model. We will show a simple query methods to do so here. We want to list all persons. 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`.

2.3 Architecture

Figure 2.1 shows how RDF2Go interacts with common Semantic Frameworks and RDF stores like Yars, Jena , NG4J or KAON[8]. 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.

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.

2.4 The API

`Model` and `ContextModel` are the main interfaces for triple or quad store usage. They provide access to a RDF model.

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.

2.4.1 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.

[8] `http://kaon.semanticweb.org/`

```
//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 languageTag)
- addStatement(Object subject, URI predicate, String literal, URI datatypeURI)
- 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 datatypeURI)
- 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 2.4.2). The contains-methods provide a simple way to check if a statement exists in the current model.

Debugging Support

RDF2Go uses Apache Jakarta Commons Logging[9]. The underlying logging implementation used is log4j[10]. The configuration for the logging can be found in `/src/log4j.properties`.

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

- size()
- 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)

[9] `http://jakarta.apache.org/commons/logging/`
[10] `http://logging.apache.org/log4j/docs/`

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 `rdf2go.ontoware.org.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 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 2.2 shows the type system and the RDF2Go adapter classes for Jena.

2.4.2 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 consumption[11].

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

[11] This idea was provided by Andreas Harth

value or a wildcard. The iterator returns all elements which fulfill the given search pattern.

2.5 How to get started

Rdf2Go is simple to install and simple to use.

Rdf2Go can be **downloaded** from ontoware.org [12]. 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.org [13]

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.org [14] – we will respond quickly and your feedback is very welcome.

[12] `http://ontoware.org/frs/?group_id=37`
[13] `http://ontoware.org/scm/?group_id=37`
[14] `http://ontoware.org/forum/forum.php?forum_id=143`

Chapter 3

Using and Extending SemVersion

In this chapter we describe how developers can actually use SemVersion to solve their versioning tasks. In Sec. 3.2 we explain how SemVersion can be extended to handle other ontology languages.

3.1 Using SemVersion

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
- End session

3.1.1 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 verions
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());
```

3.1.2 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();
```

3.1.3 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)

3.1.4 SemVersion Usage Examples

How can SemVersion be used to solve he problems outlined in the requirements section 1.2? 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 about 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.

3.2 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 metadata for

each of these concepts. Some basic versioning functions can also be used out-of-the box such as retrieve, commit and branch.

The only language specific function of SemVersion is the **semantic diff**. Ontology language specific systems built on top of SemVersion have to change one line of code and provide an appropriate implementation of a `SemanticDiffEngine`. In the class `SemVersion` we have the method

```
 public Diff getSemanticDiff(Model model1, Model model2) {

    // TODO Adapt this line to other Ontology Languages
    SemanticDiffEngine sde = new RDFS_Diff(); // OBOL_Diff();

    TripleStore ts = svi.getTripleStore();
    try {
        return sde.getSemanticDiff(ts, model1, model2);
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

The SemanticDiffEngine has only one method and should thus be easy to implement.

```
public interface SemanticDiffEngine {

    public Diff getSemanticDiff(TripleStore ts, Model a, Model b)
            throws Exception;

}
```

SemVersion provides RDF Schema semantic diff. The usage of RDFS versioning will be further discussed in the deliverable D2.3.5.a „Integration of Consensus Making Environment with RDF versioning system".

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.

Chapter 4

Conclusions and Outlook

This deliverable describes the first version of SemVersion, a RDF and ontology versioning system.

The requirements (Sec. 1.2) changed when the Gene Ontology use case was integrated. We now face mainly scalability issues which caused quite some changes in the lower implementation layers. We expect the requirements to change even more, but we are confident, that the current design can handle most imaginable requirements without much implementation effort.

SemVersion can handle RDF versioning and allows easy extension for other ontology languages. The architecture is based on Rdf2Go and RdfReactor. Both choices helped to keep the programming flexible and fast. We can now e. g. exchange Yars by NG4J by changing a single line of source code. This offers a cost effective way to experiment with even more triple (or quad) stores, until a working solution is found.

The biggest challenge for SemVersion is scalable reasoning and we are looking forward to upcoming solutions. Luckily, the OBOL ontology language (used in the Gene Ontology) seems to have much simpler semantics then e. g. OWL.

We will build a general, extendable multi-language ontology versioning system, that will help research and industry to employ ontology based technologies in dynamic settings.

Next Steps The next challenges are:

- integration of use case 2
- using SemVersion for the Gene Ontology

In fact, WP 2.3 puts most energy into the use case 3, as this seems to be the most interesting use case with the greatest impact.

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] 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, Compiegne, France, IEEE Computer Society Press (2005)

[3] 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

[4] 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

[5] Stevens, R., Wroe, C., Lord, P., Goble, C. In: Ontologies in bioinformatics. Springer (2003) 635–657

[6] Kiryakov, A., Simov, K., Ognyanov, D.: Ontology middleware: Analysis and design. Technical report, IST Project IST-1999-10132 On-To-Knowledge (2002)

[7] 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

[8] Harth, A., Decker, S.: Yet Another RDF Store: Perfect Index Structures for Storing Semantic Web Data With Contexts. Submitted (2005)