# Deep integration of the OWL ontology language into Ruby using metaprogramming

I n a u g u r a l - D i s s e r t a t i o n
zur
Erlangung des Doktorgrades der
Mathematisch-Naturwissenschaftlichen Fakultät
der Heinrich-Heine-Universität Düsseldorf
vorgelegt von

**Dominic Mainz**
aus Krefeld

Dezember 2008

Aus dem Institut für Informatik
der Heinrich-Heine-Universität Düsseldorf

Gedruckt mit der Genehmigung der
Mathematisch-Naturwissenschaftlichen Fakultät der
Heinrich-Heine-Universität Düsseldorf

Referent: Prof. Dr. Arndt von Haeseler

Korreferent: Prof. Dr. Martin Lercher
Tag der mündlichen Prüfung: 21. Januar 2009

# Publications

Parts of this thesis have been published in the following conference proceedings:

1. Dominic Mainz, Katrin Weller, Jürgen Mainz. (2008) SEMANTIC IMAGE ANNOTATION AND RETRIEVAL WITH IKEN. In *International Semantic Web Conference (Posters & Demos)*

2. Dominic Mainz, Ingo Paulsen, Indra Mainz, Katrin Weller, Jochen Kohl, Arndt von Haeseler. (2008) KNOWLEDGE ACQUISITION FOCUSED COOPERATIVE DEVELOPMENT OF BIO-ONTOLOGIES - A CASE STUDY WITH BIO2ME. In *M. Elloumi, J. Küng, M. Linial, R.F. Murphy, K. Schneider, T. Cristian (eds.) Bioinformatics Research and Development.*, 258-272, Springer, Berlin. (ISBN 978-3-540-70598-7)

Other publications:

1. Indra Mainz, Katrin Weller, Ingo Paulsen, Dominic Mainz, Jochen Kohl, Arndt von Haeseler. (2008) ONTOVERSE: COLLABORATIVE ONTOLOGY ENGINEERING FOR THE LIFE SCIENCES. Inform. Wiss. Praxis, 2, 91-99.

2. Zoulfa El Jerroudi, Stefan Weinbrenner, Dominic Mainz, Katrin Weller. (2008) ONTOVERSE: Kollaborative Ontologieentwicklung mit interaktiver visueller Unterstützung. In: *Mensch & Computer*

3. Ingo Paulsen, Dominic Mainz, Katrin Weller, Indra Mainz, Jochen Kohl, Arndt von Haeseler. (2007) ONTOVERSE: Collaborative Knowledge Management in the Life Sciences Network. In: *Proceedings of the Germany eScience Conference 2007*, Max Planck Digital Library, ID 316588.0.

4. Katrin Weller, Dominic Mainz, Indra Mainz, Ingo Paulsen: Wissenschaft 2.0? Social Software im Einsatz für die Wissenschaft. In: Marlies Ockenfeld (Hrsg.): Information in Wissenschaft, Bildung und Wirtschaft, 29. Online-Tagung der DGI, 59. Jahrestagung der DGI, Proceedings, Frankfurt (Main): DGI, 2007, S. 121-136.

5. Katrin Weller, Indra Mainz, Ingo Paulsen, Dominic Mainz: Semantisches und vernetztes Wissensmanagement für Forschung und Wissenschaft. Erscheint in: WissKom 2007, Wissenschaftskommunikation der Zukunft, 4. Konferenz der Zentralbibliothek im Forschungszentrum Jülich, Proceedings, 2007.

# Danksagung

Mein besonderer Dank gilt Herrn Prof. Dr. Arndt von Haeseler für die Überlassung des interessanten Themas, seine stete Bereitschaft für konstruktive sowie kritische Diskussionen, die mich während der Arbeit motivierend begleitet haben. Für die grosszügige Ermöglichung des Ontoverse-Projekts und den Teilnahmen an Konferenzen bin ich sehr dankbar.

Herrn Prof. Dr. Martin Lercher danke ich sehr für das entgegengebrachte Interesse und die Übernahme des Korreferats.

Sehr herzlich danke ich meiner Arbeitsgruppe für die schönen Jahre. Insbesondere: Ingo, Katrin und Jochen.

Meiner lieben Familie und meinen Freunden danke ich dafür, dass sie da sind; meine Eltern, meine Schwestern, die Ahmadinejads, Deniz, Timo, Nicki, Julia, Paola, Ilija, Tatjana, Elena, Nicki, Julia, Jürgen, Birgit und alle restlichen Mainzer, Schourens ...

Danke Nahal.

# Contents

**1. Introduction** . . . . . . . . . . **1**

1.1 Data Explosion in the Life Sciences and Multimedia Content Management 1

1.2 Ontologies and Semantic Applications . . . . . . . . . . 4

1.2.1 Deep Integration . . . . . . . . . . 5

1.3 Thesis Outline . . . . . . . . . . 6

**2. Background** . . . . . . . . . . **9**

2.1 The *Semantic Web* and its Concepts . . . . . . . . . . 9

2.1.1 Ontologies . . . . . . . . . . 11

2.1.2 The Resource Description Framework (RDF) . . . . . . . . . . 13

2.1.3 The Resource Description Framework Schema RDFS . . . . . . . . . . 13

2.1.4 Web Ontology Language (OWL) . . . . . . . . . . 14

2.1.5 Rules . . . . . . . . . . 16

2.2 Description Logic . . . . . . . . . . 17

2.3 The **ONTOVERSE** Project . . . . . . . . . . 17

2.4 Bio-ontologies . . . . . . . . . . 18

2.4.1 Open Biomedical Ontologies (OBO) . . . . . . . . . . 19

2.5 Multimedia Content and the *Semantic Web* . . . . . . . . . . 19

2.5.1 Ontology-Based Multimedia Content Indexing . . . . . . . . . . 19

2.5.2 Ontology-Based Multimedia Content Retrieval . . . . . . . . . . 20

2.6 The PELLET Reasoner . . . . . . . . . . 20

2.7 *Semantic Web* Frameworks . . . . . . . . . . 20

2.7.1 JENA2 . . . . . . . . . . 20

2.7.2 OWL API . . . . . . . . . . 21

2.7.3 ACTIVERDF . . . . . . . . . . 21

2.8 The Dynamic Programming Language RUBY ........ 22
2.8.1 RUBY Classes, Objects, and Variables ........ 22
2.8.2 RUBY Modules ........ 23
2.8.3 Reflection and Metaprogramming ........ 23
2.9 Deep Integration ........ 23

**3. DEEP SEMANTICS ........ 25**
3.1 Consistency Safeness ........ 25
3.2 Architecture ........ 27
3.2.1 Implemented RUBY classes and modules ........ 29
3.3 The Abstract Syntax of OWL LITE and its contribution to DEEP SEMANTICS ........ 33
3.4 Director: Coordinating Data Workflow and Deep Integration Process ........ 45
3.5 Triple Parser: Mapping OWL Triples to an Abstract Syntax Based Representation in RUBY ........ 48
3.5.1 The Triple Parsing Process ........ 48
3.6 Deep Integration Builder ........ 55
3.6.1 Conversion of the OWL Ontology Definition into a RUBY Module . 57
3.6.2 Conversion of OWL Properties into RUBY Objects ........ 58
3.6.3 Conversion of OWL Classes into RUBY Classes ........ 59
3.6.4 TBox Deep Integration – Assembling of the Deep Integrated RUBY Properties and Classes into a consistent RUBY Representation of the Ontology ........ 60
3.6.5 ABox Deep Integration – Conversion of Instances into RUBY Objects 64
3.7 Utilization of the Deep Integrated Ontology ........ 67
3.7.1 Using DEEP SEMANTICS to convert an Ontology into a RUBY Representation ........ 67
3.7.2 Working with OWL Classes, Properties and Instances in DEEP SEMANTICS ........ 67
3.7.3 XPERIMENTR– A Simple Semantic Application using DEEP SEMANTICS ........ 71
3.8 Comparison of DEEP SEMANTICS with other Semantic Web Frameworks . 81
3.8.1 DEEP SEMANTICS versus OWL API ........ 81
3.8.2 DEEP SEMANTICS versus JENA2 ........ 87
3.8.3 DEEP SEMANTICS versus ACTIVERDF ........ 92

3.9 Discussion . . . . . 96
3.9.1 Programming Complexity . . . . . 98
3.9.2 Runtime and Main Memory Complexity . . . . . 98
3.9.3 Handling Multiple Ontologies in DEEP SEMANTICS . . . . . 99
3.9.4 Conclusions and Outlook . . . . . 99

**4. DEEP SEMANTICS in Action: IKEN and the BIO2ME . . . . . 101**
4.1 **IKEN** . . . . . 101
4.1.1 The **IKEN** ontology . . . . . 102
4.1.2 The **IKEN** Architecture . . . . . 116
4.1.3 The **IKEN** Graphical User Interface . . . . . 119
4.2 The **BIO2ME** Ontology and Information System . . . . . 120
4.2.1 **BIO2ME** Ontology . . . . . 120
4.2.2 **BIO2ME** Information System . . . . . 121
4.3 Discussion . . . . . 121
4.3.1 Semantic Image Management with **IKEN** . . . . . 122
4.3.2 Experiences applying DEEP SEMANTICS . . . . . 122
4.3.3 Conclusions and Outlook . . . . . 123

**5. Summary . . . . . 125**

**6. Zusammenfassung . . . . . 127**

**Bibliography . . . . . 129**

**Appendix . . . . . 135**
6.1 XPERIMENTR Ontology: Classes, Properties and Instances . . . . . 135
6.2 Abstract syntax of OWL LITE . . . . . 139
6.3 Reference Test Implementations . . . . . 142
6.3.1 Test 1: list all classes of the ontology . . . . . 142
6.3.2 Test 2: find all instances matching a certain search term . . . . . 145

# List of Figures


1.1 Growth of GENBANK sequence entries. . . . . . . . . . . . . . . . . 2
1.2 Chronological development of the number of databases listed by the Nucleic Acids Research online Molecular Biology Database Collection. . . . . . 3

2.1 Internet as a maze 1. . . . . . . . . . . . . . . . . . . . . . . . . 10
2.2 Internet as a maze 2. . . . . . . . . . . . . . . . . . . . . . . . . 11
2.3 The *Semantic Web* layers. . . . . . . . . . . . . . . . . . . . . . 12
2.4 Schematic representation of the constructs of an ontology. . . . . . . . 13
2.5 Screenshot of the RDF/XML serialization of a simple RDF Schema for the description of resources related to animals and plants. . . . . . . . . . 15

3.1 Conversion of an OWL class *Program* into a RUBY class. . . . . . . . 26
3.2 Top-level architectural diagram of DEEP SEMANTICS including input and output values. . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
3.3 UML class diagram of the DEEP SEMANTICS main classes. . . . . . . 29
3.4 UML class diagram of the DEEP SEMANTICS helper classes. . . . . . . 30
3.5 UML class diagram of DEEP SEMANTICS custom datatypes. . . . . . . 31
3.6 Simplified UML class diagram extended with meta-programming symbols. 34
3.7 Simplified UML class diagram for the *Ontology* construct extended with meta-programming symbols. . . . . . . . . . . . . . . . . . . . . . 35
3.8 Simplified UML class diagram for OWL class constructs extended with meta-programming symbols. . . . . . . . . . . . . . . . . . . . . . 37
3.9 Simplified UML class diagram for the OWL restriction constructs extended with meta-programming symbols. . . . . . . . . . . . . . . . . . 40
3.10 Simplified UML class diagram for the OWL datatype property constructs extended with meta-programming symbols. . . . . . . . . . . . . . . 41
3.11 Simplified UML class diagram for the OWL object property constructs extended with meta-programming symbols. . . . . . . . . . . . . . . 43
3.12 Activity diagram 1: overview. . . . . . . . . . . . . . . . . . . . . 45
3.13 Activity diagram 2: *Director*. . . . . . . . . . . . . . . . . . . . . 46
3.14 Activity diagram 3: *TripleParser*. . . . . . . . . . . . . . . . . . . 46
3.15 Activity diagram 4: *Adapter*. . . . . . . . . . . . . . . . . . . . . 47
3.16 Activity diagram 5: triple parsing and ontology conversion. . . . . . . . 47

3.17 Activity diagram 6: parsing details. . . . . . . . . . . . . . . . . . . . 48
3.18 Activity diagram 7: TBox parsing. . . . . . . . . . . . . . . . . . . . 49
3.19 Activity diagram 8: property parsing. . . . . . . . . . . . . . . . . . . 50
3.20 Activity diagram 9: datatype property parsing. . . . . . . . . . . . . . . 51
3.21 Activity diagram 10: object property parsing. . . . . . . . . . . . . . . 52
3.22 Activity diagram 11: class parsing. . . . . . . . . . . . . . . . . . . . 53
3.23 Activity diagram 12: ABox parsing. . . . . . . . . . . . . . . . . . . . 53
3.24 Activity diagram 13: instance parsing. . . . . . . . . . . . . . . . . . . 54
3.25 Activity diagram 14: TBox deep integration. . . . . . . . . . . . . . . 56
3.26 Activity diagram 15: ontology module creation. . . . . . . . . . . . . . 58
3.27 Activity diagram 16: property object creation. . . . . . . . . . . . . . . 59
3.28 Activity diagram 17: RUBY ontology class creation. . . . . . . . . . . 60
3.29 Activity diagram 18: deep integration details 1. . . . . . . . . . . . . . 61
3.30 Schematic illustration of the deep integration of an example property. . . . 63
3.31 Activity diagram 19: deep integration details 2. . . . . . . . . . . . . . 65
3.32 Activity diagram 20: *Ghost* creation. . . . . . . . . . . . . . . . . . . 66
3.33 Activity diagram 21: using an existing *Ghost* class. . . . . . . . . . . 67

4.1 Screenshot of the novel semantics enabled web-interface of the **IKEN** application. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103
4.2 Extract of the **IKEN** class hierarchy and a sample photo annotation. . . . 107
4.3 Ontology data pre-processing for **IKEN**. . . . . . . . . . . . . . . . 117
4.4 **IKEN** architecture and implementation set-up. . . . . . . . . . . . . . 118
4.5 **IKEN** details view in the annotation interface. . . . . . . . . . . . . . 118
4.6 **IKEN** refinement view in the annotation interface. . . . . . . . . . . . 119
4.7 **IKEN** search interface. . . . . . . . . . . . . . . . . . . . . . . . . 120

6.1 EBNF of the abstract syntax of OWL LITE. . . . . . . . . . . . . . . . 140
6.2 Extended UML class diagram: EBNF of OWL LITE constructs in relation to their DEEP SEMANTICS classes and modules counterparts. . . . . . . . 141

# List of Tables

3.1 Types of global property constraints that have to be considered by DEEP SEMANTICS . . . . . . . . . . . . . . . . . . . . . . . . 88
3.2 Types of local property constraints that have to be consider by DEEP SEMANTICS . . . . . . . . . . . . . . . . . . . . . . . . 89
3.3 Runtime and main memory complexity comparison between DEEP SEMANTICS and OWL API . . . . . . . . . . . . . . . . . . 90
3.4 Runtime and main memory complexity comparison between DEEP SEMANTICS and JENA2 . . . . . . . . . . . . . . . . . . . . 93
3.5 Runtime and main memory complexity comparison between DEEP SEMANTICS and ACTIVERDF . . . . . . . . . . . . . . . . 96

# Listings


3.1 An concluding *setter* method example . . . . . . . . . . . . . . . . . . . . . 64

3.2 Using DEEP SEMANTICS to create a functional model of **IKEN** . . . . . . . 67

3.3 Including DEEP SEMANTICS into custom RUBY code . . . . . . . . . . . . . 67

3.4 Adding the namespace of the ontology and creating an functional ontology model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68

3.5 Printing out the labels of class *Protein* . . . . . . . . . . . . . . . . . . . . . 68

3.6 Printing out the labels of every *Protein* instance . . . . . . . . . . . . . . . . 68

3.7 Creating a *Protein* and a *BiologicalFunction* . . . . . . . . . . . . . . . . . . . 69

3.8 Adding information about its molecular function to insulin . . . . . . . . . . 69

3.9 Trying to add a second molecular function to the instance *insulin* . . . . . . . 70

3.10 Access of a property RUBY object in DEEP SEMANTICS . . . . . . . . . . . 70

3.11 Using DEEP SEMANTICS to prepare the ontology knowledge base of XPERIMENTR . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72

3.12 Accepting and processing user input . . . . . . . . . . . . . . . . . . . . . . 72

3.13 Implementation of the information retrieval method *retrieveInformation()*: search for matching classes segment . . . . . . . . . . . . . . . . . . . . . 73

3.14 Implementation of the information retrieval method *retrieveInformation()*: search for matching laboratory material instances . . . . . . . . . . . . . . . 73

3.15 Implementation of the information retrieval method *retrieveInformation()*: search for matching laboratory equipment instances . . . . . . . . . . . . . . 74

3.16 Implementation of the information retrieval method *retrieveInformation()*: search for matching *Protocol* instances . . . . . . . . . . . . . . . . . . . . . 75

3.17 Implementation of the XPERIMENTR method *findProtocolsByExperimentMaterials()* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76

3.18 Implementation of the XPERIMENTR method *findProtocolsByExecTime()* . . 77

3.19 XPERIMENTR implementation using OWL API: Print out the class hierarchy of the ontology. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82

3.20 XPERIMENTR implementation using OWL API: *printHierarchy()* methods. . 83

3.21 XPERIMENTR implementation using DEEP SEMANTICS: Print out the class hierarchy of the ontology. . . . . . . . . . 83
3.22 XPERIMENTR implementation using OWL API: information retrieval implementation for instances of ontology class *Protocol*. . . . . . . . . . 84
3.23 XPERIMENTR implementation using OWL API: *findInstancesByLabel()* method. . . . . . . . . . 86
3.24 DEEP SEMANTICS intern implementation of method *find_instances_by_label(label)*. 86
3.25 XPERIMENTR implementation using JENA2: Print out the class hierarchy of the ontology. . . . . . . . . . 91
3.26 XPERIMENTR implementation using JENA2: *printHierarchy()* methods. . . . 91
3.27 XPERIMENTR implementation using JENA2: source code for the retrieval of protocols that can be executed in a specified duration. . . . . . . . . . 91
3.28 KITCHEN MENTOR implementation using ACTIVERDF: retrieving a recipe for a set of included materials. . . . . . . . . . 93
3.29 Consistency problems using *setter* in ACTIVERDF. . . . . . . . . . 95
6.1 Test 1 implementation using OWL API. . . . . . . . . . 142
6.2 Test 1 implementation using JENA2. . . . . . . . . . 143
6.3 Test 1 implementation using DEEP SEMANTICS. . . . . . . . . . 145
6.4 Test 2 implementation using OWL API. . . . . . . . . . 145
6.5 Test 2 implementation using JENA2. . . . . . . . . . 147
6.6 Test 2 implementation using ACTIVERDF. . . . . . . . . . 148
6.7 Test 2 implementation using DEEP SEMANTICS. . . . . . . . . . 149

# 1

# Introduction

Many scientific and industrial areas today face a tremendous increase in the amount of produced data and information. In recent years this development created a large demand for new knowledge management strategies. One very promising approach in particular is the modeling of formal knowledge representation called ontologies (Gruber & Gruber, 1993). Ontologies provide means for incorporating some degree of information about the context and semantics of processed content in computer programs. Furthermore, ontologies represent one of the central components of the Semantic Web (Hendler, 2001). The Semantic Web primarily differs from the current World Wide Web by extending it with a semantic layer of machine-processable metadata, which enables computers and people to interact with each other and exchange data in a meaningful way. The concepts and technologies constituting the Semantic Web are described in more detail in Section 2.1.

This thesis deals with the development of a novel Semantic Web framework. The design of an ontology in the field of image management and a Web application using this ontology as knowledge base are introduced in this work.

## 1.1 Data Explosion in the Life Sciences and Multimedia Content Management

The advancement of life sciences disciplines is strongly related to the efficient management and retrieval of already collected knowledge, information and data (Sahoo *et al.*, 2006). Since the advent of modern high-throughput and laboratory automation technologies, data production about living systems has exploded. Figure 1.1 shows for example the exponential growth of the number of nucleotide sequence entries in GenBank (Benson *et al.*, 2004). GenBank offers an open access, an annotated collection of all publicly available nucleotide and protein sequences. From December 1994 to August 2008, which means in less than 14 years, the number of stored sequences increased 390 times from 237,775 to 92,740,599, doubling approximately every 18

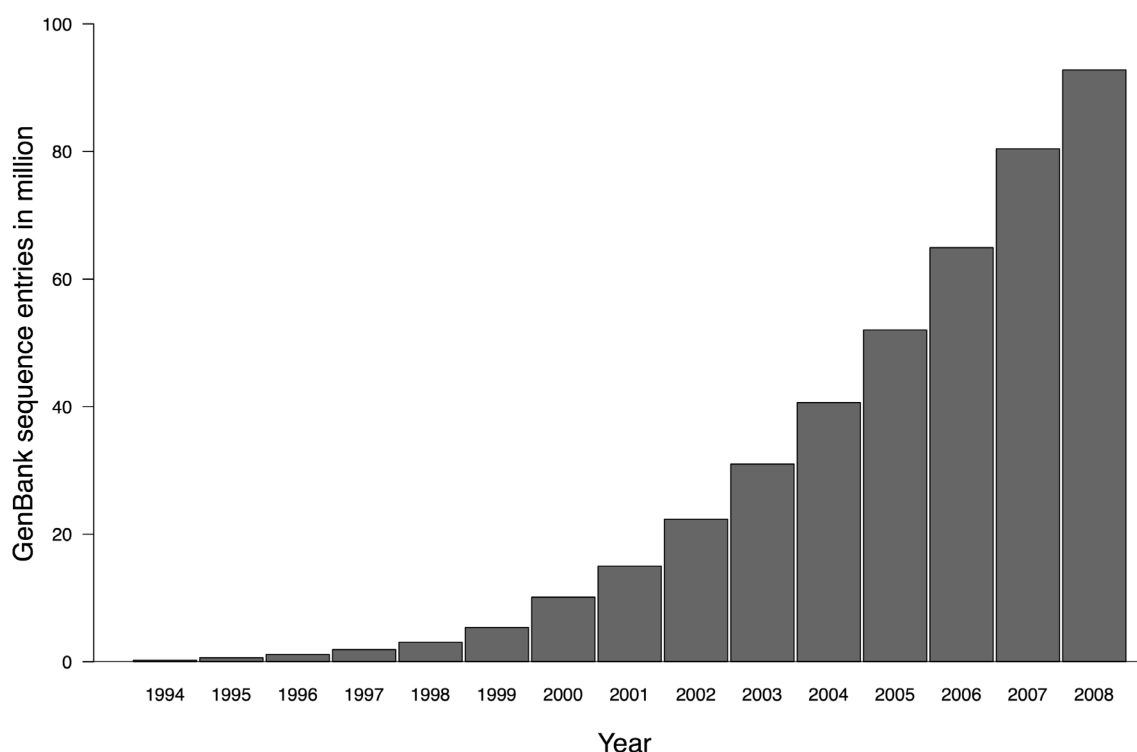


**Figure 1.1: Growth of GENBANK sequence entries.** The figure shows the chronological development of the number of stored sequence entries in the GENBANK databse. The graphic is based on GENBANK statistic reports that are periodically published by the *National Center for Biotechnology Information* of the United States of America.

month. The development of even faster and cheaper sequencing techniques will continue this exponetial growth in the foreseeable future.

A consequence of the explosion of stored nucleotide sequences and other experimentally produced biomedical data (e.g. molecular structures and gene expression data) is the need for analyzing these data in order to enable the extension of the biomedical knowledge and scientific insight into the configurations and processes of living systems. These analysis activities in turn result among other things in the growth of primary and secondary databases as shown in Figure 1.2. The figure shows the chronological development of the number of entries in a collection of biomedical databases, the Nucleic Acids Research online Molecular Biology Database Collection. This collection is a public repository, which currently lists more than 1,000 freely available databases with contents ranging from molecular structures to human genes and diseases, just to name a few. While this collection does not exhaust the number of available databases, it still is an indicator for a general development: a growth that yielded a fivefold increase of the number of available databases from 2000 to 2008.

The data explosion driven increase of biomedical databases lead to a new problems in life sciences: the appropriate management, interchange and indexing of the stored content. Scientists today face three main problems when using biomedical databases:

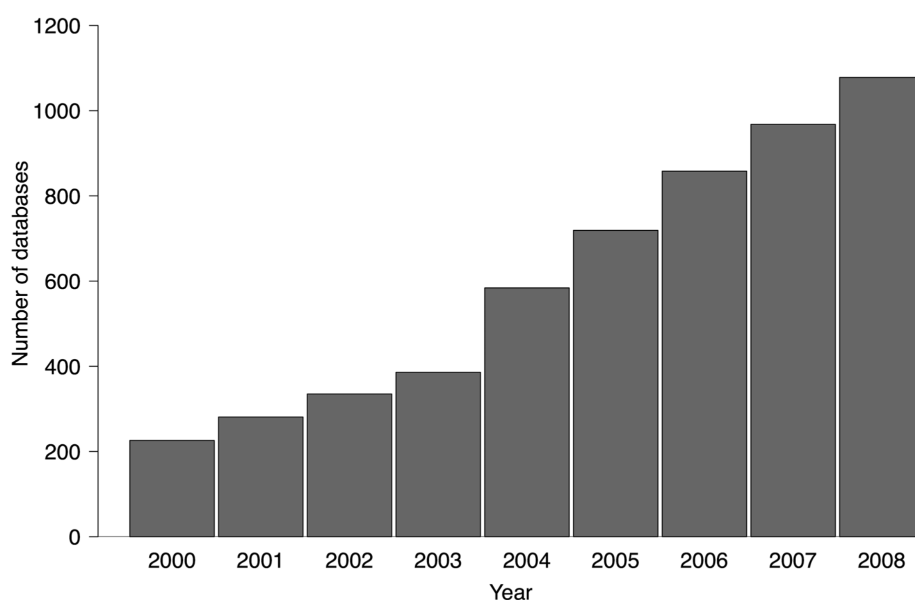


**Figure 1.2: Chronological development of the number of databases listed by the Nucleic Acids Research online Molecular Biology Database Collection.** The used values for the figure where published in Baxevanis (2000, 2001, 2002, 2003) and Galperin (2004, 2005, 2006, 2007, 2008).

---

- Different use of terminology: the lack of a standardized terminology complicates discussion and reproducibility of scientific data. This leads to the use of synonyms and misspelling of concepts, organism or gene names in the diverse databases, which makes it even more difficult to retrieve all information about a certain knowledge entity, like for example a gene or protein.

- Inconsistent terminologies: the existing scientific terminologies are based on natural language and can therefore be interpreted differently. As an example, the fundamental biological term 'gene' is not clearly defined and still discussed within different biological meanings (Pesole, 2008; Stevens *et al.*, 2000). A gene may be defined as *'the coding region of DNA'*, as a *'DNA fragment that can be transcribed and translated into a protein'* or *'DNA region of biological interest with a name and that carries a genetic trait or phenotype'* as a third.

- Navigation in knowledge space: a third challenge is the navigation in the knowledge space spanned by the diverse databases and scientific publications. The sequence of a particular gene may have been stored in one database while additional corresponding information of the expression characteristics may be stored in a second one and additional details about functionalities of the encoded protein even in a third database. It is extremely difficult for biologists to deal with all this scattered information. The continuous growing amount of data, databases and biological knowledge only compounds the situation.

Another important field that benefits from the application of semantic technologies is the broad area of multimedia content management. Similar to the situation in the life sciences, the amount

of digital multimedia information that is accessible via the Internet is growing every day. New devices like digital still cameras, multimedia cellphones and digital video cameras have hit the mass market in recent years and lead to an explosion of multimedia content shared in the Internet. Popular examples of social multimedia sharing Web applications are YouTube (Cheng *et al.*, 2007) for short video broadcasting, and the image exchange platform Flickr (van Zwol, 2007). In these applications only insufficient search functionalities are provided. A semantic-based search for the keyword "animal" on the other hand could for example return pictures showing a dog based on the knowledge model, which consists the relation that a dog is an animal.

Like scientists in the life sciences have a growing demand for knowledge management support as a consequence of the biological data explosion, the acquisition, processing and distribution of multimedia content has raised a demand of diverse user groups, ranging from private users, over medical professionals to employees in the media industry, for more sophisticated semantic multimedia content management solutions (Ahmad, 2007). One of the currently most promising approaches is the bridging of the semantic gap (Hare *et al.*, 2006) through the use of ontologies. The semantic gap between the set of facts a human can identify in a picture, for example, and the quality of the corresponding annotation data can be improved by moving from a mere term based indexing of content to content annotations based on a supporting ontology (Hollink *et al.*, 2003).

As a result of these developments and challenges, bioinformaticians and life sciences' researchers as well as professionals working with multimedia content identified the need to create systems that add and apply the knowledge in the minds of domain experts to the processed content. This knowledge is nowadays in both fields captured and made accessable to both, computers and humans with the use of ontologies.

## 1.2 Ontologies and Semantic Applications

Ontologies are represented using specialized ontology languages that provide inference and integrity rules, which means rules that are used to make implicit knowledge in an ontology explicit as well as to guarantee an ontology's logical validity. Currently, the most important and mostly used ontology language is the *Web Ontology Language* (OWL; Bechhofer *et al.*, 2004). Ontologies are usable in a wide range of applications.

In Jasper & Uschold (1999) a description of a typical ontology application scenario is made that comprises:

1. **Ontology as Specification:** An ontology is used as a formal basis for software specification and documentation in order to improve the specification quality, reliability and to foster knowledge reuse.
2. **Common Access to Information:** In this scenario the ontology primarily functions as a means to provide a shared understanding of the terms and their interrelations in a certain domain, a controlled vocabulary. This vocabulary is then used to support collaborations

between persons and computer applications, respectively. A prominent example for this kind of ontology application scenario is the GENE ONTOLOGY (Ashburner *et al.*, 2000).

3. **Ontology-Based Search:** An ontology is used for searching a repository for desired content, e. g. publications (Delfs *et al.*, 2004), websites (Esmaili & Abolhassani, 2006), images (Ahmad, 2007) or sequence entries (Lu *et al.*, 2006). This semantic information retrieval approach offers faster access to relevant information resources (Finin *et al.*, 2005).

Additional to these ontology application scenarios, further examples for specific application types are semantic-based database integration (Cheung *et al.*, 2007) and ontology-based information extraction (Hu *et al.*, 2004).

### 1.2.1 Deep Integration

The widespread adoption of the Semantic Web largely depends on the support of logic reasoners like RACERPRO (Haarslev & Móller, 2003), FACT++ (Tsarkov & Horrocks, 2006) and PELLET (Parsia & Sirin, 2004), and specialized frameworks like OWL API (Horridge & Bechhofer, 2007), JENA2 (Carroll *et al.*, 2004) and ACTIVERDF (Oren & Delbru, 2006), to name some prominent ones. Developers of Semantic Web applications use these specialized frameworks to convert, retrieve and edit ontology entities. Semantic Web frameworks make ontologies accessible to software applications. However, current frameworks implemented in JAVA like OWL API and JENA2 provide quite complex application programming interfaces (API), which constitute an important obstacle with respect to their adoption in the general Web application development community.

Vrandecic (2005) discusses scripting languages and dynamic programming languages, respectively, as a promising approach to address the aforementioned API complexity challenges of JAVA-based frameworks. Dynamic programming languages like RUBY (Thomas *et al.*, 2004) or PYTHON (Van Rossum, 2003) use implicit declared data types, automatic memory management (garbage collection; Soman & Krintz, 2007) and metaprogramming capabilities to achieve a higher level of programming and more rapid application development (Ousterhout, 1998). Babik & Hluchy (2006) introduce an approach for the deep integration of PYTHON with OWL, offering a more intuitive mapping of OWL into the programming context than classic API's. The deep integration paradigm, which states the reproducing the semantics of OWL in a dynamic programming language, has introduced the idea of importing an ontology directly into the programming context in a way that its classes can be used directly like other classes of the language and therefore avoiding a complicated API.

Oren & Delbru (2006) describe the above mentioned Semantic Web framework ACTIVERDF which is based on the deep integration of RUBY with the *Resource Description Framework Schema* (RDFS; RDF, 2004). While ACTIVERDF provides an object-oriented access to *Resource Description Framework* (RDF) data (Klyne & Carroll, 2004) including its modification, and considerably reduces programming complexity (see also the corresponding comparisons in Section 3.8), it is not sufficient for its utilization in editing OWL ontologies. OWL extends RDFS (Lacy, 2005) with new language constructs of which logical constraints are of particular

importance with respect to ACTIVERDF's current shortcomings. For example, logical constraints can be used to restrict the number (cardinality restriction) and type of entities (universal quantification and existential quantification) that can be interrelated in an ontology. Disregarding logical constraints leads to inconsistencies in the ontology which in turn has the effect that this ontology cannot further be processed using a Semantic Web reasoner. As ACTIVERDF does not enforce any OWL constraints, processed ontologies might become inconsistent.

## 1.3 Thesis Outline

The tremendous increase in the amount of accessible data and information in the life sciences and the multimedia content management has further increased the demand for semantic technologies. Current Semantic Web frameworks offer complex APIs that need significantly more lines of code to accomplish the same functionality as frameworks based on dynamic programming languages.

The dynamic programming language RUBY has gained a large distribution among *World Wide Web* developers. Especially since the release of the rapid World Wide Web development framework RUBY ON RAILS (Thomas *et al.*, 2005), an increasing number of Semantic Web applications (Oren *et al.*, 2007) and life sciences related websites (Goble & Roure, 2007; Roure & Goble, 2007) have been developed in this language.

As described before, ACTIVERDF constitutes a first generation of Semantic Web frameworks for RUBY ON RAILS and RUBY developers, which however is not sufficient when used for editing ontologies in the most prominent ontology language OWL. To further foster the broad use of semantic technologies, especially those making use of OWL, this thesis introduces DEEP SEMANTICS, a second generation of Semantic Web frameworks, that enables the use of logical constraints and which can be used to safely modify OWL ontologies.

Chapter 2 contains background information about technologies and areas of research that are needed to follow the discussion of the results of this thesis. The Semantic Web and its concepts will be described as well as Semantic Web frameworks and tools that were used. Furthermore, does this chapter introduce two relevant application domains of semantic technologies: Section 2.4 *The Semantic Web for Life Sciences* and Section 2.5 *Multimedia Content and the Semantic Web*. Chapter 2 ends with an introduction into the dynamic programming language RUBY, whose metaprogramming capabilities are crucial for the realization of this work.

The ensuing two chapters present and discuss the results of this thesis. Chapter 3, deals with a detailed description of the newly developed Semantic Web framework DEEP SEMANTICS. This is the most important chapter, especially with respect to new scientific insights on how to utilize metaprogramming for deep integration of OWL into RUBY. Section 3.2 of this chapter describes the overall architecture of the framework while Sections 3.4, 3.5 and 3.6 cover implementation details including solutions for specific challenges, which were solved in this work. To give the reader a detailed impression of the particular benefits of the framework, Section 3.7 describes how to program with ontologies converted by DEEP SEMANTICS in RUBY. Section 3.8 contains a comparison of DEEP SEMANTICS with other Semantic Web frameworks

with respect to programmatic complexity and runtime characteristics. The chapter ends with a discussion of the experiences, challenges and various features of DEEP SEMANTICS.

Chapter 4 describes the development of a web-based semantic image management application called **IKEN**. This application is a self-developed proof-of-concept, showing the practical benefits of DEEP SEMANTICS. This chapter presents the developed ontology, designed architecture and a new kind of semantic user interface. Additionally, Section 4.2 describes the **BIO2ME** ontology and information system, which was developed by Mainz (2008) utilizing the framework introduced in this dissertation.

This dissertation is related to two research projects: **ONTOVERSE** and **IKEN**. While the original idea for the development of DEEP SEMANTICS originated in the work for the **ONTOVERSE** project (for details see Section 2.3), the results presented in Chapters 3 and 4 can be considered independent of this project.

In summary, the work contains the following subjects:

- DEEP SEMANTICS (introduced in Chapter 3): Is the main part of this dissertation and was completely designed and implemented by myself – including the XPERIMENTR example (see Subsection 3.7.3).

- **IKEN** (introduced in Chapter 4): A proof-of-concept for the DEEP SEMANTICS framework. While conceptual work, especially regarding overall requirement specifications, marketing and business model elaboration, was done in collaboration with VARION GmbH, the actual implementation, the design of semantic application as well as the underlying ontology was solely developed by myself during work on this thesis. The implementation of OWL DL support was delayed up to a later date in favour of the development of **IKEN**. This decision is well-grounded in the focus on best possible usability features that could be tested and improved during **IKEN**'s development.

- **BIO2ME** information system (described in Section 4.2.2): DEEP SEMANTICS was used by Mainz (2008) for the developed of this system. In the context of this dissertation the **BIO2ME** information system functions as an external reference application which provided experience reports are discussed in Section 4.3.

# 2

# Background

## 2.1 The *Semantic Web* and its Concepts

The idea of the *Semantic Web* (Kanellopoulos & Kotsiantis, 2007) was first coined in 2001 by Tim Berners-Lee (Hendler, 2001), the inventor of the *World Wide Web* (in the following abbreviated as *WWW*, *Web* or Internet) and current director of the *World Wide Web Consortium (W3C)*. The main idea of the *Semantic Web* is to enhance the current *Web* with a semantic layer of machine-processable metadata that enables computers to interact and exchange data in a meaningful way. This metadata is organized in formal models of concepts and their relations called ontologies.

The *Semantic Web* is frequently described as being a web of data while the current *Web* can be seen as a web of documents which are linked to other related web pages. In contrast, a web of data is characterized by the additional availability of metadata that describes and connects certain kinds of data (like address data, document descriptions or image content annotations). This metadata layer can then be used to foster the exchange of data between applications as well as to record how the data relates to real world objects.

From a user related point of view, the *Semantic Web* enables a new kind of navigation in knowledge spaces accessible via the Internet. Figure 2.1 shows a typical search process using current Internet navigation possibilities. A user enters a query into a search engine. She gets a list of hyperlinks with some additional information about the linked resources, for example websites, web-interfaces of biomedical databases or publication databases, as well as possible Intranet resources as response. From that point on she has to click through this list of links until one of the visited websites contains the needed information. Instead of just looking at the results listed by the search engine she can also follow additional links she encounters on the visited websites. In this respect, the Internet is like an information maze where the user has to search through different paths of hyperlinks, often returning back to a previous visited websites, until the user finds more or less by chance the desired information.

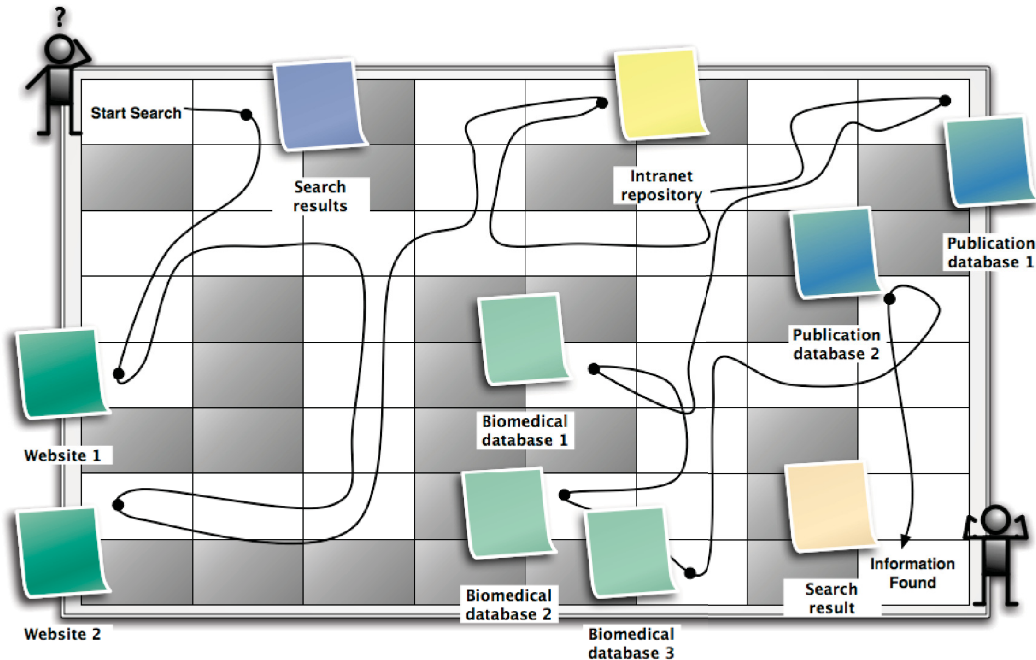


**Figure 2.1: Internet as a maze 1.** The *Web* until version 2.0 is characterized by keyword-based searching that often resembles an attempt to find the right path through a maze – in this case the path to the searched information.

---

In comparison to the current *Web* search processes, semantic technologies can offer some improvements. Figure 2.2 schematically shows how the semantic technologies can convert the search process into a knowledge navigation process. Every information source in this example is semantically annotated and provides metadata about its content. Based upon this metadata it is possible to develop applications that deliver context-sensitive interfaces which enable the users to navigate over different knowledge sources. In Figure 2.2 these kind of knowledge navigation applications is coined a *Semantic Web Guide*. The interaction with a *Semantic Web Guide* starts once again with a search query of the user and the response is typically a list of possible sources of interest, too. The big difference of the semantics based approach is, that the *Semantic Web Guide* can use the metadata of the connected resources to provide the user with information about the relation of the presented search result with the initial search query. The *Semantic Web Guide* offers the user the same results as in Figure 2.1. But this time the user can directly see the semantic nature of the relation (indicated by the *Relation Type* labels) and hyperlink, respectively.

The *Semantic Web* is composed of layers of increasingly specialized technologies. Figure 2.3 shows this layered approach including the following technologies: the *Resource Description Framework* (described in Subsection 2.1.2), the *Resource Description Framework Schema*

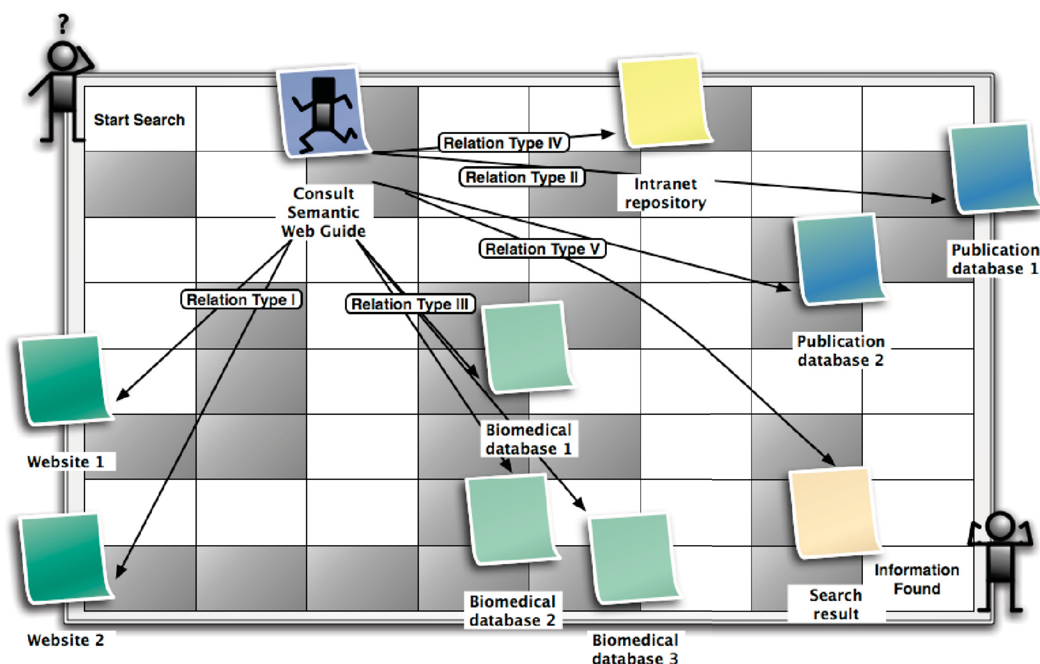


**Figure 2.2: Internet as a maze 2.** The *Semantic Web* as an enabling technology to get from a searchable *Web* to a naviagble knowledge space.

---

(described in Subsection 2.1.3), the *Web Ontology Language* (described in detail in Subsection 2.1.4), and the *Rules* (described in Subsection 2.1.4).

### 2.1.1 Ontologies

The philosophical discipline ontology is the study or concern about what kinds of things exist - what entities or *'things'* there are in the universe (Blackburn, 2007). Computer science has borrowed the term where it is now used in the more narrow sense of special kinds of knowledge representation models in the subsection artificial intelligence. Ontologies consist of concepts, the relations between them and instances of concepts. Additionally they can make use of logical axioms and restrictions. Ontology languages provide inference and integrity rules for ontologies, that means rules that are used to make implicit knowledge stored in the ontology explicit as well as for the guarantee of their logical validity. The most cited defintion of ontology in computer science is the following one by Gruber: *'the specification of conceptualisations, used to help programs and humans share knowledge'* (Guarino *et al.*, 1993).

Figure 2.4 schematically shows the most important building blocks of an ontology.

Ontologies are used for different purposes. Some of the most important application cases are:

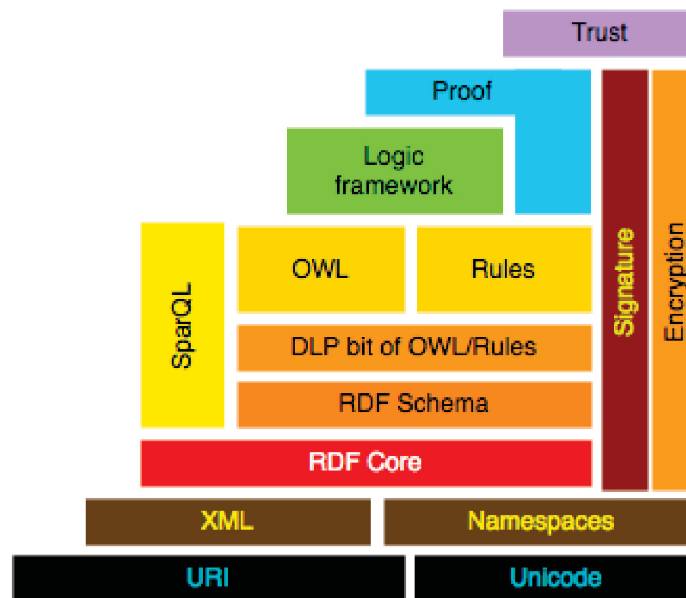


**Figure 2.3: The *Semantic Web* layers.** A layered approach to the *Semantic Web*. The included technologies are: **URI**: an URI is a compact string of characters to identify an entity, accessible on the Internet. The Uniform Resource Locator (URL) to a website is an example of an URI. **Unicode**: an industry standard which provides a unified system for representing textual data. **XML**: the Extensible Markup Language is a general-purpose specification for creating custom markup languages. XML is recommended by the *World Wide Web Consortium*. Regarding to the *Semantic Web*, XML is used for the serialization of RDF, RDFS, OWL and SWRL for example. **Namespaces**: an abstract container that is used for the grouping of URI's. **RDF core**: the Resource Description Framework (see Subsection 2.1.2). **RDF Schema**: the Resource Description Framework Schema (see Subsection 2.1.3). **DLP**: the Description Logic Programs or Description Logic Programming. **OWL**: the *Web Ontology Language* (see Subsection 2.1.4). **Rules**: While an ontology describes a set of objects in a machine-readable way, *Rules* describe how to infer new information from an ontology (see Subsection 2.1.5). **SPARQL**: a query language for RDF. SPARQL allows for a query to consist of triple patterns, conjunctions, disjunctions, and optional patterns. **Logic framework**: for example OWL API (see Subsection 2.7.2), JENA2 (see Subsection 2.7.1) and DEEP SEMANTICS (see Chapter 3). **Proof**: the valid derivation of the correctness (verification) or incorrectness (falsification) of a statement out of true premises. **Signature**: an electronic signature links data with electronic information, which enables the identification of the signer and allows to check the integrity of the signed electronic information. **Encryption**: the process, with which a readable text (or other kind of information that is typically converted into a string of 0's and 1's) is converted with the help of an encoding procedure into an "illegible" (not simply interpretable character sequence). **Trust**: Trust in the semantic applications, in the *Semantic Web* and in all of its actors is the last and most abstract layer. *Source: Tim Berners-Lee. Web for real people, 2005. URL: http://www.w3.org/2005/Talks/0511-keynote-tbl/*

---

- A knowledge base for a community - context and definitions in an ontology make a meaning explicit and support knowledge sharing.
- A controlled vocabulary for indexing/annotating data, enabling semantic search, data integration and knowledge access.
- A knowledge base for a computer program (an expert system for example).

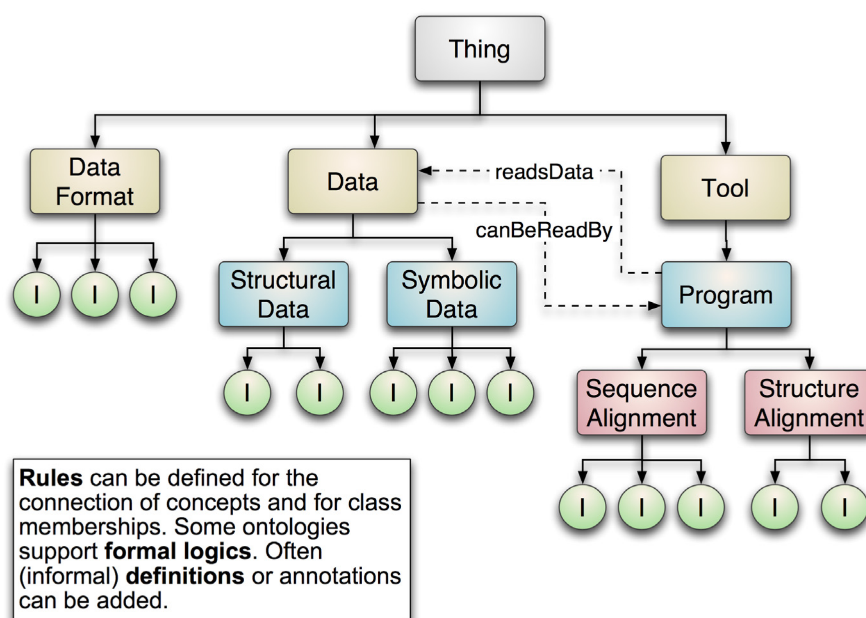


**Figure 2.4: Schematic representation of the constructs of an ontology.** The example is based on the **BIO2ME** ontology which is further described in Subsection 4.2 (page 120). Rectangles symbolize ontology classes and concepts, respectively. Green circles represent instances of the classes they are connected with. Solid arrows between classes indicate their hierarchical relations, pointing to the subclasses. The dotted arrows represent custom defined object property relations between classes.

---

### 2.1.2 The Resource Description Framework (RDF)

The Resource Description Framework (RDF) is a general-purpose language for representing information or metadata (data about data) about *Web* resources. It is based on the *Semantic Web* effort of the *World Wide Web Consortium* and allows multiple metadata schemes to be read by humans as well as to be parsed by machines. The RDF data model is constructed out of statements in the form of subject-predicate-object expressions, called triples in RDF terminology. An example for one of such triple is the following: *<bio2me:StructureVisualizationTool>* *<rdfs:subClassOf>* *<bio2me:StructureAnalysisTool>*. It offers a simple but useful semantic model based on directed acyclic graph structures. Both, nodes and edges are marked with unique designators. RDF exists in two common serialization formats: one XML format and the so called Notation 3 (N3).

A resource in the sense of RDF is everything that can be identified by an Uniform Resource Identifier (URI). URI's for example can be used to name web pages, images or abstract concepts. RDF is a modeling language for defining statements about these resources and the interrelations among them.

### 2.1.3 The Resource Description Framework Schema RDFS

RDF Schema (RDFS) is the RDF's vocabulary description language, according to the W3C *Semantic Web* Activity Statement: *"RDF's vocabulary description language, RDF Schema, is a*

*semantic extension of RDF. It provides mechanisms for describing groups of related resources and the relationships between these resources. RDF Schema vocabulary descriptions are written in RDF using the terms described in this document. These resources are used to determine characteristics of other resources, such as the domains and ranges of properties."*

From the "view" of a computer system, the URI designators introduced by the user are simply character strings without a meaning beyond that. Additional semantic assertions therefore have to be added, in order to bring computer systems to draw conclusions which are based on this type of human background knowledge. By means of RDFS it is possible to specify such schema knowledge with the terms used in a vocabulary. RDFS thereby helps users to define the proper use of a vocabulary composed in RDF. Although an RDF Schema isn't required for valid, RDF it does help to prevent confusion when sharing a vocabulary.

The possibility of modeling schematic knowledge constitutes RDFS as an ontology language, with which it is possible to describe a whole number of semantic dependencies occurring in a domain. RDFS has also its boundaries, thus it is also called a lightweight ontology language. For more demanding use cases, more expressive ontology languages like OWL are necessary that, however come along with longer inference runtimes.

RDFS consists of definitions about what classes and properties are and how they can be further described. RDFS classes can be considered as groups of RDF resources. The members of a class are known as instances of this class. Although being groups of resources, classes are themselves resources, too.

### 2.1.4 Web Ontology Language (OWL)

The Web Ontology Language (OWL) is a set of knowledge representation languages, administrated by the W3C. OWL has been specifically designed to be used for *Semantic Web* applications that need support for sharing knowledge and meaning between machines and humans. OWL is built on RDF and RDFS and adds more vocabulary for describing properties and classes like cardinality constraints (for example "at least one"), disjointness between classes or characteristics of properties (for example transitivity). OWL is available in three different expressiveness levels: OWL LITE, OWL DL and OWL LITE.

#### OWL FULL

OWL FULL is the most expressive sub-language of OWL. OWL FULL accomplishes all restrictions of OWL DL and OWL LITE but ignores decidability issues. It permits classes to be treated simultaneously as both, collections of instances and individuals. In addition, a given datatype property can be specified as being inverse-functional, thus enabling for example, the specification of a string as an unique key.

#### OWL DL

OWL DL puts certain restrictions on the available OWL constructs:

```
– <rdf:RDF xml:base="http://www.owl-ontologies.com/Ontology1225890942.owl">
    <owl:Ontology rdf:about=""/>
    <rdfs:Class rdf:ID="Organism"/>
  – <rdfs:Class rdf:ID="Plant">
      <rdfs:subClassOf rdf:resource="#Organism"/>
    </rdfs:Class>
  – <rdfs:Class rdf:ID="Animal">
      <rdfs:subClassOf rdf:resource="#Organism"/>
    </rdfs:Class>
  – <rdf:Property rdf:ID="eats">
      <rdfs:domain rdf:resource="#Animal"/>
      <rdfs:range rdf:resource="#Organism"/>
    </rdf:Property>
    <Plant rdf:ID="Tree"/>
  – <Animal rdf:ID="Antelope">
    – <eats>
        <Plant rdf:ID="Grass"/>
      </eats>
    </Animal>
  – <Animal rdf:ID="Tiger">
      <eats rdf:resource="#Antelope"/>
    </Animal>
  – <Animal rdf:ID="Elephant">
      <eats rdf:resource="#Grass"/>
    </Animal>
  – <Animal rdf:ID="Lion">
      <eats rdf:resource="#Antelope"/>
    </Animal>
  </rdf:RDF>
```

**Figure 2.5: Screenshot of the RDF/XML serialization of a simple RDF Schema for the description of resources related to animals and plants.**

---

- OWL DL requires a pairwise separation between classes, datatypes, datatype properties, object properties, annotation properties, ontology properties, individuals, data values and the built-in vocabulary. This means that, for example, a class cannot be at the same time an individual or a property.

- Because of the disjointness of datatype properties and object properties the property characteristics "inverse of", "inverse functional", "symmetric", and "transitive" cannot be specified for datatype properties.

- Cardinality constraints cannot be asserted to transitiv properties, their inverse properties or any of their superproperties.

- As the sets of object properties, datatype properties, annotation properties, and ontology properties have to be mutually disjoint an annotation property cannot be at the same time a datatype or object property.

**OWL LITE**

OWL LITE has the weakest expressivity of the three sublanguages of OWL. It primarily supports the construction of hierarchies of classes and properties as well as simple constraints. For example, while it supports cardinality constraints, it only permits cardinality values of 0 or 1. OWL Lite imposes additional restrictions on the available OWL constructs:

- OWL LITE requires a pairwise separation between classes, datatypes, datatype properties, object properties, annotation properties, ontology properties, individuals, data values and the built-in vocabulary. This means that, for example, a class cannot be at the same time an individual or a property.
- Because of the disjointness of datatype properties and object properties the property characteristics inverse of, inverse functional, symmetric and transitive cannot be specified for datatype properties.
- Cardinality constraints cannot be asserted to transitiv properties, their inverse properties or any of their superproperties.
- As the sets of object properties, datatype properties, annotation properties and ontology properties have to be mutually disjoint an annotation property cannot be at the same time a datatype or object property.
- OWL LITE forbids the use of enumerations (owl:oneOf), unions (owl:unionOf), complements (owl:complementOf), default values (owl:hasValue) and disjointness between classes (owl:disjointWith).

### 2.1.5 Rules

Rules play an important role in artificial intelligence expert system shells and in knowledge based systems, respectively, as a means to infer new facts out of a set of known facts. Ontology languages, such as RDFS and OWL, are primarily designed to describe concepts in a knowledge domain. They offer language constructs to describe classes (concepts), properties (attributes and relationships, respectively) as well as constructs to capture certain restrictions on the use of properties and to define complex classes using set operators for example. Although ontology languages have some built-in inference rules to deduce new facts on the basis of hierarchical *is-a* relations (*"If A is a B and B is a C, then A is a C."*), they do not allow the explicit definition of custom rules in order to synthesize new facts from those stored in the knowledge base. To overcome this shortage of ontology languages rule languages like SWRL are designed to specify data transformation rules.

### SWRL

The *Semantic Web Rule Language* (SWRL) is a W3C proposal for a rule language that is based on a combination of the OWL (DL and Lite) with the Unary/Binary Datalog RuleML sublanguages of the Rule Markup Language (Horrocks *et al.*, 2003b). SWRL rules can express knowledge, which is not expressible in OWL and extends the set of OWL axioms to include Horn-like rules. SWRL rules are of the form of an implication between an antecedent (body) and consequent (head). An example of SWRL is the following:

$$hasParent(?x, ?y) \ \textbf{AND} \ hasBrother(?y, ?z)$$

$$\Rightarrow hasUncle(?x, ?z)$$

Explanation: If **x** has parent **y** and **y** has brother **z** then **x** has uncle **z**.

## 2.2 Description Logic

Description logics (Baader *et al.*, 2003) are a family of knowledge representation languages. Most description logics are a subset of first order logics, however contrary to these, description logics are decidable. This allows to reason over a description logic to infer new knowledge from available knowledge. Usually a description logic is formally divided into a terminological box (TBox) and an assertional box (ABox). The TBox contains the knowledge about the concepts of a domain, including the possible relations between the concepts, the terminological knowledge.

The ABox in contrast, contains the knowledge about the entities or instances of these concepts, as well as their relations among themselves, and represents the status of the modeled world. Description logic is of great importance for ontology languages like OWL DL and therefore the *Semantic Web*.

## 2.3 The ONTOVERSE Project

The **ONTOVERSE** consortium consists of several scientific and economical project partners. These partners jointly develop a web-based platform for collaborative ontology engineering and management in life sciences. The **ONTOVERSE** project aims to establish a platform to provide tools for designing ontologies, which helps scientists to build social networks. It therefore comprises support for collaborative ontology engineering (a collaborative ontology editor), an ontology based publication management system and solutions for knowledge exchange in scientific communities.

The idea for DEEP SEMANTICS originated within the development of **ONTOVERSE**, a cooperative project within the promotional focus of the German Federal Ministry of Education and Research on eScience and knowledge management. While being one of the initiators of **ONTOVERSE**, the central objective of that project is the development of a new, internet-based application for cooperative and interdisciplinary ontology building. Originally,

DEEP SEMANTICS was intended to be used in cooperation with the rapid web development framework RUBY ON RAILS for the implementation of an ontology editing interface supporting insertion of automatically extracted concepts and instances into the ontology. During the project duration new requirements and technical results arose, that gave a new direction in the development of DEEP SEMANTICS: instead of supporting schema and facts processing (more precisely TBox and ABox editing – see Section 2.2 on page 17) DEEP SEMANTICS's design priority was to put solely on facts manipulation. Reasons for that decision were:

1. While used design principles (deep integration and meta-programming) are ideal for the development of systems based on static concept schemata and intended for facts manipulation additional schema editing capabilities cannot benefit.
2. Concept schemata manipulations require a fundamentally different implementation approach than support for facts manipulation does. Therefore, two different types of implementation would have been required counteracting the idea of an easy to use framework.
3. The version of DEEP SEMANTICS realized in this work supports facts editing and gained the highest priority in order to realize the development of **IKEN** and another application called **BIO2ME** which exemplifies the advantages of DEEP SEMANTICS for application development in bioinformatics.

## 2.4 Bio-ontologies

Soldatova & King (2007) discuss the formalization of scientific knowledge, particularly ontology engineering for biological applications. Bodenreider & Stevens (2006) review current trends and future directions of bio-ontologies and their applications within biomedicine. Summarizing key points of the aforementioned publication are the following:

- Use of ontologies within biomedical domains is already mainstream.
- Necessity of ontologies in order to be able to compute with the knowledge components in biology and medical research is recognized. This becomes also apparent regarding to the growing list of specialized bio-ontology centers like the German *Institute for Formal Ontology and Medical Information Science (IFOMIS)* [1] or *The National Center for Biomedical Computing* [2] which is one of the *National Centers for Biomedical Computing* in the United States of America.
- While particularly in biology, ontologies are often used as controlled vocabularies for describing data (Avraham *et al.*, 2008), there is a tendency to use increasingly complex formalities (Stevens *et al.*, 2007).

[1] http://www.ifomis.org
[2] http://bioontology.org

### 2.4.1 Open Biomedical Ontologies (OBO)

The Open Biomedical Ontologies (OBO) (Smith *et al.*, 2007) are an umbrella consortium that provides ontologies for shared use across different biomedical domains. In November 2008, 54 ontologies were available via the OBO website. The OBO Foundry has the goal to create a suite of orthogonal – that means complemental – interoperable reference ontologies in the biomedical domain. OBO ontologies are serialized in OBO format or in OWL with OBO ontologies being map-able to OWL (Golbreich *et al.*, 2007).

One of the most important ontologies listed by OBO is the GENE ONTOLOGY (Ashburner *et al.*, 2000). GENE ONTOLOGY currently[3] contains over 27,769 terms ordered in the three sub-ontologies: biological processes, molecular functions, and cellular components for gene products. As a controlled annotation vocabulary the GENE ONTOLOGY has become a de facto standard for many biomedical databases. However, it has to be mentioned that beside its great success story and huge merit for bioinformatics and *Life Sciences*, the logical complexity of GENE ONTOLOGY is rather weak. GENE ONTOLOGY uses only *is-a*, *part-of*, *regulates*, *positively_regulates* and *negatively_regulates* to connect concepts. It makes no use of sophisticated logical constructs (for example constraints on properties or set operators and axioms, respectively) provided by state-of-the-art ontology languages like OWL.

## 2.5 Multimedia Content and the *Semantic Web*

Production and consumption of multimedia content is steadily growing, which makes semantics-based multimedia content indexing and retrieval essential for effective multimedia management. The next two subsections introduce ontology-based multimedia content indexing and retrieval to give some background information for Chapter 4.

### 2.5.1 Ontology-Based Multimedia Content Indexing

Semantic content annotation is the basis for semantic content retrieval (Halaschek-wiener *et al.*, 2005). Annotating (or indexing) is the process of adding content-descriptive keywords to multimedia content (for example pictures, videos, music or documents). If no underlying vocabulary is provided (as in social tagging systems), the semantics of keywords, their meanings and connections, are not represented. An ontology can capture these semantics, for example words that have the same meaning (synonyms), concepts that are parts of others (meronymy, *part-of-relation*), sub-concepts (hyponymy, *is-a-relation*) or various self-defined relations (for example *isLocatedIn*, *hasColour*).

[3] Release of December 1st, 2008

### 2.5.2 Ontology-Based Multimedia Content Retrieval

Semantic Image Retrieval promises a more precise search for documents than retrieval based on non-semantic keywords or full texts. Traditional keyword based system search for *"blue car"*, would retrieve all images which are labeled with *"blue"* as well as with *"car"*. The result would include a picture of a blue bicycle next to a red car). Yet, an underlying ontology can capture the fact that blue is a color which may be a property for several objects such as cars.

## 2.6 The PELLET Reasoner

PELLET (Parsia & Sirin, 2004) is an open source JAVA OWL DL inference engine that can be used standalone or combined with frameworks like JENA2 and OWL API. PELLET implements tableau algorithms that manipulate description logic expressions. Inference engines are primarily used to determine if an ontology is logical consistent and/or to infer all implicit facts included in a consistent ontology. PELLET was used in this thesis for the realization of the semantic image management application **IKEN** described in Chapter 4 to check consistency before used with DEEP SEMANTICS. Other OWL reasoners that could have been used are for example FACT++ (Tsarkov & Horrocks, 2006) and (Haarslev & Moller, 2001).

## 2.7 *Semantic Web* Frameworks

The three *Semantic Web* frameworks described in the following subsections are used as references for a comparison with DEEP SEMANTICS in Chapter 3 (see Section 3.8).

### 2.7.1 JENA2

JENA2 is a JAVA framework (Carroll *et al.*, 2004) for the development of *Semantic Web* applications. JENA2 is an open source project, originally developed at the HP Labs Semantic Web Programme (for further details of this program see: http://www.hpl.hp.com/semweb/). It supports a wide range of *Semantic Web* technologies like RDF, RDFS, SPARQL and OWL. JENA2 offers inference functionalities and can be easily used together with external reasoners like PELLET for example. JENA2 transforms an OWL/RDF ontology into an object-oriented abstract data model that enables the programmatic access of ontology constructs via an API – lowering its usability for developers who are not experts in ontologies and semantic technologies. To get a superficial insight into the API, the following list comprises some of its most important classes and JAVA interfaces, respectively:

- **ModelFactory**: Provides methods for reading ontology data and creating *InfModel* and *OntModel* objects.
- **InfModel**: JENA2 representation object for an RDF/RDFS model including inferred triples (implicit facts are made explicit using inference mechanisms).

- **OntModel**: This JAVA interface provides simplification methods for the access of OWL language constructs. *OntModel* can be used together with base or inferred models.

- **OntClass**: Ontology classes are converted into instances of *OntClass*.

- **OntProperty**: OWL datatype properties and object properties translated into instances of *OntProperty*.

### 2.7.2 OWL API

The OWL API (Bechhofer *et al.*, 2003; Horridge & Bechhofer, 2007) is an open source JAVA framework for the processing of OWL. Similar to the JENA2 framework, ontology constructs are accessible via an API. This in turn – also analogous to JENA2– has the disadvantage that software developers have to deal with this API additionally to OWL. One of the main distinguishing factors between OWL API and JENA2 are their conceptual processing paradigms. While JENA2 manages the ontology as RDF graphs, OWL API is oriented towards OWL abstract syntax (Peter *et al.*, 2004).

OWL API is found at the core of many *Semantic Web* tools for example in version 4.0 of the ontology editor PROTÉGÉ(Knublauch *et al.*, 2004) and the *Semantic Web* editor SWOOP (Kalyanpur *et al.*, 2006) and supports fundamental tasks such as reading, saving, and manipulating ontologies. For inference support, OWL API currently offers access to reasoners PELLET and FACT++. Important classes and JAVA interfaces are:

- **OWLOntologyManager**: This interface is the central access point of the API and is used to load, create and access ontologies.

- **OWLOntology**: Comprises a set of axioms making up the ontology. These axioms then reference the ontological building blocks: classes, properties and instances.

- **OWLClassAxiom**: Represents an OWL class axiom like defined in the abstract syntax (Peter *et al.*, 2004) – see also Section 3.3 for details about the structure of OWL abstract syntax.

- **OWLClass**: While the previous interface models class axioms, *OWLClass* represents actual OWL ontology classes. It can be used for example, to access a subclass of a class, instances and disjoint classes.

### 2.7.3 ACTIVERDF

Oren & Delbru (2006) describe the *Semantic Web* framework ACTIVERDF which is implemented in RUBY. This framework utilizes RUBY's metaprogramming capabilities in order to avoid the implementation of an abstract and – with respect to software development – complicating API. Instead, ontology's classes, properties and instances are transformed into RUBY

classes, RUBY instance methods and RUBY instances. Thereby, this framework provides an object-oriented access to RDF data, considerably reducing programming complexity.

Although being processable as RDF data, specific OWL language features, like logical constraints, are not supported. This restricts its possible application areas significantly.

## 2.8 The Dynamic Programming Language RUBY

DEEP SEMANTICS as well as large parts of **IKEN** project were developed in the dynamic programming language RUBY. In contrast to pure programming languages, dynamic programming languages are not compiled into machine code, but interpreted during runtime. This offers possibilities to modify the code base during runtime, enabling reflection and metaprogramming. RUBY is multi-paradigm programming language allowing procedural, as well as object-oriented or functional programming.

The creator of the programming language RUBY, Yukihiro Matsumoto, stated[4] that RUBY is designed for programmer productivity and ease-of-use, making it perfectly suitable for the implementation of the easy to apply *Semantic Web* framework developed in this dissertation. RUBY includes aspects of the languages PERL (Christiansen & Torkington, 2003), SMALLTALK (Goldberg & Robson, 1989), EIFFEL (Meyer, 1992), ADA (International, 1995) and LISP (Graham, 1995). RUBY's characteristic, that everything is an object for example, is inherited from SMALLTALK while its syntax largely resembles that of PERL.

Further language features, amongst others, are automatic garbage collecting, a large standard library and operator overloading. Operators like "+", "=", or "==" have different implementations depending on the types of their arguments. The following subsections give some background information for the implementation details of DEEP SEMANTICS covered in the next chapter.

### 2.8.1 RUBY Classes, Objects, and Variables

RUBY is a completely object-oriented language. Everything is an object: instances of a particular class are objects as well as these classes are objects of type *Class*. RUBY classes can always be extended by adding new methods or modifying existing ones – even at runtime, which is described in Subsection 2.8.3.

Variables hold references to objects, not the objects themselves. This was useful for the implementation of DEEP SEMANTICS because it reduces the amount of required main memory which otherwise could have an exponential growth, depending on the relations defined in a particular ontology. Variables can be defined belonging to different scopes ranging from local variables – having most limited scope – to instance variables, class variables and global variables (which are accessible form everywhere in the program).

[4] http://www.artima.com/intv/ruby4.html

### 2.8.2 RUBY Modules

In object-oriented languages, inheritance allows programmers to create a class that is a specialization of another class. Though RUBY does not support multiple inheritance, classes can import modules, so called *mixins*. Modules have been important for the consideration of multiple inheritance as provided by OWL LITE into DEEP SEMANTICS (see Chapter 3).

Modules can be used to group methods, classes and constants. They provide a namespace which is helpful for avoiding name clashes. Modules cannot be instantiated directly but have to be inherited by a class first.

### 2.8.3 Reflection and Metaprogramming

Metaprogramming means that the program code is produced by other program code. RUBY offers mechanisms for metaprogramming for it by providing for example methods *module_eval* and *define_method*. The weakened form of metaprogramming is reflection. Reflection means that a program knows its own structure and can modify it, if necessary, whereby values can be modified, but the program structure remains fixed. Without metaprogramming, deep integration of OWL into RUBY, or similar dynamic programming languages, is not possible. Therefore, metaprogramming is one of the central concepts of the next chapter.

## 2.9 Deep Integration

The process of converting an OWL ontology into functional code is called deep integration. In the context of the Semantic Web the term deep integration was coined by Obie Fernandez (Fernandez, 2005). It represents the idea of an integration of OWL with a dynamic typed language like RUBY or PYTHON that goes beyond conventional OWL libraries like OWL API or JENA2 for the JAVA programming language. The main idea of deep integration is to integrate OWL ontologies into a RUBY library (in the case of DEEP SEMANTICS) in a way that the integrated ontology exists at the same level of implementation as other functional libraries.

Instead of using a library like JENA2 to deal with OWL via an dedicated API the deep integration process of DEEP SEMANTICS extends RUBY's set of available libraries by making the ontology including its logic available as natural RUBY objects. This approach enables RUBY developers the use of the logic programming paradigm (Grosof, 2003) by the utilization of Description Logics. Thus, developers are able to use OWL constructs additionally to normally defined classes.

# 3

# DEEP SEMANTICS

The development of the **DEEP SEMANTICS** framework constitutes the main part of this thesis and is described in detail in this chapter. DEEP SEMANTICS has been designed to foster the use of semantic technologies by providing a framework that converts a given OWL ontology into a functional RUBY code representation. The architecture of the DEEP SEMANTICS framework has been prepared to support conversion of the OWL expressivity levels OWL LITE and OWL DL. The current version 0.9 of DEEP SEMANTICS however works exclusively with OWL LITE ontologies as the implementation of OWL DL support is still in alpha state.

The deep integration approach as it is integrated in DEEP SEMANTICS is exemplified by Figure 3.1. This schematizes the conversion of an OWL class *Program* into a deep-integrated RUBY representation of this class. The local name of the OWL class is used as name of the corresponding RUBY class. In the case of class *Program* a *subClassOf* relation to OWL class *Tool* is stated. For the deep integration process this superclass-subclass relation has to be considered in such a way that all attributes as well as their associated *setter* and *getter* methods are inherited by the subclass. *Program* inherits the *getter* methods *isPredecessorOf*, *isSuccessorOf* and *isUsedInProgram* from *Tool*.

## 3.1 Consistency Safeness

DEEP SEMANTICS is *consistency safe*. In this thesis *consistency safeness* is defined as follows: an OWL ontology processing software is *consistency safe*, if it is not able to generate logical inconsistencies during runtime. *Consistency safeness* is an important feature for ontology-based applications, which are utilized to modifiy facts of an ontology, e. g. systems that use ontologies for semantic annotation (like **IKEN** described in Chapter 4) or semantic decision support.

DEEP SEMANTICS is the first *consistency safe* Semantic Web framework for OWL LITE using deep integration. The currently primarily used frameworks JENA2 and OWL API avoid the problem of *consistency safeness* by providing reasoning capabilities that detect inconsistencies.

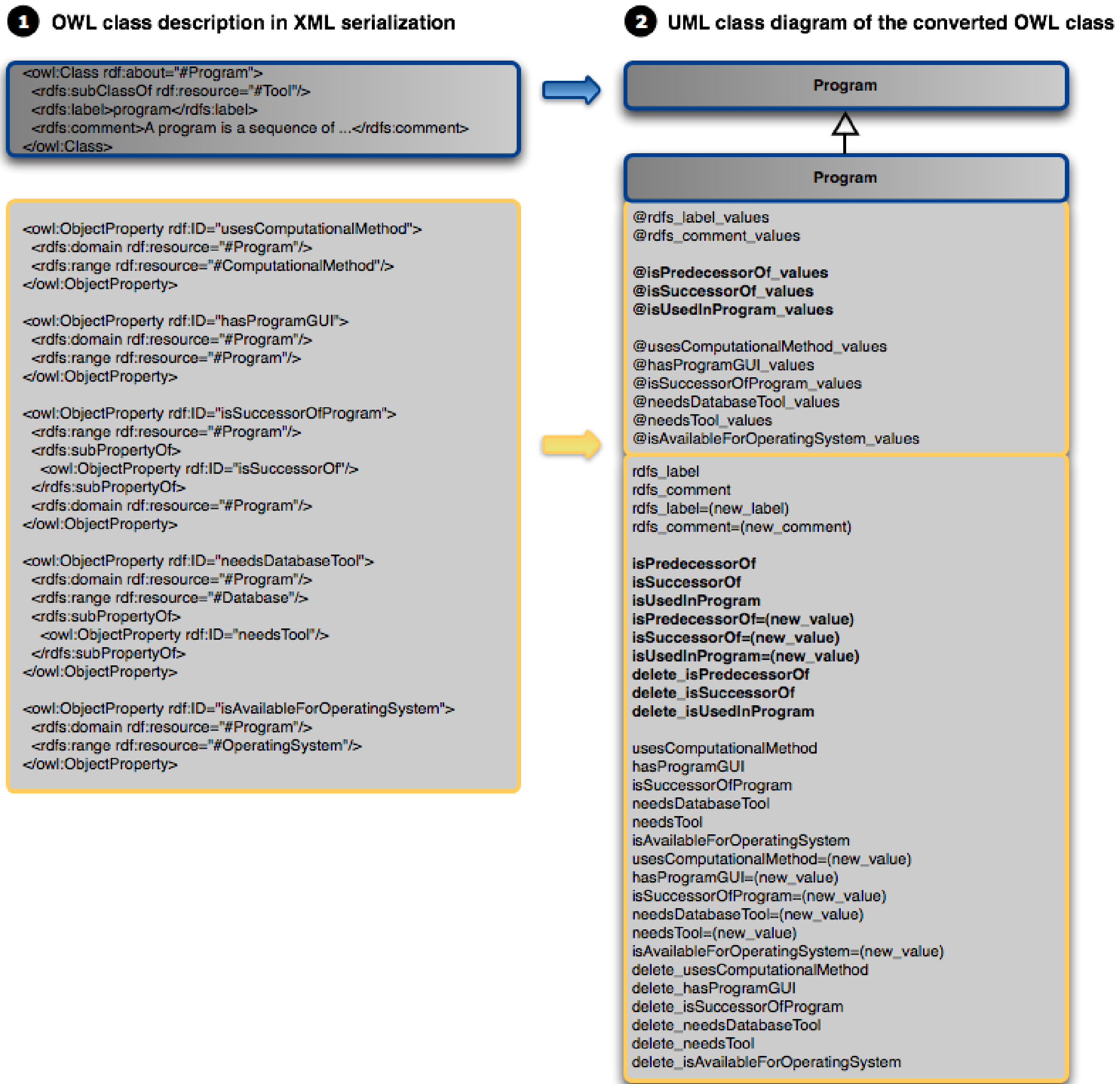


**Figure 3.1: Conversion of an OWL class *Program* into a RUBY class.** Schematic illustration of the conversion of an OWL class **(1)** into an object-oriented RUBY representation **(2)**. The RUBY class is shown here in UML notation. The URI definition of the OWL class as well as its superclass dependency is marked by a blue border. The corresponding conversion is indicated by a blue arrow. Attributes and instance methods that have been inherited by superclass *Tool* are shown in bold letters. OWL and UML components marked in orange relate to OWL properties, for which class *Program* is defined as OWL domain.

---

No *consistency safe* framework was published yet. Frameworks like JENA2 and OWL API aim at supporting any kind of TBox modification, therefore they cannot be *consistency safe* per definition.

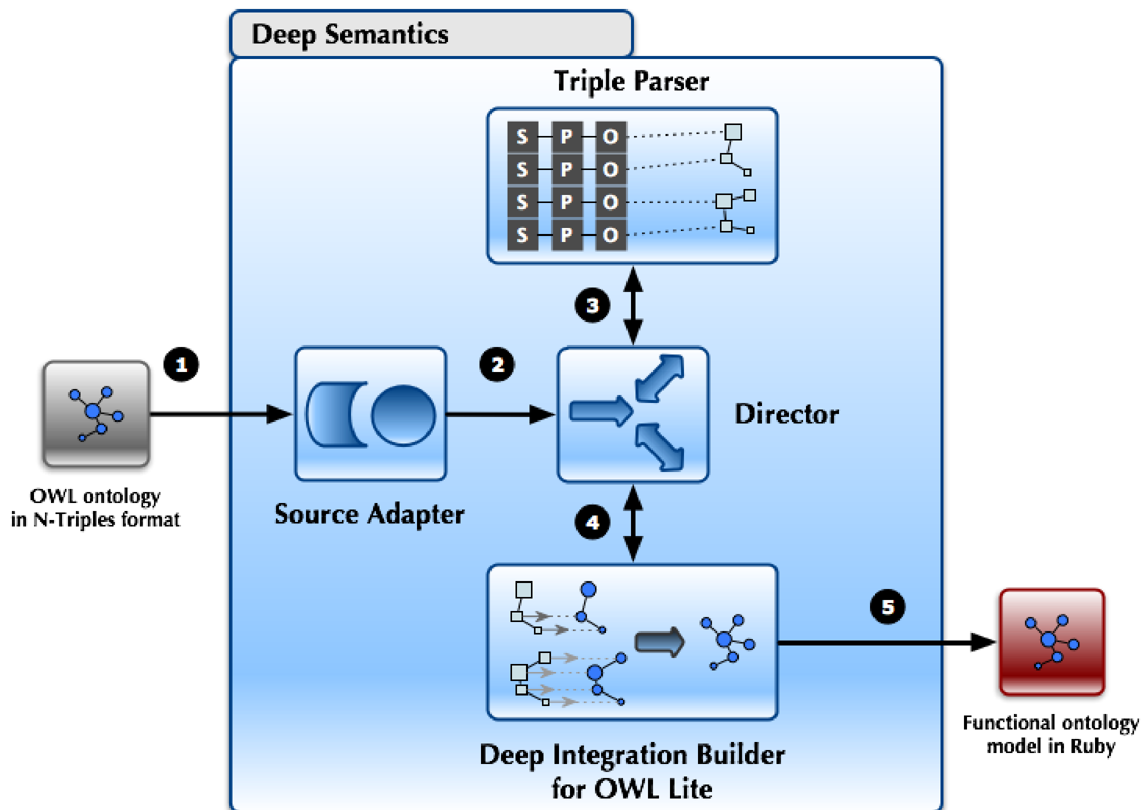


**Figure 3.2: Top-level architectural diagram of DEEP SEMANTICS including input and output values.** The director controls the deep integration process, in which ontology triples consisting of subjects (S), predicates (P) and objects (O) are mapped to RUBY code. Source adapter are used to access the ontology triples form a specified resource while triple parser is responsible for the correct translation of the triples into RUBY objects. Deep integration builder integrates these RUBY objects into a functional ontology model.

## 3.2 Architecture

DEEP SEMANTICS' architecture is based on the *Builder* design pattern (Freeman *et al.*, 2004). Using this pattern one separates the construction of a complex object from its representation so that the same construction process can create different representations. The decision to use this pattern was motivated by the initial intention to use DEEP SEMANTICS for both the development of an ontology editor (as part of the **ONTOVERSE** project) and as a framework for rapid Semantic Web development (i. e. its current use case). While its application for the ontology editor in **ONTOVERSE** was not followed up DEEP SEMANTICS has been extended up to a point where it can be successfully applied for Semantic Web development projects like **BIO2ME** or **IKEN**. The *Builder* pattern based architecture is in this context important for its future maintenance and enhancement, especially with regard to a potential transition of the DEEP SEMANTICS project from a one man development project to an open source project (see the discussion chapter on the future of DEEP SEMANTICS).

Figure 3.2 gives an overview of the architecture of DEEP SEMANTICS. It comprises the main building blocks of the framework: the *Source Adapter*, the *Director*, the *Triple Parser* and the *Deep Integration Builder*. As indicated by the rounded numbers in the figure the processing pipeline basically comprises five steps that are shortly described in the following (for a more detailed description see Section 3.4 to Section 3.6):

1. Input for DEEP SEMANTICS is always an OWL ontology in N-Triples format. This ontology can be stored in a file or a database table.

2. *Director* coordinates data flow in the processing pipeline. At first it instructs *Source Adapter* to read in the ontology triples, depending on the type of source (file or database). *Source Adapter* returns an object *triples_set* of class *TripleSet* containing normalized serializations of the read triples.

3. Then *Director* passes the set of triples to the *Triple Parser* module. *Triple Parser* converts the triples into objects of type *TemplateValues*. These template value objects are collected and returned to the *Director* according to their affiliation to the TBox or ABox of the ontology.

4. In the next step *Director* pipes these template values into the selected *Deep Integration Builder* – Figure 3.2 shows *Deep Integration Builder for* OWL LITE . *Deep Integration Builder* takes the template values and converts the corresponding ontology definition, ontology classes, properties and instances – at first all TBox constructs (classes and property axioms) and then the ABox instances. After every ontology construct is converted the final deep integration step is triggered: the assembly of the functional ontology model in RUBY.

5. A functional ontology model in RUBY is the final result of the DEEP SEMANTICS processing pipeline. This functional ontology model can then be applied in RUBY programs to access and operate on integrated ontologies.

As mentioned above this architecture was designed to build different implementations of its subparts without having to re-implement the whole system and to be able to provide alternative versions for every subpart. For the current version of DEEP SEMANTICS this means that two different *Source Adapter* implementations are available – for N-Triples files and N-Triples stored in a database table. A potentially additional version of the *Source Adapter* could be for example an adapter for RDF/XML serialized ontologies.

Concerning the *Deep Integration Builder* a version for OWL DL ontologies is already prepared up to the parsing of the corresponding triple sets. What is missing is the conversion of ontology classes and properties into RUBY classes and objects, respectively. The final implementation of a *Deep Integration Builder for* OWL DL however is not subject of this thesis, but will be discussed in Section 3.9.

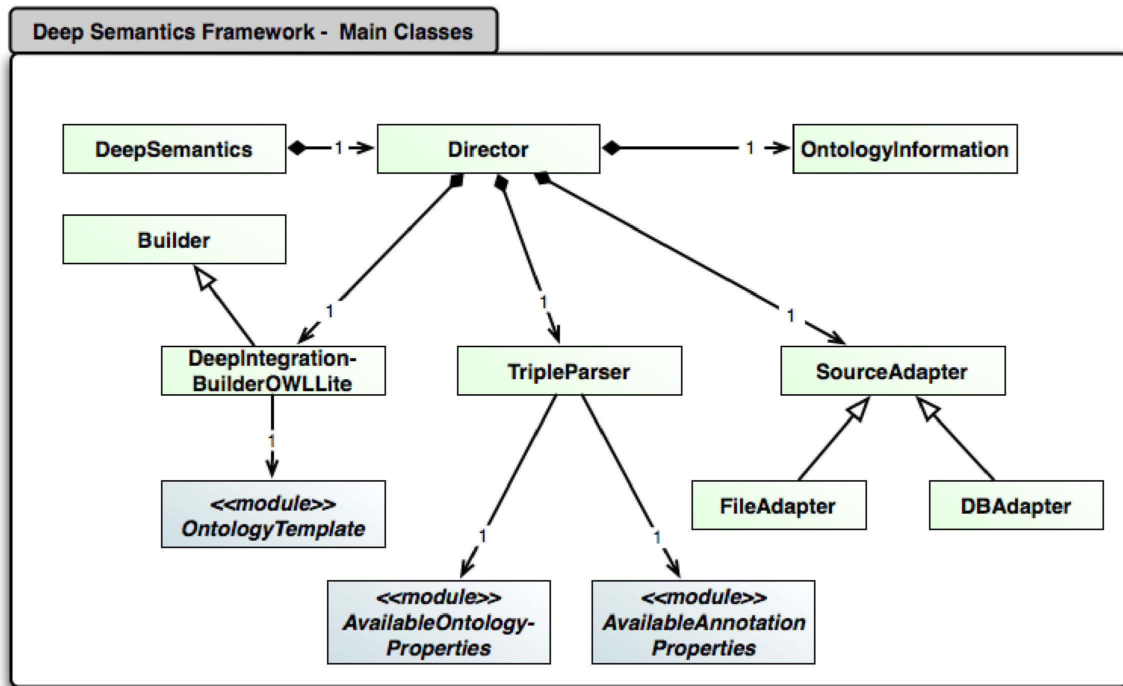


**Figure 3.3: UML class diagram of the DEEP SEMANTICS main classes.** Green rectangles indicate RUBY classes while blue ones represent RUBY modules. The central class in DEEP SEMANTICS is *Director*. This class is related to *SourceAdapter*, *TripleParser*, *OntologyInformation* and *DeepIntegrationBuilderOWLLite* as well as *DeepSemantics*.

---

### 3.2.1 Implemented RUBY classes and modules

While Figure 3.2 shows a top-level view of the architecture this subsection covers the details on the RUBY classes and modules DEEP SEMANTICS is consisting of. Figure 3.3 is an UML class diagram of the main classes of DEEP SEMANTICS. The pivotal class is *DeepSemantics*. Its main function is to generate an object of type *Director* using the method *set_director(director_options)*. Setting up *Director* one can specify which kind of ontology adapter should be used, what the access path to the ontology is and which kind of builder should be used. As indicated by the arrow connecting *Director* and *DeepSemantics* class *Director* is compositionally integrated into class *DeepSemantics* – the term "compositionally" refers to a *composition relation* in UML class diagrams between a container class and a contained class.

As it is the main functionality of *Director* to coordinate the flow and processing steps of ontology data this class compositionally integrates exactly one of the following classes *SourceAdapter*, *TripleParser*, *OntologyInformation* and *DeepIntegrationBuilderOWLLite*. As described before instances of *SourceAdapter* are used to access the respective ontology data source. It currently has two subclasses *FileAdapter* and *DBAdapter*. *OntologyInformation* is required to save the corresponding adapter, builder, source path and triple parser for one ontology. It supports the application of DEEP SEMANTICS together with two or more ontologies.

The task of *TripleParser* is to transform ontology triples into template values. To accomplish this task it uses the modules *AvailableOntologyProperties* and *AvailableAnnotationProperties*

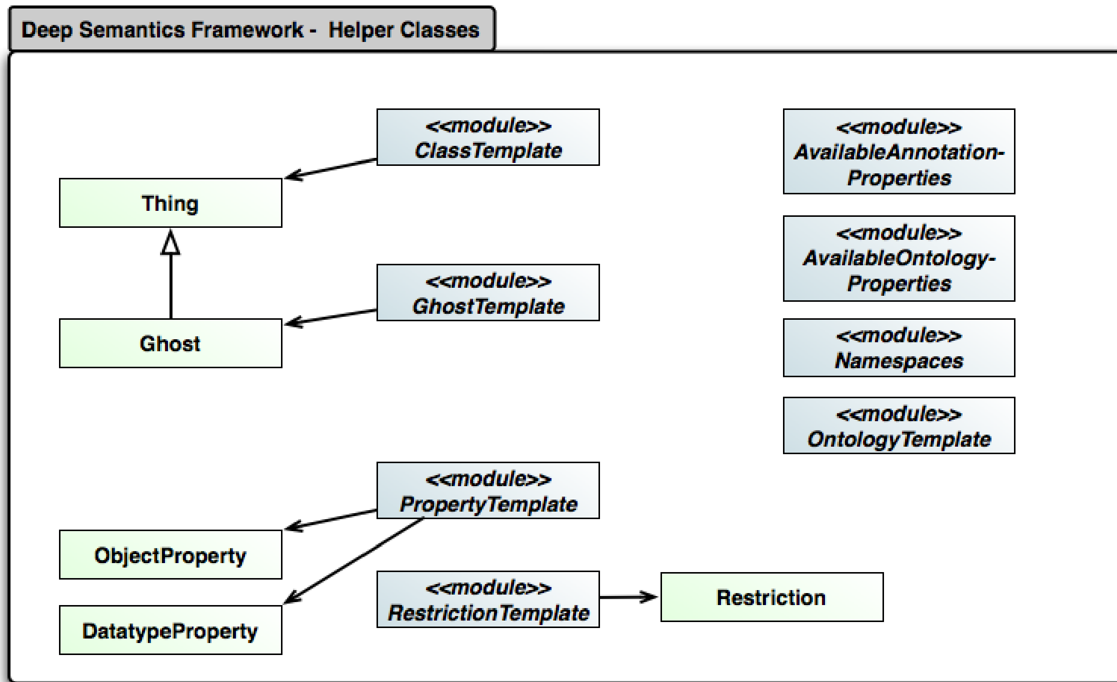


**Figure 3.4: UML class diagram of the DEEP SEMANTICS helper classes.** Classes *Thing*, *Ghost*, *Restriction*, *DatatypeProperty* and *ObjectProperty* directly relate to corresponding ontology constructs. Green rectangles indicate RUBY classes while blue ones represent RUBY modules.

---

that provide lists of the ontology and annotation properties that have to be considered. *DeepIntegrationBuilderOWLLite* is a subclass of *Builder*. Its task is to use the template values to build up a functional model of the ontology in RUBY. The module *OntologyTemplate* is utilized by *DeepIntegrationBuilderOWLLite* to assemble the ontology model.

The class diagram in Figure 3.4 models the helper classes I designed for DEEP SEMANTICS. Beside the classes *AvailableOntologyProperties* and *AvailableAnnotationProperties* already mentioned before the helper classes and modules are directly related to the converted ontology classes, datatype properties and object properties as well as namespaces. Converted ontology classes all inherit class *Thing*. This allows to easily check if a RUBY object belongs to the ontology or not. Module *ClassTemplate* used to generate the ontology RUBY classes makes use of *Thing*. Likewise module *GhostTemplate* generates so called *Ghost* classes. Class *Ghost* is used for classes that are not explicitly defined in the ontology but have to be created for the deep integration of instances that belong to more than one type of class and which additionally are not in a hierarchical relation to each other. Class *Ghost* is logically equal to the intersection of these classes concerning constraints and inheritance.

Module *OntologyTemplate* is of special importance as it is directly used to generate the RUBY module that represents both the namespace and the enclosing construct of the functional ontology model. Unlike ontology classes, datatype properties and object properties are converted into RUBY objects of type *DatatypeProperty* and *ObjectProperty*, respectively. These conversions are controlled by the module *PropertyTemplate*. One crucial processing step during deep integration is the consideration of restrictions and constraints, respectively,

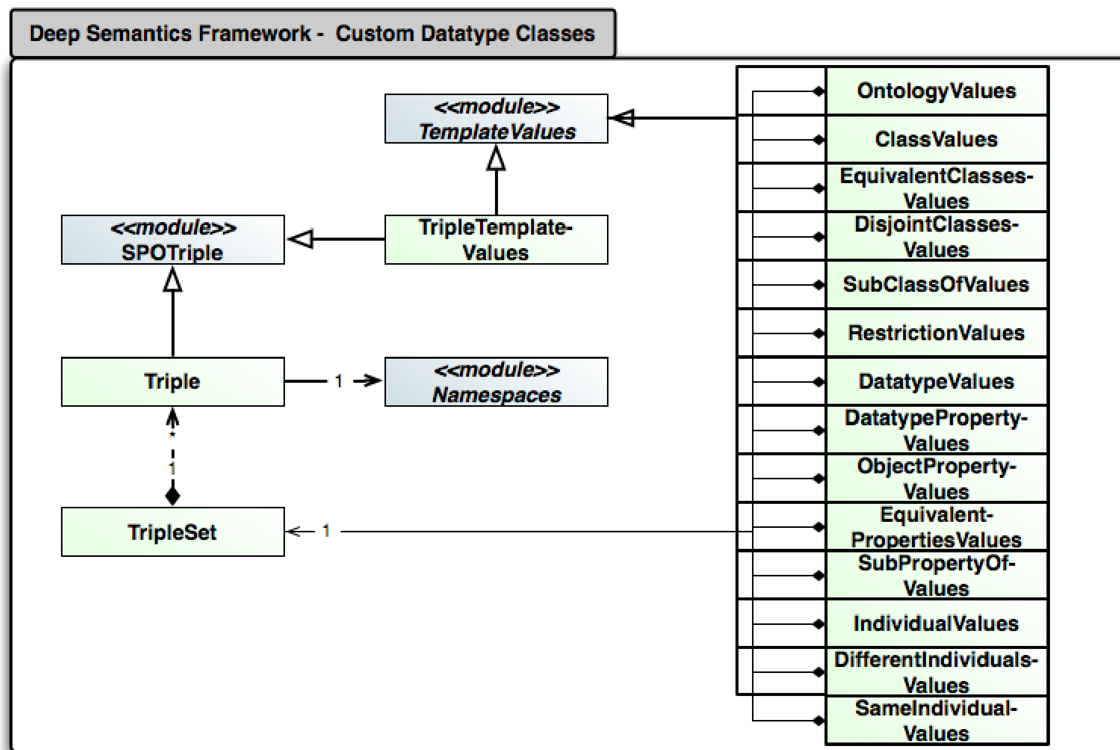


**Figure 3.5: UML class diagram of DEEP SEMANTICS custom datatypes.** Of special importance are all classes that inherit the module *TemplateValues*. They are used to save the specific values for the diverse ontology constructs. Green rectangles indicate RUBY classes while blue ones represent RUBY modules.

---

for the generation of appropriate ontology RUBY classes. Module *RestrictionTemplate* coordinates the generation of RUBY instances of *Restriction* that are used as base material for the corresponding code generation (in particular for the generation of instance *setter* methods). The last module shown in Figure 3.4 that has not yet been described here is *Namespaces*. *Namespaces* stores every namespace that is used in the processed ontology. While a new namespace can be added to the framework using the method *add_namespace*, per default the namespaces *"http://www.w3.org/1999/02/22-rdf-syntax-ns#"*, *"http://www.w3.org/2000/01/rdf-schema#"*, *"http://www.w3.org/2001/XMLSchema#"* and *"http://www.w3.org/2002/07/owl#"* are stored in *Namespaces*.

Like similar sophisticated software frameworks DEEP SEMANTICS requires a variety of custom datatypes. Figure 3.5 shows the UML class diagram of these custom datatypes. Of special importance are all classes that inherit the module *TemplateValues*. They are used to save the specific values for the diverse ontology constructs. These values are used together with corresponding templates like *PropertyTemplate* or *ClassTemplate* to produce their deep-integrated counterparts (ontology classes and properties in RUBY). In the following classes that inherit *TemplateValues* are listed:

- *OntologyValues*: Predominantly stores information about the URI of the ontology as well as its annotations.

- *ClassValues*: Objects of this type store information about the corresponding classes' URI, annotations and superclasses as well as a boolean value *TRUE* if the class is deprecated.
- *EquivalentClassesValues*: This class includes information about a set of equivalent classes.
- *DisjointClassesValues*: Analogous to *EquivalentClassesValues* but storing a set of disjoint classes.
- *SubClassOfValues*: Besides directly storing subclass-superclass relation information as part of *ClassValues* objects, OWL allows the definition of stand-alone subclass-of statements. These are considered in DEEP SEMANTICS using *SubClassOfValues*.
- *RestrictionValues*: This class records information about a restriction like the type of the restriction (for example *"all-values-from"*) and the property the restriction acts on.
- *DatatypeValues*: Primarily stores information about the URI of the datatype as well as its annotations.
- *DatatypePropertyValues*: Objects of this type store information about the corresponding properties' URI, annotations, ranges, domains and super-properties as well as boolean values *TRUE* if the class is deprecated or functional.
- *ObjectPropertyValues*: Similar to *DatatypePropertyValues* above *ObjectPropertyValues* additionally records whether the object property is inverse functional, transitive, symmetric or has an inverse property.
- *EquivalentPropertiesValues*: Analogous to *EquivalentClassesValues* but storing a set of equivalent datatype properties or object properties.
- *SubPropertyOfValues*: Like *SubClassOfValues* but for sub-property/super-property relations.
- *IndividualValues*: *IndividualValues* is the first ABox related class in this list. It stores the URI of the described instance or individual, respectively, as well as its related annotations, type affiliations and property/value pairs (attributes).
- *DifferentIndividualsValues*: Also part of the ABox *DifferentIndividualsValues* stores a set of individuals that are explicitly stated as being different.
- *SameIndividualsValues*: Analogous to *DifferentIndividualsValues* but stores a set of equal individuals.

Furthermore, Figure 3.5 comprises the module *SPOTriple* as well as class *Triple* which inherits this module. *SPOTriple* includes regular expressions for the identification of blank nodes, literals and URI nodes as well as the methods *hasBNodeSubject?* and *hasBNodeObject?*. The more specialized class *Triple* adds the methods *hasLiteralObject?* and *normalize*. Objects of type *Triple* are related to *TripleSet*. *TripleSet* contains methods to retrieve certain triples (for example *find_by_S(subject)* to fetch triples that include the subject passes as parameter).

## 3.3 The Abstract Syntax of OWL LITE and its contribution to DEEP SEMANTICS

Theoretical foundations of OWL comprise definitions of its abstract syntax and a mapping of RDF triples to this abstract syntax (Peter *et al.*, 2004). In my thesis I used this abstract syntax in combination with the mentioned mappings as a theoretical fundament for the development of DEEP SEMANTICS. Figure 6.1 on page 140 in the appendix shows the complete abstract syntax of OWL LITE. A list of the corresponding mapping rules can be found in Peter *et al.* (2004).

Figure 3.6 illustrates the relations of OWL LITE constructs in Extended Backus-Naur Form (EBNF) as defined in the official abstract syntax to the static and dynamically generated classes and modules I have designed for DEEP SEMANTICS (a not simplified version of Figure 3.6 can be found on page 141 in the appendix as Figure 6.2). *'Ontology()'* is the enclosing construct of every OWL LITE ontology. Correspondingly is its DEEP SEMANTICS counterpart a RUBY module that comprises all class, property and instance ontology constructs. As indicated in Figure 3.6 by a blue rectangle and the symbol *"?OntologyLocalName?"* the RUBY module representing the ontology construct is dynamically generated by DEEP SEMANTICS. One can see that the EBNF construct *'Ontology(' [* ***ontologyID*** *]* ***directive*** *')'* is modeled including zero or more so called "directives" (that means ontology annotations, class axioms, property axioms or instances). Analogously the module *"?OntologyLocalName?"* comprises zero or more objects of type *Restriction*, *DatatypeProperty* and *ObjectProperty*.

Additionally, *"?OntologyLocalName?"* includes, and acts as the namespace of, zero or more dynamically generated classes indicated by symbols *?ClassLocalName?* and *?GhostName?* in Figure 6.2. *?ClassLocalName?* indicates that the corresponding meta-programmed RUBY classes are named like the local name of the equivalent ontology class (that means like the local name part of classes URI – indicated in related EBNF fragment as *classID*). While *?ClassLocalName?* RUBY classes directly relate to their equivalent ontology classes *?GhostName?* don't have such direct mappings. I have added *?GhostName?* classes – I call this type of classes *Ghost* classes – to DEEP SEMANTICS to be able to generate instances that belong directly to more than one ontology class (i.e. multiple inheritance). In RUBY each object is instance of exactly one RUBY class. While this RUBY class can itself inherit one superclass and zero or more RUBY modules, a RUBY class instance object cannot be declared to be instance of more than one RUBY class. Therefore every ontology instance that belongs to a set of more than one distinct ontology class belongs to a *Ghost* class specifically meta-programmed by DEEP SEMANTICS.

Figure 3.7 shows the relations between the abstract syntax of the *'Ontology()'* construct and its realization in DEEP SEMANTICS. The *'Ontology()'* construct can have an optional identifier in the form of an URI reference and zero or more ***directive*** extensions. The local name part of the ontology's URI is used by DEEP SEMANTICS as name of the automatically generated RUBY module (indicated by the symbol *"?OntologyLocalName?"*).

A ***directive*** can be an ontology annotation (for example ontology properties *owl:imports*, *owl:priorVersion*, *owl:backwardCompatibleWith* and *owl:incompatibleWith*), an ***axiom*** or a ***fact***. Facts are ontology individuals as well as statements about instance equality or disjointness. In DEEP SEMANTICS I consider facts during the generation of ontology instances by

**Figure 3.6: Simplified UML class diagram extended with meta-programming symbols.** Illustrated are the relations of OWL LITE constructs in Extended Backus-Naur Form (EBNF) as defined in the official abstract syntax to the static and dynamically generated classes and modules

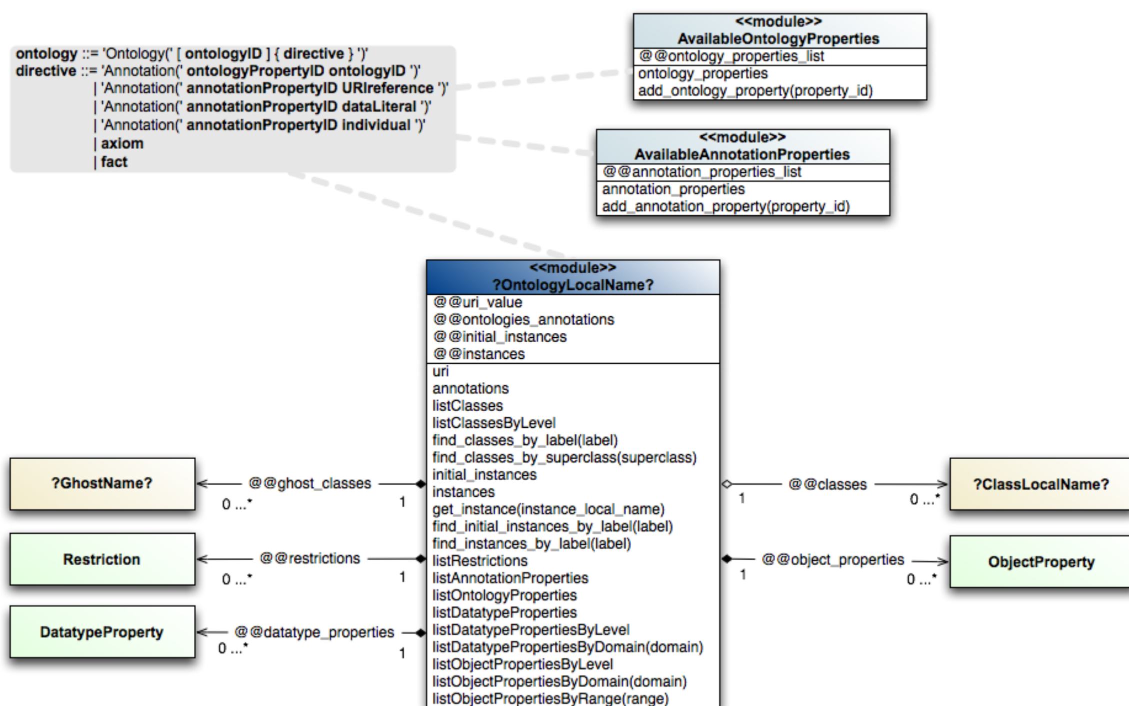


**Figure 3.7: Simplified UML class diagram for the *Ontology* construct extended with meta-programming symbols.** EBNF of OWL LITE ontology construct in relation to its DEEP SEMANTICS classes and modules counterparts.

---

instantiating the corresponding *?ClassLocalName?* classes or *Ghost* classes after the deep integration process. As all the figures (e. g. Figure 6.2 and Figure 3.7) containing the mappings between the abstract syntax and DEEP SEMANTICS relate to RUBY classes or modules, and a ***fact*** basically relate to a RUBY instance, facts are not covered in more detail in this section but later in Subsection 3.6.5 on page 64.

*'Annotation()'* directives in EBNF are considered in DEEP SEMANTICS using the modules *AvailableOntologyProperties* and *AvailableAnnotationProperties*. By calling *add_annotation_property(property_id)* or *add_ontology_property(property_id)* one can add custom annotation properties to the functional ontology model.

The EBNF nonterminal ***axiom*** represents either class axioms or property axioms. Figure 3.7 indicates the modality of the relation between an *"?OntologyLocalName?"* module and ***axiom*** related *?ClassLocalName?*, *?GhostName?*, *ObjectProperty*, *DatatypeProperty* and *Restriction* RUBY classes by displaying the corresponding module variables (*"@@classes"*, *"@@ghost_classes"*, *"@@object_properties"*, *"@@datatype_properties"* and *"@@restrictions"*).

The following list recapitulates the focussed DEEP SEMANTICS modules and classes included in Figure 3.7:

- *«module» ?OntologyLocalName?*: Directly maps to the EBNF nonterminal ***ontology*** in the abstract syntax. The module is dynamically generated via meta-programming by

DEEP SEMANTICS. The modules variables are @@*uri_value* (saves the URI of the ontology), @@*ontologies_annotations* (the annotations concerning the ontology), @@*initial_instances* (an array comprising all instances that have been initially generated during start-up), @@*instances* (all instances in the ontology including initial instances and newly generated ones), @@*classes* (an array recording all ontology classes), @@*ghost_classes* (an array recording all *Ghost* classes), @@*object_properties* (an array recording all object properties), @@*datatype_properties* (an array recording all datatype properties) and @@*restrictions* (all objects of type *Restriction*). Additionally the module includes a variety of module methods of which some are representatively described here:

  - *listClasses*: The method returns all ontology classes. Similar methods exist for properties and instances.
  - *find_classes_by_label(label)*: Returns classes that have a certain rdfs:label which is passed as argument.
  - *listObjectPropertiesByLevel*: This method returns a hash which keys indicate the object property hierarchy level. The hash values are arrays recording the ontology classes belonging to the current hierarchy level.

- *«module» AvailableOntologyProperties*: This module includes a list of all valid ontology annotation properties in the module variable @@*ontology_properties_list*. This list can be extended using *add_ontology_property(property_id)* or retrieved using *ontology_properties*.

- *«module» AvailableAnnotationProperties*: Analogous to *AvailableOntologyProperties* this module include a list of all valid annotation properties (except ontology annotation properties) in the module variable @@*annotation_properties_list*. This list can be extended using *add_annotation_property(property_id)* or retrieved using *annotation_properties*().

Figure 3.8 contains the simplified UML class diagram (extended with meta-programming symbols) for OWL class constructs in DEEP SEMANTICS as well as the EBNF of OWL LITE class constructs. *?ClassLocalName?* ontology classes are dynamically generated during the deep integration process. The EBNF building block *'Class()'* defines a named OWL class and consists of an URI reference as its ID, an optional flag *'Deprecated'* defining if this definition is deprecated, a nonterminal ***modality*** statement, zero or more ***annotation*** constructs as well as zero or more superclass relations (indicated by the nonterminal ***super***). As indicated by *"?ClassLocalName?"* the name of the ontology class representing RUBY class is derived from the local name part of the class URI.

In OWL LITE class axioms are used to state that a class is exactly equivalent to, for the modality *'complete'*, or a subclass of for the modality *'partial'*, of a collection of superclasses and OWL LITE restrictions. Both, URI identifiable OWL classes as well as OWL LITE restrictions, can be used as superclasses in OWL LITE.

The abstract syntax of class axioms is conceptional incorporated into the DEEP SEMANTICS implementation as follows: Concerning the mapping of OWL abstract syntax onto RUBY code

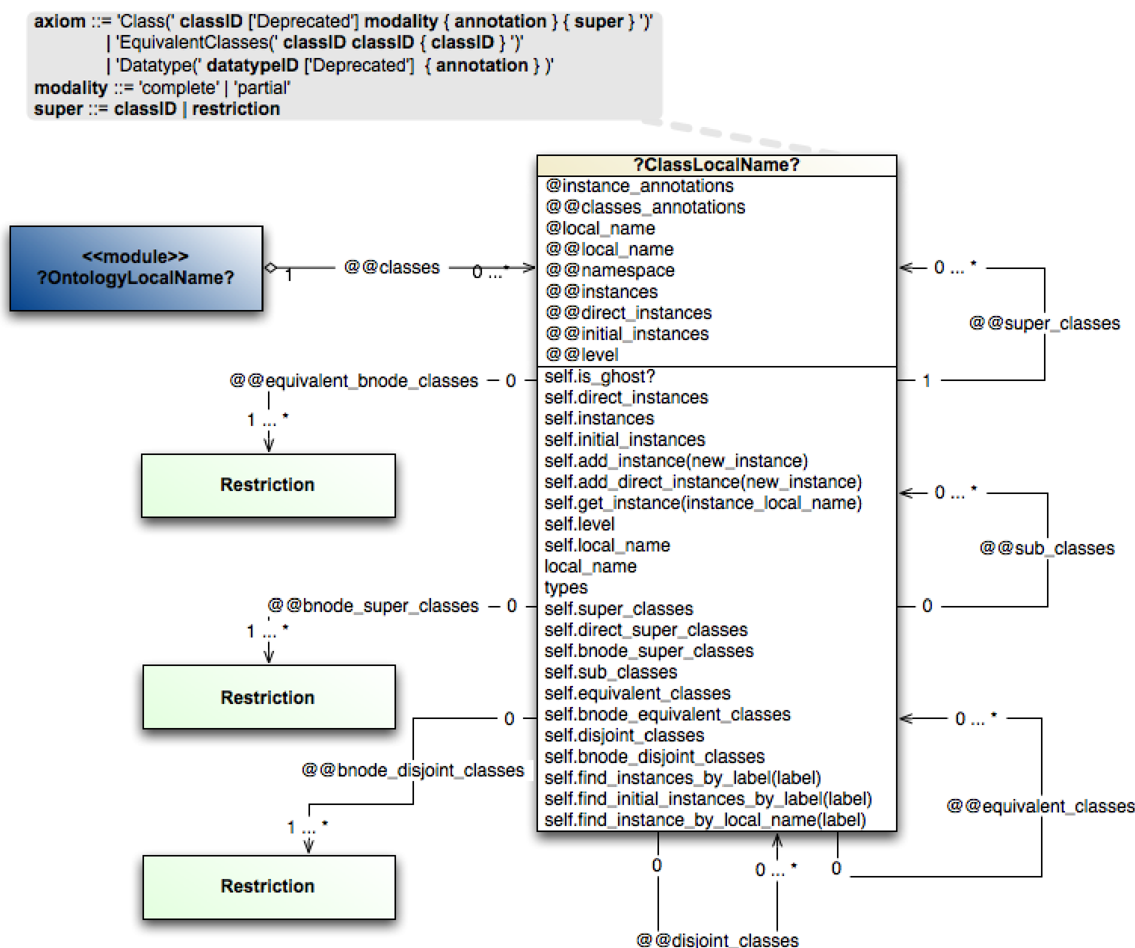


**Figure 3.8: Simplified UML class diagram for OWL class constructs extended with meta-programming symbols.** EBNF of OWL LITE class constructs in relation to its DEEP SEMANTICS class counterpart.

---

the ***modality*** of a class axiom has no effect whatsoever. In the case of the superclass being URI-identifiable OWL classes, a reference to the corresponding class object is saved in a RUBY class variable (indicated by the *@@super_classes* relation in Figure 3.8). If the superclass is a ***restriction*** (valid restrictions in OWL LITE are: all-values-from, some-values-from, minimum cardinality, maximum cardinality and cardinality) then DEEP SEMANTICS alters the property associated *setter* methods for this restriction. If for example an object property has a maximum cardinality of one then the implementation of the corresponding instance *setter* method has to be extended in way that it is guaranteed that at no time more than one value for this property is stored. Restrictions are also called B-Node classes in OWL with "B" being an abbreviation for "blank". B-Node classes are classes without an URI. Therefore, superclass relations to B-Nodes are saved in the class variable *@@bnode_super_classes* of *"?ClassLocalName?"*.

*'EquivalentClasses()'* axioms in EBNF state that two or more URI-identifiable OWL classes are equivalent. Concerning the implementation of DEEP SEMANTICS *'EquivalentClasses()'* axioms are used in order to generate exactly one RUBY module that represents the functional extension of all stated equivalent classes. This module is then inherited in every RUBY class

representation part of the equivalent classes set. In the deep-integrated model of the ontology these equivalent classes relate to each other using @@*equivalent_classes* as indicated in Figure 3.8. Equivalent B-Node classes constitute a special case. Although the EBNF of the abstract syntax does imply that only named ontology classes can be elements of *'EquivalentClasses()'* sets, restrictions are actually allowed as equivalent classes (in Figure 3.8 this is indicated by the @@*equivalent_bnode_classes* class variable).

OWL LITE does not support the definition of disjointness between ontology classes. However, DEEP SEMANTICS does because firstly, the inferred models produced by reasoners like PELLET (Parsia & Sirin, 2004) (which are used as input for DEEP SEMANTICS) include explicit class disjointness statements and secondly, because disjoint class information is crucial to enable DEEP SEMANTICS to be consistency safe. I define "consistency safeness" as the following particular property ontology editing framework: an ontology editing framework is "consistency safe" if it is not possible to produce any logical inconsistencies as defined by the corresponding ontology language. *"?ClassLocalName?"* classes provide information about and relations to, respectively, disjoint classes using @@*disjoint_classes* and @@*bnode_disjoint_classes* (see Figure 3.8).

Recapitulating UML class *"?ClassLocalName?"*:

- Instance variables:
  - @*instance_annotations*: Is an array storing the valid annotation properties. Per default these are at least "rdfs_label", "rdfs_comment" and for **IKEN** "iken_primaryLabel", "iken_isInitialIndividual".
  - @*local_name*: Saves the local name of the instance.
- Class variables:
  - @@*classes_annotations*: Variable holding all used annotation properties for the current OWL Class.
  - @@*local_name*: Saves the local name of the class.
  - @@*namespace*: The namespace the class belongs to.
  - @@*direct_instances*: An array, which contains only those instances that have been created with the *initialize()* method of this class.
  - @@*initial_instances*: An array which contains the initial instances of this class – that are those instances that were in the ontology before deep integration and the start of DEEP SEMANTICS, respectively.
  - @@*instances*: An array which contains all instances of the class.
  - @@*level*: Indicates the classes' hierarchy level (direct subclasses of owl:Thing are of level zero).
  - @@*sub_classes*: An array containing all subclasses of the class.
  - @@*super_classes*: An array containing all superclasses of the class.

- @@*equivalent_classes*: An array containing all classes that are equivalent to the current class.
- @@*disjoint_classes*: An array containing all classes that are disjoint with the current class.
- @@*bnode_disjoint_classes*: An array containing all B-Node classes that are disjoint with the current class.
- @@*equivalent_bnode_classes*: An array containing all B-Node classes that are equivalent to the current class.
- @@*bnode_super_classes*: An array containing all B-Node superclasses of this class.

- Instance methods:
  - *local_name*: Returns the value of @*local_name*.
  - *types*: Returns an array with one element (self.class) to be consistent with the *GhostTemplate types* method.
- An abstract of the most important class methods:
  - *self.is_ghost?*: Returns *FALSE* for *"?ClassLocalName?"* classes.
  - *self.instances*: Returns @@*instances* – that is all instances of the class.
  - *self.add_instance(new_instance)*: A *setter* method that can be used to add a new instance to @@*instances*.
  - *self.local_name*: Returns the value of @@*local_name*.
  - *self.super_classes*: *getter* method that returns @@*super_classes*.
  - *self.sub_classes*: *getter* method that returns @@*sub_classes*.
  - *self.find_instances_by_label(label)*: A convenient method that determines all instances of class that have the passed label as stored rdfs:label.

A *'Datatype()'* axiom defines how to describe custom data types in OWL. However, as DEEP SEMANTICS currently does not support the definition of new data types this construct is not further described in this context.

While the abstract syntax of restrictions as shown in Figure 3.9 in EBNF is quite complex the related DEEP SEMANTICS class is not. It simply contains the RUBY instance variables @*local_name*, @*on_property* and @*restriction_type*. Like it was the case for the already described ontology constructs (e.g. *?ClassLocalName?*) the @*local_name* variable of *Restriction* relates to the local name part of the class URI, too.

The variable @*on_property* saves the reference to the datatype property or object property on which the restriction is defined. In the related EBNF description *datavaluedPropertyID* and *individualvaluedPropertyID* model the value stored in @*on_property*. Variable @*restriction_type* is a hash which keys can be one of these strings *"all_values_from"*,

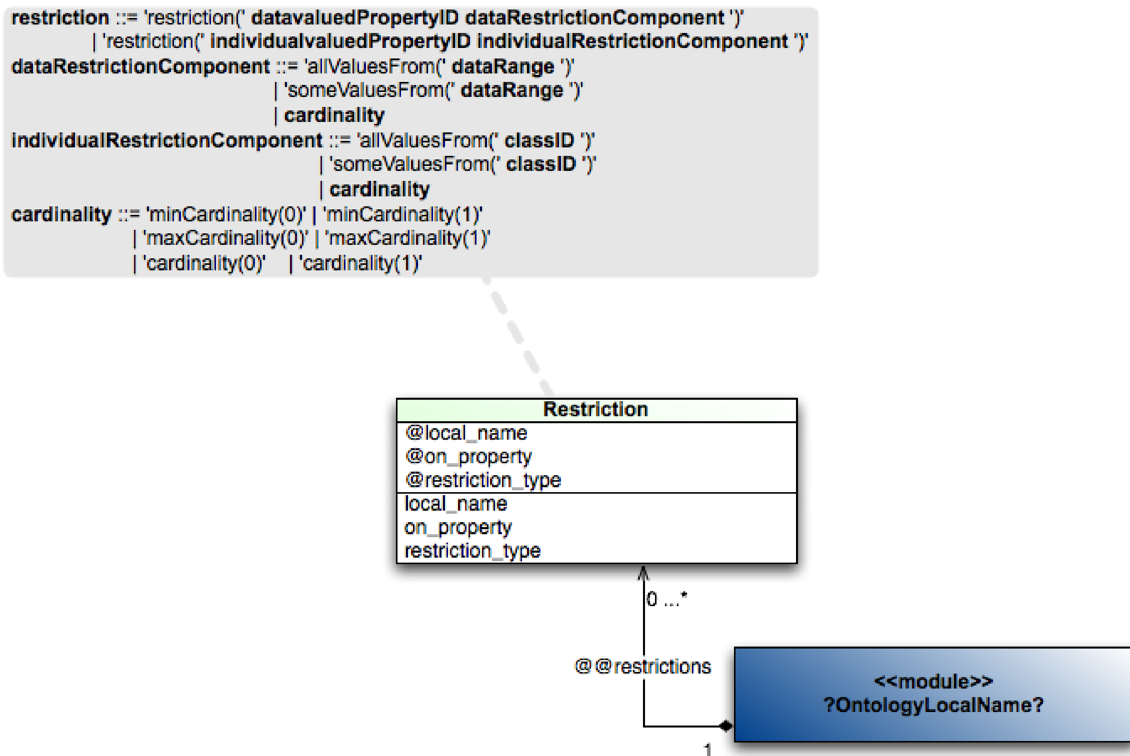


**Figure 3.9: Simplified UML class diagram for the OWL restriction constructs extended with meta-programming symbols.** EBNF of OWL LITE restriction constructs in relation to its DEEP SEMANTICS class counterpart.

---

*"some_values_from"*, *"min_cardinality"*, *"max_cardinality"* and *"cardinality"*. Like defined in the abstract syntax the values of *"all_values_from"* and *"some_values_from"*, respectively, can be either a named ontology class (in the case of object properties) or a datatype (in the case of datatype properties). The valid values for the keys respectively restriction types *"min_cardinality"*, *"max_cardinality"* and *"cardinality"* are only zero and one.

EBNF property axioms define one of the following constructs: *'DatatypeProperty()'*, *'ObjectProperty()'*, *'AnnotationProperty()'*, *'OntologyProperty()'*, *'EquivalentProperties()'* or *'SubPropertyOf()'*. Figure 3.10 shows the relation of *'DatatypeProperty()'*, *'EquivalentProperties()'* and *'SubPropertyOf()'* abstract syntax with DEEP SEMANTICS *DatatypeProperty* class. Each OWL datatype property (and object property) is converted into an instance of *DatatypeProperty* (respectively *ObjectProperty*).

*'EquivalentProperties()'* define sets of equivalent properties – in this case datatype properties. These equivalent properties sets are considered in DEEP SEMANTICS using instance variable *@equivalent_properties* in RUBY class *DatatypeProperty*. Akin is the set *'SubPropertyOf()'* of super-properties incorporated into *DatatypeProperty* with *@super_properties*.

The instance variable *@sub_properties* stores all sub-properties of the current datatype property. It serves as a convenient instance variable and is computed by *DeepIntegrationBuilderOWLLite*. Details about *DatatypePropery*:

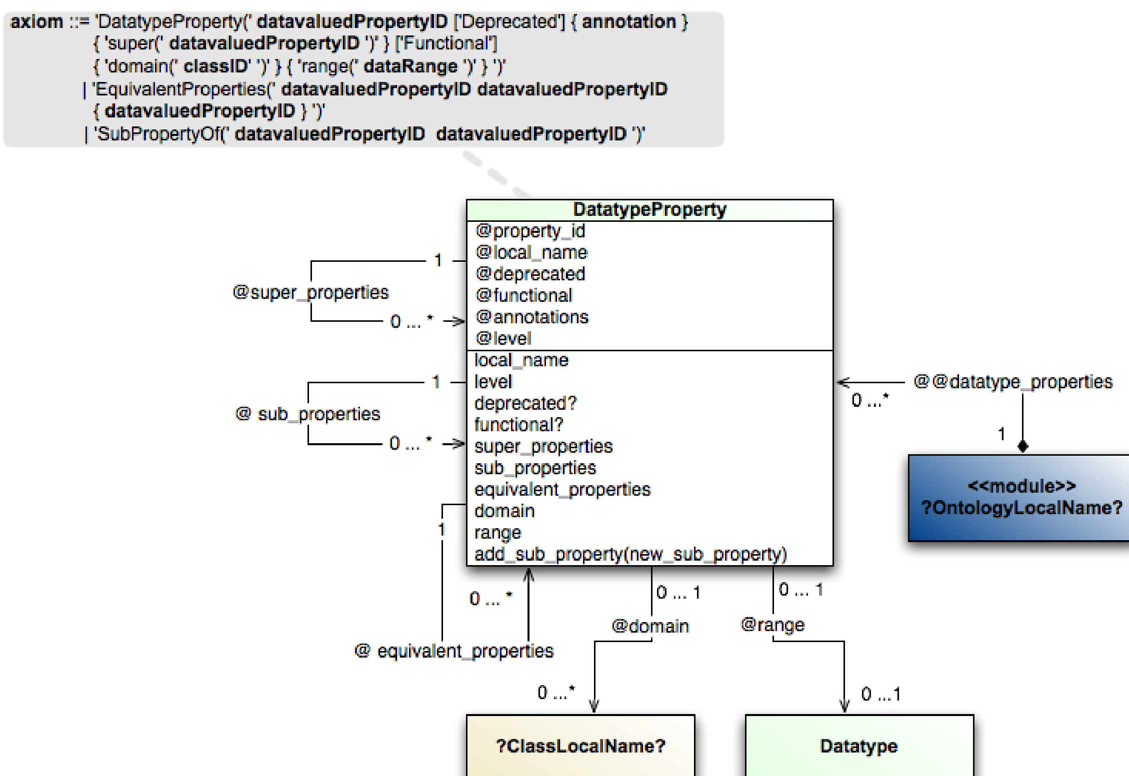


**Figure 3.10: Simplified UML class diagram for the OWL datatype property constructs extended with meta-programming symbols.** EBNF of OWL LITE datatype property constructs in relation to its DEEP SEMANTICS class counterpart.

- Instance variables:
  - *@property_id*: A DEEP SEMANTICS internal integer value uniquely identifying every instance of *DatatypeProperty*.
  - *@local_name*: Saves the local name of the instance.
  - *@deprecated*: Boolean value that indicates whether the property is deprecated.
  - *@functional*: Boolean value that indicates if the property is functional.
  - *@annotations*: Is either boolean false if there are no annotations defined for the current property or an array holding all used annotation properties for the current OWL datatype property.
  - *@level*: Indicates the property's hierarchy level.
  - *@super_properties*: An array containing all super-properties.
  - *@sub_properties*: An array containing all sub-properties.
  - *@equivalent_properties*: An array containing all equivalent properties.
  - *@domain*: All ontology classes that can be used as domain of this property stored in an array.

  - @*range*: All ontology classes that can be used as range of this property stored in an array.

- Instance methods:
  - *local_name*: Returns the local name of the property.
  - *level*: Returns the value of @*level*.
  - *deprecated?*: Returns **TRUE** if the property is defined as deprecated.
  - *functional?*: Return value is boolean @*functional*.
  - *super_properties*: Gives the array @*super_properties* back.
  - *sub_properties*: Returns @*sub_properties*.
  - *equivalent_properties*: Return value is @*equivalent_properties*.
  - *domain*: Gives the array @*domain* back.
  - *range*: Gives the array @*range* back.
  - *add_sub_property(new_sub_property)*: A *setter* method that can be used to add a new datatype properties to @*sub_properties*.

Similar to the just described datatype properties modeling Figure 3.11 shows the influence of OWL LITE abstract syntax onto DEEP SEMANTICS *ObjectProperty* class implementation. Like the other ontology constructs *'ObjectProperty()'* instances can be stated as being *'Deprecated'* and can have zero or more annotations (symbolized by nonterminal ***annotation*** in EBNF and by @*annotations* in the class diagram, respectively).

Multiple relations can exist to super-properties (@*super_properties*), sub-properties (@*sub_properties*), equivalent properties (@*equivalent_properties*), domain (@*domain*) and range (@*range*) classes. In comparison to Figure 3.10 Figure 3.11 differ primarily in respect of additional global constraints that can be defined on object properties. These are in EBNF notation *['inverseOf('* ***individualvaluedPropertyID*** *')']*, *'Symmetric'*, *'InverseFunctional'*, *'Functional' 'InverseFunctional'* and *'Transitive'*. As defined in the abstract syntax of OWL LITE object properties can have at maximum one inverse property *['inverseOf('* ***individualvaluedPropertyID*** *')']*. An example for an inverse property is: persons own cars, cars are owned by persons (i.e. "own" is the inverse property of "owned by"). A *'Symmetric'* property *P* is a property for which holds that if an instance *X* is related to an instance *Y* over *P* than *Y* is related over *P* to *X*, too. For example: If *"isBrotherOf"* is a symmetric object property and the statement *"Jacob" "isBrotherOf" "Wilhelm"* then it holds that *"Wilhelm" "isBrotherOf" "Jacob"*.

Additionally object properties can be defined as being *'InverseFunctional'* or both *'Functional' and 'InverseFunctional'*. If an object property is declared to be inverse-functional, then the object of a property statement uniquely determines the subject individual. As indicated by the EBNF definition in Figure 3.11 an object property can only be either functional, inverse-functional, both functional and inverse-functional or transitive. An object property cannot be for example functional and transitive as this could lead to logical inconsistencies.

Details about *ObjectProperty*:

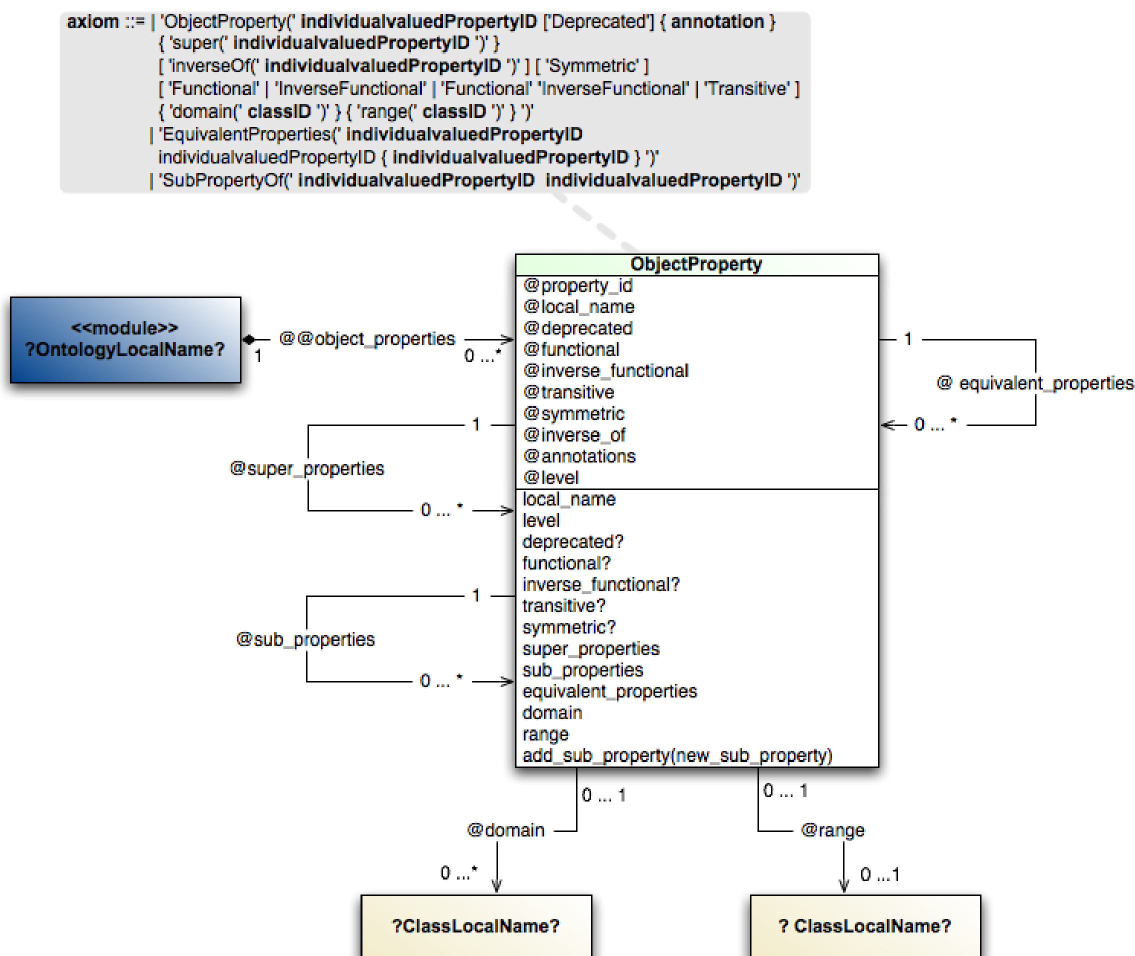


**Figure 3.11: Simplified UML class diagram for the OWL object property constructs extended with meta-programming symbols.** EBNF of OWL LITE object property constructs in relation to its DEEP SEMANTICS class counterpart.

---

- Instance variables:
  - *@property_id*: A DEEP SEMANTICS internal integer value uniquely identifying every instance of *DatatypeProperty*.
  - *@local_name*: Saves the local name of the instance.
  - *@deprecated*: Boolean value that indicates whether the property is deprecated.
  - *@functional*: Boolean value that indicates if the property is functional.
  - *@inverse_functional*: Boolean value that indicates if the property is inverse-functional.
  - *@transitive*: If the corresponding OWL object property is defined transitive than the value of *@transitive* is **TRUE**.
  - *@symmetric*: Boolean value that indicates if the property is symmetric.
  - *@inverse_of*: Stores either no or exactly one inverse property of the current property.

  - @*annotations*: Is either boolean false if there are no annotations defined for the current property or an array holding all used annotation properties for the current OWL datatype property.
  - @*level*: Indicates the property's hierarchy level.
  - @*super_properties*: An array containing all super-properties.
  - @*sub_properties*: An array containing all sub-properties.
  - @*equivalent_properties*: An array containing all equivalent properties.
  - @*domain*: All ontology classes that can be used as domain of this property stored in an array.
  - @*range*: All ontology classes that can be used as range of this property stored in an array.

- Instance methods:
  - *local_name*: Returns the local name of the property.
  - *level*: Returns the value of @*level*.
  - *deprecated?*: Returns **TRUE** if the property is defined as deprecated.
  - *functional?*: Return value is boolean @*functional*.
  - *inverse_functional?*: Gives back boolean @*inverse_functional*.
  - *transitive?*: Returns **TRUE** if the property is defined as transitive.
  - *symmetric?*: Returns the boolean value of @*symmetric*.
  - *super_properties*: Gives the array @*super_properties* back.
  - *sub_properties*: Returns @*sub_properties*.
  - *equivalent_properties*: Return value is @*equivalent_properties*.
  - *domain*: Gives the array @*domain* back.
  - *range*: Gives the array @*range* back.
  - *add_sub_property(new_sub_property)*: A *setter* method that can be used to add new object properties to @*sub_properties*.

This section gave a detailed overview about the OWL LITE abstract syntax being the foundation of the DEEP SEMANTICS class architecture. It was shown how ontology definitions are converted to RUBY modules, ontology classes to RUBY classes, OWL properties and property restrictions to RUBY objects. The next three sections describe the DEEP SEMANTICS workflows involved in the deep integrating ontology conversion process.

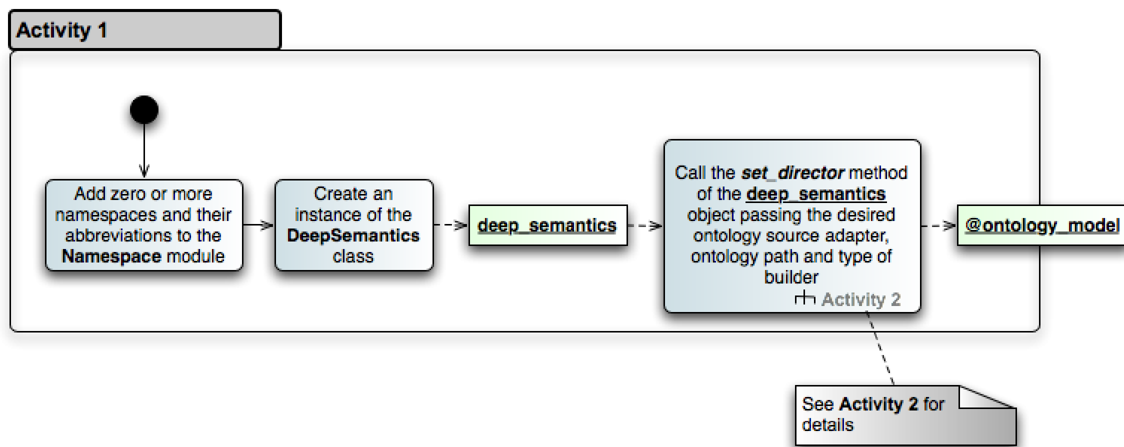


**Figure 3.12: Activity diagram 1: overview.** Creating a deep-integrated RUBY model of an input OWL LITE ontology (actions are indicated as rounded rectangles in blue; variables as not rounded ones in green).

---

## 3.4 Director: Coordinating Data Workflow and Deep Integration Process

To coordinate flow and processing steps of ontology data is the main functionality of the DEEP SEMANTICS *Director*. Figure 3.12 illustrates the process of *Director* instance generation. Before the *Director* is instantiated one can add zero or more namespaces as well as their abbreviations to the DEEP SEMANTICS *Namespace* module. In a next step an instance of *DeepSemantics* (in Figure 3.12 indicated as *deep_semantics*) is created that functions primarily as an interface for setting up of *Director*. Using *deep_semantics.set_director* and passing parameters defining the used source adapter, ontology path and ontology model builder one can then instantiate variable *@director*.

Figure 3.13 details that *@director* creates instances of *TripleParser* and *Builder* using the parameters passed. This step is further described in Figure 3.14 showing that first *@triple_parser* is instantiated. Then *@director* creates an instance of *OntologyInformation* named *@ontology_information* storing information about the current ontology. The next processing step in Figure 3.14 is then depending on whether the chosen builder is for OWL LITE or OWL DL. For the following activity diagrams we assume that *@director* creates an instance of *DeepIntegrationBuilderOWLLite* saved in *builder*. Next *@director* registers the used builder in *@ontology_information*.

After processing activity 3 the next director action shown in Figure 3.13 is the creation of an instance of *Adapter* and the import of the ontology triples using this adapter. Activity diagram 4 in Figure 3.15 illustrates that *@ontology_information* is used by *@director* to decide whether an instance of *DBAdapter* or an instance of *FileAdapter* has to be created and saved in variable *adapter*. Next *@director* registers the used *adapter* using *@ontology_information*. After that *@director* invokes the *get_triples()* method of adapter and returns the variable *triples_set* which contains a record of all ontology triples.

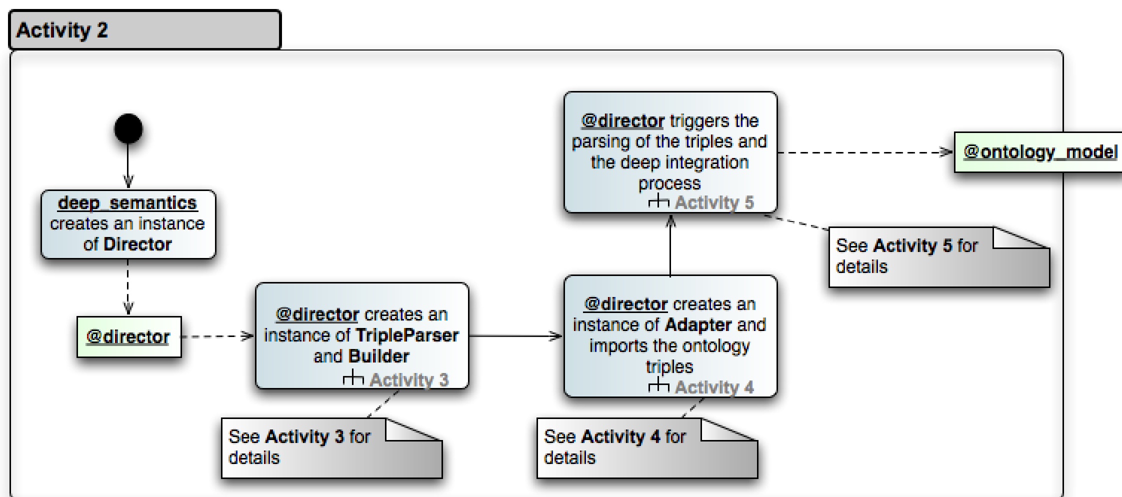


**Figure 3.13: Activity diagram 2: *Director*.** Creation of a *Director* instance that triggers the parsing and the deep integration process (actions are indicated as rounded rectangles in blue; variables as not rounded ones in green).

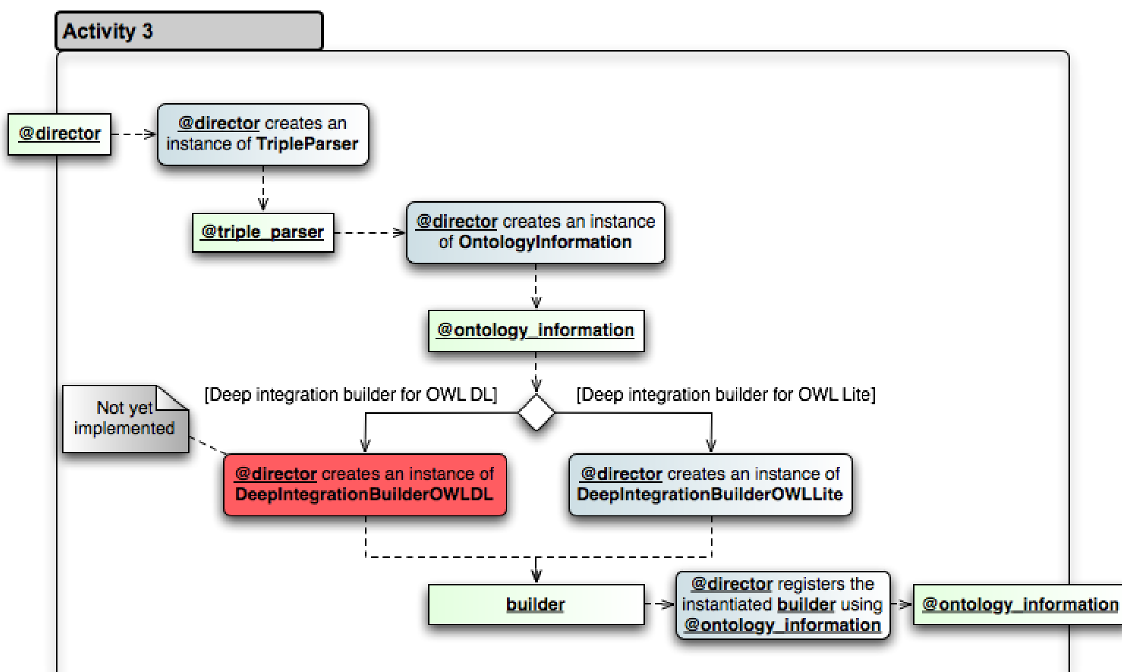


**Figure 3.14: Activity diagram 3: *TripleParser*.** Creation of a *TripleParser* and a *Builder* instances (actions are indicated as rounded rectangles in blue; variables as not rounded ones in green).

---

We get back to activity diagram 2 (Figure 3.13) and see that the next action is *@director* triggering the parsing of the triples and the deep integration process. Activity diagram 5 in Figure 3.16 comprises the details of this action: If *triples_set* contains no triple then this processing step is finished – as well as the complete ontology conversion process. Otherwise *@director* invokes the *parse_triples_set()* method of *@triple_parser* passing *triples_set* as in-

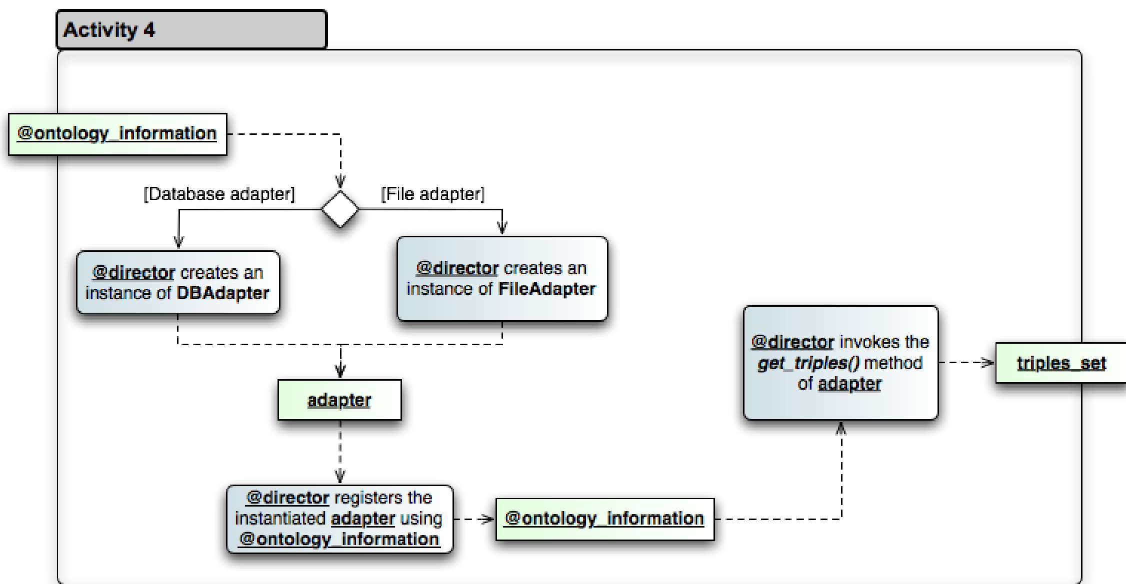


**Figure 3.15: Activity diagram 4: *Adapter*.** Creation of an *Adapter* and import of the ontology triples (actions are indicated as rounded rectangles in blue; variables as not rounded ones in green).

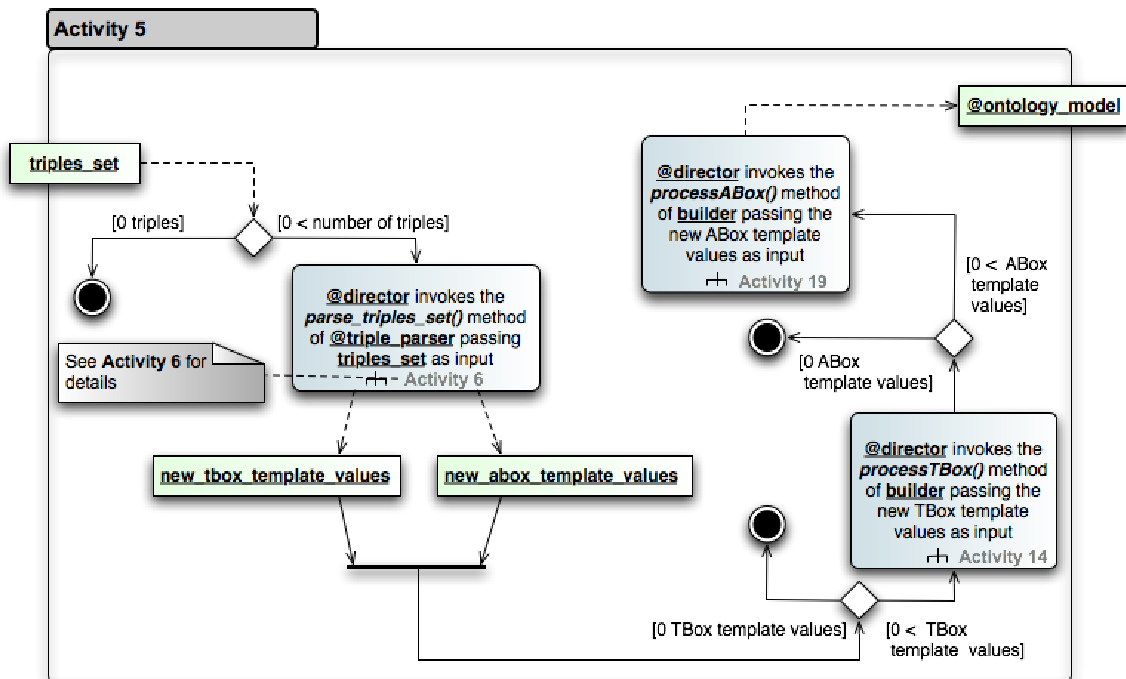


**Figure 3.16: Activity diagram 5: triple parsing and ontology conversion.** Parsing the ontology triples and deep integration (actions are indicated as rounded rectangles in blue; variables as not rounded ones in green).

---

put – the corresponding activity diagram describing the parsing process is further described in Section 3.5 starting with activity diagram 6. Method *parse_triples_set()* returns two arrays *new_abox_template_values* and *new_tbox_template_values*.

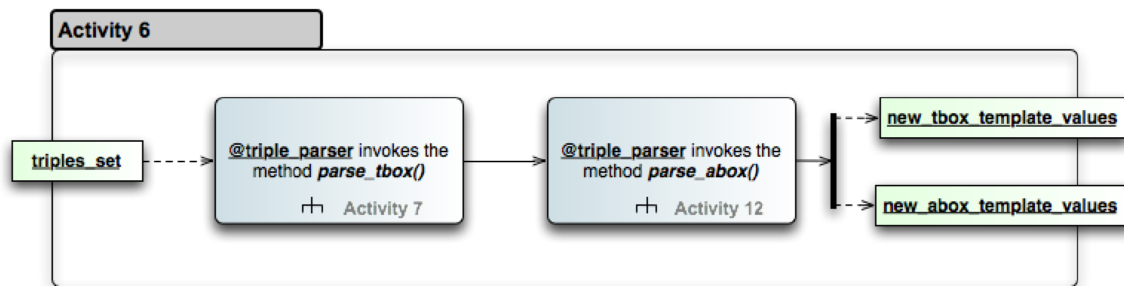


**Figure 3.17: Activity diagram 6: parsing details.** Details of the ontology triples parsing process (actions are indicated as rounded rectangles in blue; variables as not rounded ones in green).

---

After the ontology triples are parsed into pre-processed template values first the TBox is deep-integrated. If at least one TBox template value is existing *@director* invokes the *processTBox()* method of *builder* passing the new TBox template values as input (see activity diagram 14 in Section 3.6 starting page 55). With the TBox being converted and if at least one ABox template value exists *@director* invokes the *processABox()* method of *builder* passing the new ABox template values as input (see activity diagram 19 in Section 3.6). These deep integration actions are described in detail in Section 3.6.

## 3.5 Triple Parser: Mapping OWL Triples to an Abstract Syntax Based Representation in RUBY

DEEP SEMANTICS' class *TripleParser*, respectively a RUBY instance of it, is responsible for mapping OWL triples to an abstract syntax based representation in RUBY. Figure 3.17 shows the two top-level steps of the ontology triples parsing process: a) *@triple_parser* invokes the method *parse_tbox()* to pre-process the TBox triples (activity diagram 7) and b) *@triple_parser* invokes then the method *parse_abox()* to pre-process the ABox triples (activity diagram 12). To sum up *TripleParser* reads the ontology triples in *triples_set*, parses them and returns the two variables *new_tbox_template_values* and *new_abox_template_values* storing the pre-processed values for the later deep integration. The details of the triple parsing process are described in the next subsection.

### 3.5.1 The Triple Parsing Process

The triple parsing process comprises all processing steps that are needed to convert a set of input ontology triples into a set of pre-processed template values. These template values can then be used to deep integrate the functional ontology model. The parsing process itself is complex. Activity diagram 7 (as shown in Figure 3.18) which constitutes the pipeline of the primary parsing step comprises eleven actions alone – with two of these actions (action 4 and 8) being further described in activity diagram 8 (Figure 3.19) and 11 (Figure 3.22). In the following list all eleven actions are described in order of their invocation:

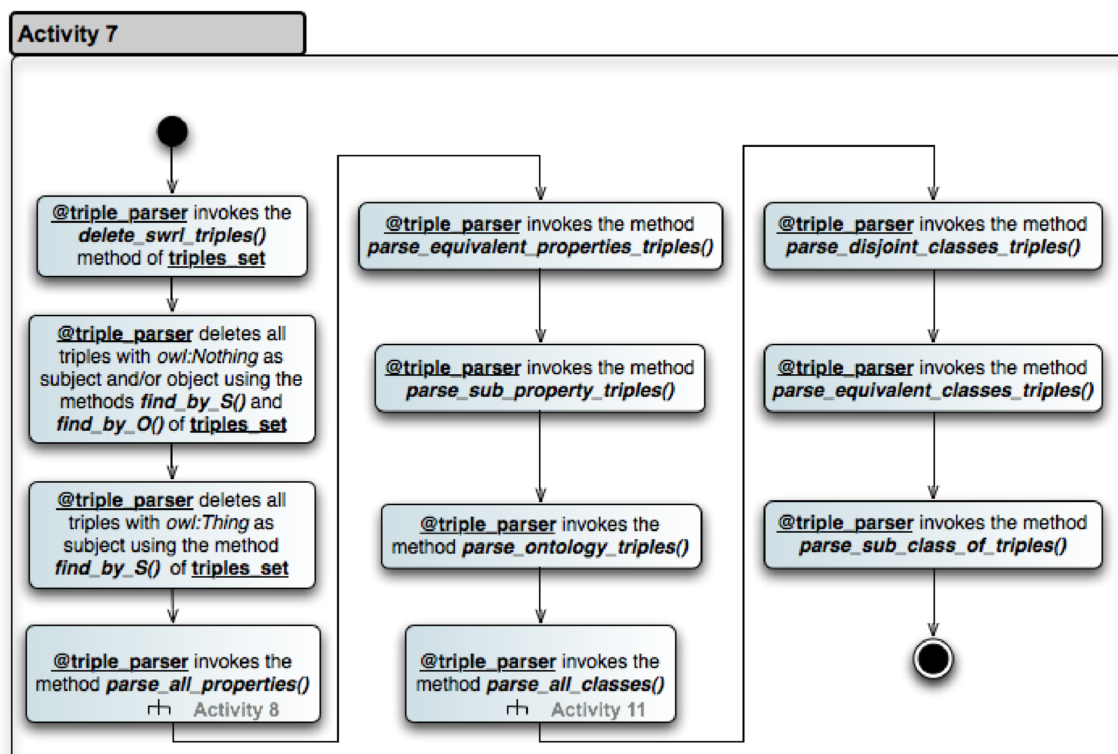


**Figure 3.18: Activity diagram 7: TBox parsing.** Details for the TBox parsing process (actions are indicated as rounded rectangles in blue; variables as not rounded ones in green).

---

1. *@triple_parser* invokes the *delete_swrl_triples()* method of *triples_set*. This action is required to delete possible SWRL related triples in the ontology as they are not needed for DEEP SEMANTICS but would extend the computing time of triple parsing.

2. *@triple_parser* deletes all triples with *owl:Nothing* as subject and/or object using the methods *find_by_S()* and *find_by_O()* of *triples_set*. Comment: triples including *owl:Nothing* do not encode any utilizable information with regard to the functional RUBY ontology model. As the case for SWRL triples in the previous action do these triples only extend the required computing time without having any functional relevance for DEEP SEMANTICS. The methods *find_by_S()* and *find_by_O()* are convenient methods that all exploit the fundamental method *findTriple()*. *findTriple()* implements the functionality to fetch all triples from *triples_set* that match to an arbitrary combination of subject, predicate and object patterns. For example *find_by_S()* in turn can be used to pass a subject pattern with predicate and object values being irrelevant.

3. *@triple_parser* deletes all triples with owl:Thing as subject using the method *find_by_S()* of *triples_set*. Triples with owl:Thing in the subject are not utilizable for DEEP SEMANTICS and therefore omitted.

4. *@triple_parser* invokes the method *parse_all_properties()*. This method is used to parse all datatype property and object property triples. Details are shown in activity diagram 8 in Figure 3.19

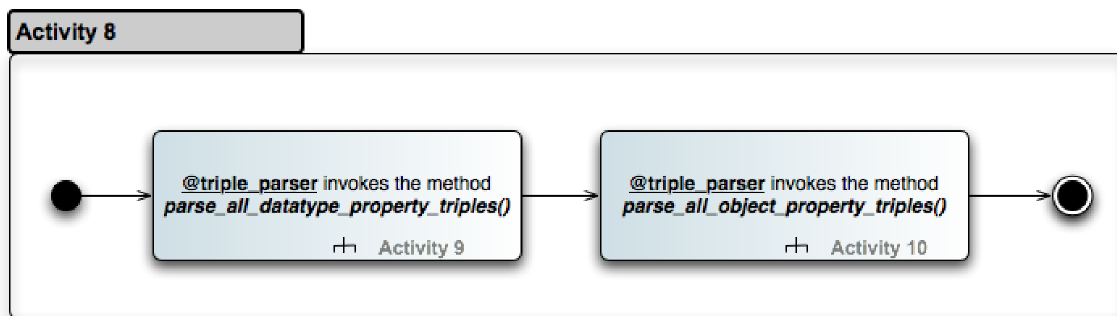


**Figure 3.19: Activity diagram 8: property parsing.** The property parsing process (actions are indicated as rounded rectangles in blue; variables as not rounded ones in green).

5. *@triple_parser* invokes the method *parse_equivalent_properties_triples()*. Triples defining equivalence between datatype properties and triples defining equivalence between object properties are parsed and integrated into the corresponding template value objects.

6. *@triple_parser* invokes the method *parse_sub_property_triples()*. Sub-property relations defining triples are parsed and integrated into the corresponding template value objects.

7. *@triple_parser* invokes the method *parse_ontology_triples()*. This method fetches and parses all ontology construct related triples using *find_by_PO('rdf:type', 'owl:Ontology')* – that means all triples with predicate rdf:type and object owl:Ontology are considered for this step.

8. *@triple_parser* invokes the method *parse_all_classes()*. This method parses all OWL URI classes, all OWL classes with blank nodes as subject as well as all restrictions (which are also ontology classes). Details are shown in activity diagram 11 as illustrated in Figure 3.22.

9. *@triple_parser* invokes the method *parse_disjoint_classes_triples()*. Triples defining disjointness between ontology classes are parsed and integrated into the corresponding class template value objects.

10. *@triple_parser* invokes the method *parse_equivalent_classes_triples()*. Analogous to the previous action triples defining equivalence between ontology classes are parsed and integrated into the corresponding class template value objects.

11. *@triple_parser* invokes the method *parse_sub_class_of_triples()*. Subclass relations defining triples are parsed and integrated into the corresponding template value objects.

Property parsing as shown in Figure 3.19 includes two distinct actions. Firstly, *@triple_parser* invokes its method *parse_all_datatype_property_triples()* to parse all datatype property related triples. Secondly, does *@triple_parser* invoke the method *parse_all_object_property_triples()* to parse analogously all object property related triples.

Activity diagram 9 in Figure 3.20 illustrates the detailed datatype property parsing process. The first action in this activity diagram states the following: *@triple_parser* fetches all triples that

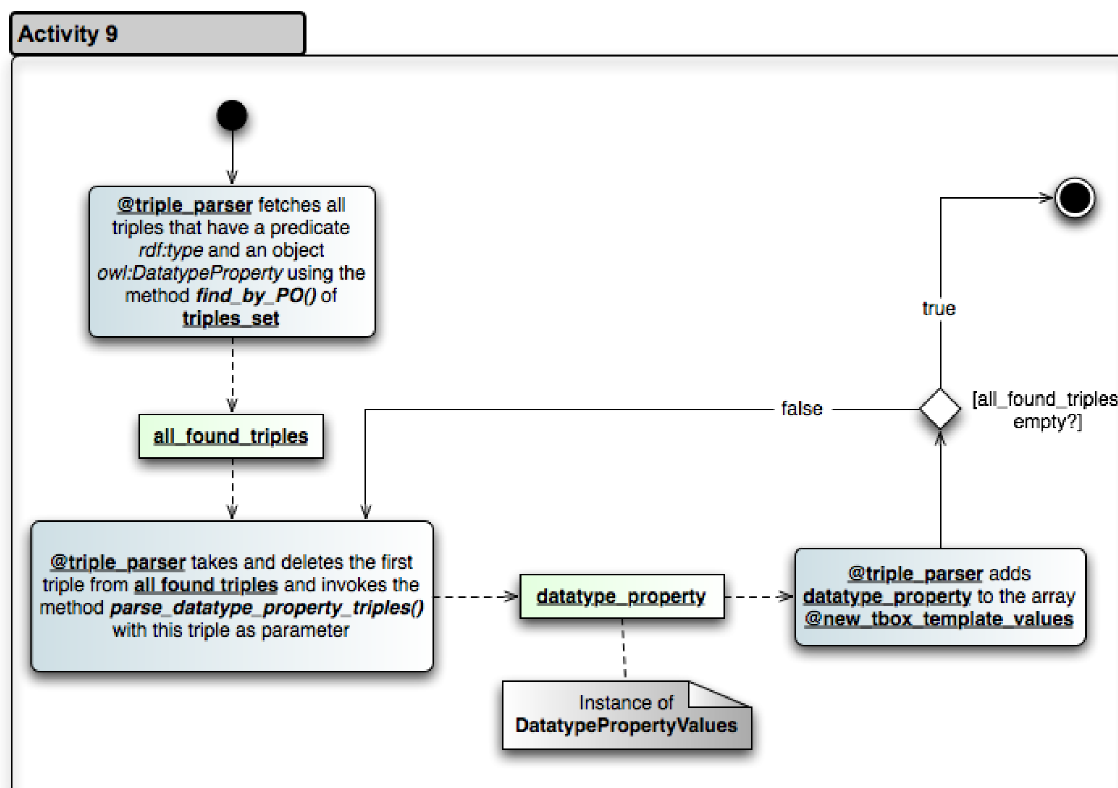


**Figure 3.20: Activity diagram 9: datatype property parsing.** The datatype property parsing process (actions are indicated as rounded rectangles in blue; variables as not rounded ones in green).

---

have a predicate *rdf:type* and an object *owl:DatatypeProperty* using the method *find_by_PO()* of *triples_set*.

The found triples are temporary stored in the variable *all_found_triples* and further processed as described in the next action: *@triple_parser* takes and deletes the first triple from *all_found_triples* and invokes the method *parse_datatype_property_triples()* with this triple as parameter – *parse_datatype_property_triples()* for example identifies triples defining whether the particular datatype property is a functional property calling *find_by_SPO(datatype_property.block_id, 'rdf:type', 'owl:FunctionalProperty')*. The returned *datatype_property* variable contains the parsed datatype property template value object that is then added to array *@new_tbox_template_values* by *@triple_parser*. If no more triples are left in *all_found_triples* datatype property parsing is completed. Otherwise *@triple_parser* takes and deletes the next triple from *all_found_triples*.

Figure 3.21 shows the details of the object property parsing process which resembles datatype property ones:

- *@triple_parser* fetches all triples that have a predicate rdf:type and an object owl:ObjectProperty using the method *find_by_PO()* of *triples_set* and returns the found triples recorded in *all_found_triples*.

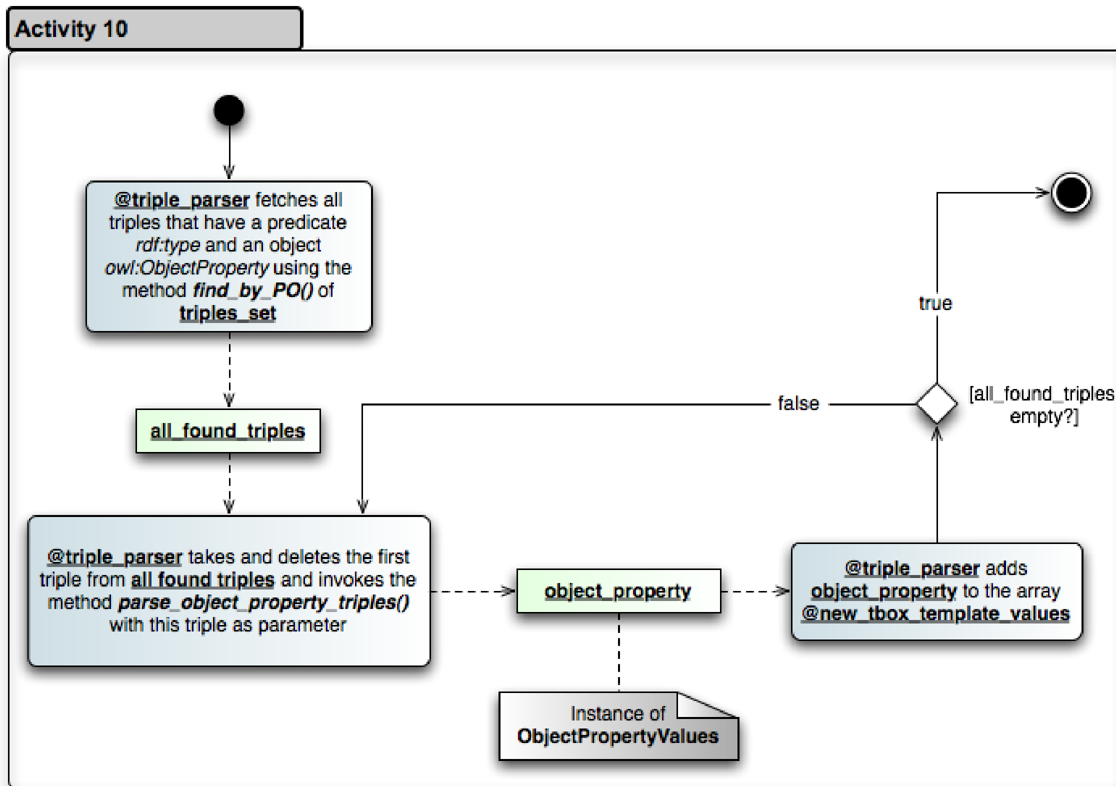


**Figure 3.21: Activity diagram 10: object property parsing.** The object property parsing process (actions are indicated as rounded rectangles in blue; variables as not rounded ones in green).

---

- *@triple_parser* takes and deletes the first triple from *all_found_triples* and invokes the method *parse_object_property_triples()* with this triple as parameter. The returned *object_property* variable contains the parsed object property template value object.
- *@triple_parser* adds *object_property* to the array *@new_tbox_template_values*.
- If no more triples are left in *all_found_triples* object property parsing is completed. Otherwise *@triple_parser* takes and deletes the next triple from *all_found_triples* as illustrated in Figure 3.21.

The details of ontology class parsing are illustrated in Figure 3.22. The shown activity diagram starts with *@triple_parser* fetching all triples that have a predicate rdf:type and an object owl:Class using the method *find_by_PO()* of *triples_set* returning *all_found_triples*. The next action displays *@triple_parser* taking and deleting the first triple from *all_found_triples*. If the triple has a B-Node as subject *@triple_parser* invokes the method *parse_restriction_triples()* with this triple as parameter. The parsing output value *restriction* – an instance of DEEP SEMANTICS class *RestrictionValues* – is then added by *@triple_parser* to the *@new_tbox_template_values* array.

If the triple on the other hand has not a B-Node as subject *@triple_parser* invokes the method *parse_class_triples()* with this triple as parameter. In this case the output is stored in *klass*. Like

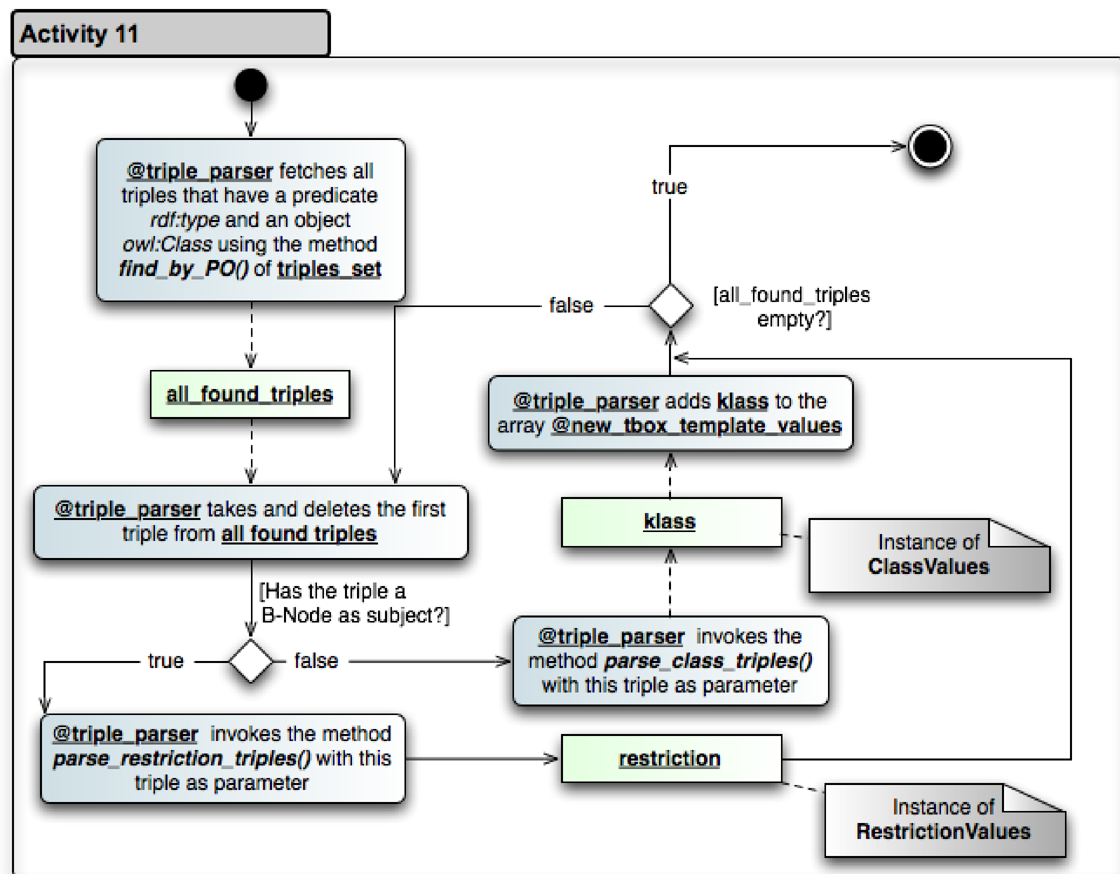


**Figure 3.22: Activity diagram 11: class parsing.** The class parsing process (actions are indicated as rounded rectangles in blue; variables as not rounded ones in green).

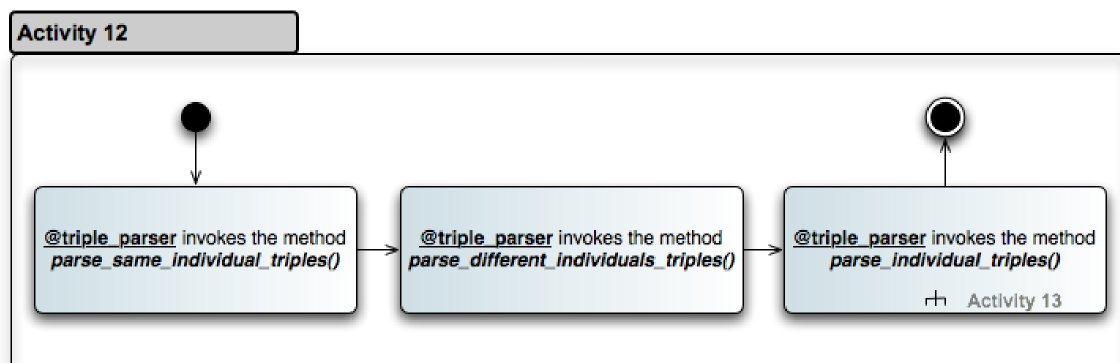


**Figure 3.23: Activity diagram 12: ABox parsing.** The ABox parsing process (actions are indicated as rounded rectangles in blue; variables as not rounded ones in green).

---

*restriction* above variable *klass* is added to *@new_tbox_template_values*, also. In either case – whether a URI class or a restriction has just been parsed – the process is re-iterated until *all_found_triples* is empty.

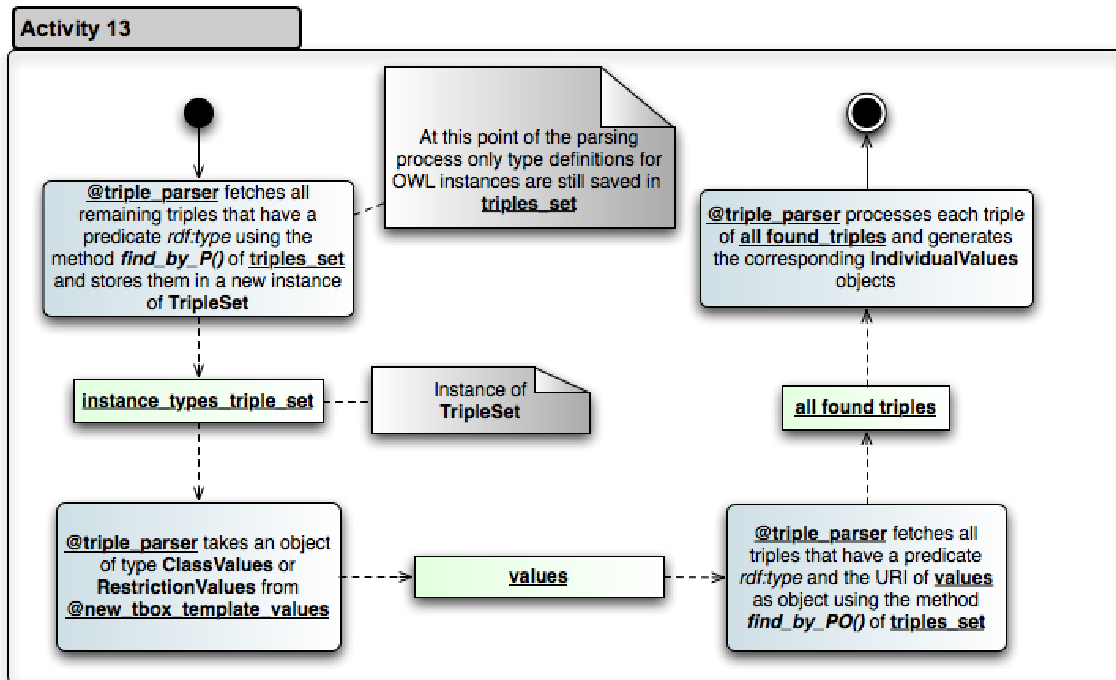


**Figure 3.24: Activity diagram 13: instance parsing.** The OWL individuals parsing process (actions are indicated as rounded rectangles in blue; variables as not rounded ones in green).

---

After considering class triples TBox parsing is complete. Taking a look back on Figure 3.17 we see that the next process action states that *@triple_parser* now invokes the method *parse_abox()*. The ABox parsing is covered by activity diagram 12. This activity diagram is illustrated in Figure 3.23. It contains three primary actions:

1. *@triple_parser* invokes the method *parse_same_individual_triples()*. The built-in OWL property owl:sameAs links an individual to an individual indicating that both individuals are identical. This method parses the corresponding triples and generates RUBY instances of *SameIndividualValues* with each of these instances representing a set of mutual identical instances.

2. *@triple_parser* invokes the method *parse_different_individuals_triples()*. An owl:differentFrom statement indicates that two URI references refer to different individuals. Analogous to the previous processing of identical individuals this method parses triples defining sets of mutual exclusive individuals. The method generates RUBY instances of *DifferentIndividualsValues* with each of these instances representing a set of mutual exclusive instances.

3. *@triple_parser* invokes the method *parse_individual_triples()*. An individual has to be an rdf:type of either OWL class or OWL restriction. Method *parse_individual_triples()* parses all individual related triples in *triples_set* – except the ones already considered in the two actions before. Because this parsing step is quite complex it is illustrated in its own activity diagram in Figure 3.24 which is described in the following:

(a) At this point of the parsing process only type definitions for OWL instances are still left in *triples_set*. @*triple_parser* fetches all remaining triples that have a predicate rdf:type using the method *find_by_P()* of *triples_set* and stores them in a new instance of *TripleSet* named *instance_types_triple_set*.

(b) @*triple_parser* takes an object *values* of type *ClassValues* or *RestrictionValues* from @*new_tbox_template_values*.

(c) @*triple_parser* fetches all triples that have rdf:type as predicate and the URI of *values* as object using the method *find_by_PO()* of *triples_set*. The fetched triples are stored in *all_found_triples*.

(d) @*triple_parser* processes each triple of *all_found_triples* and generates the corresponding *IndividualValues* objects including annotations of these individuals as well as property and value pairs (facts about the individual and instance, respectively).

At this point of the DEEP SEMANTICS processing pipeline the ontology triples have been read in using an instance of *SourceAdapter* and parsed into ABox and TBox related instances of *TemplateValues* using an instance of *TripleParser*. The next and last major conversion step is the deep integration of the OWL ontology into the RUBY functional ontology model described in detail in the following Section 3.6.

## 3.6 Deep Integration Builder

*Deep Integration Builder* takes the parsed template values of the corresponding ontology definition, ontology classes, properties and instances, and converts these into a consistent functional ontology model in RUBY. The currently available implementation of *Deep Integration Builder* is the RUBY class *DeepIntegrationBuilderOWLLite*. *DeepIntegrationBuilderOWLLite* first converts the TBox ontology classes and property axioms, deep integrates these into a functional model and then generates the ABox instances.

Figure 3.25 illustrates the activity diagram of the ontology TBox deep integration. This activity diagram reads in variable *template_values*, comprises thirteen sophisticated actions and after the execution of these actions returns the functional TBox @*ontology_model*. The following list details the thirteen actions:

1. The *builder* takes the *ontology_values* object being part of *template_values* and invokes the method *newOntology()* with this object as parameter. The details for this action are described in Subsection 3.6.1.

2. The *builder* takes all *DatatypePropertyValues* and *ObjectPropertyValues* objects from *template_values* and for every object iteratively invokes either the method *newDatatypeProperty()* or *newObjectProperty()* with the respective object as parameter. The details for this action are described in Subsection 3.6.2.

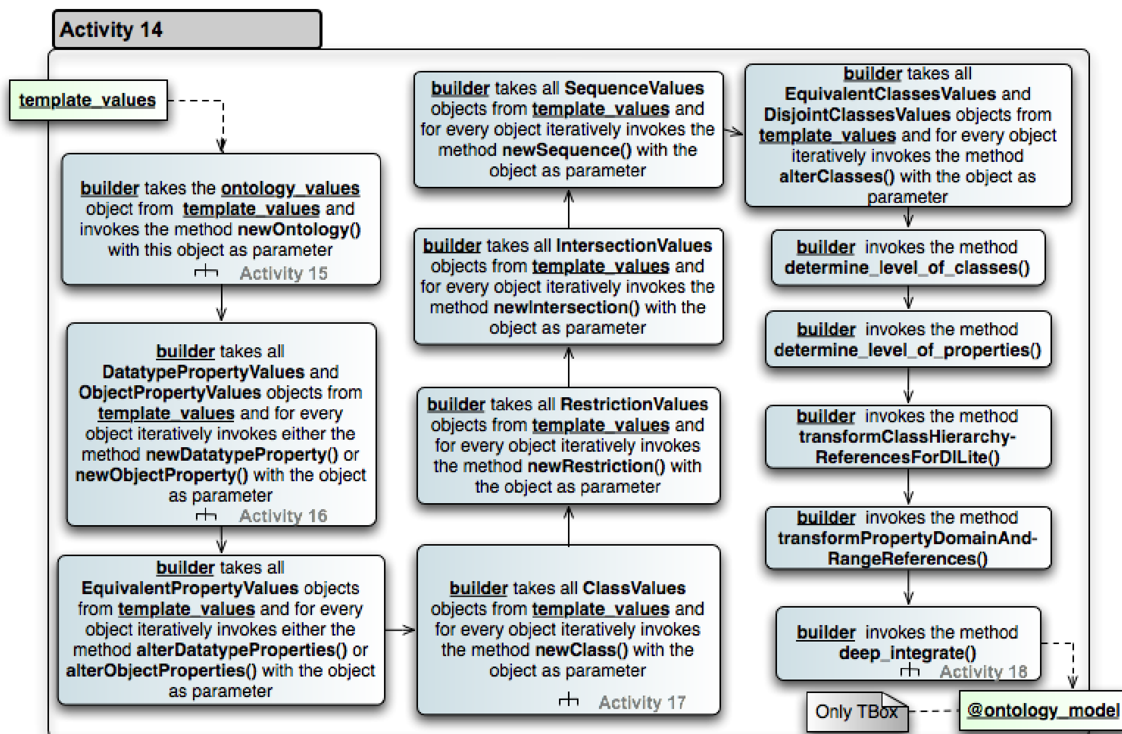


**Figure 3.25: Activity diagram 14: TBox deep integration.** The deep integration of the ontology TBox activity (actions are indicated as rounded rectangles in blue; variables as not rounded ones in green).

3. Next *builder* takes all *EquivalentPropertyValues* objects from *template_values* and for every object it iteratively invokes either the method *alterDatatypeProperties()* or *alterObjectProperties()* with the respective object as parameter. As the ontology properties have been already converted into RUBY objects in the previous action this action extends the *@equivalent_properties* (see Figure 3.10 for RUBY class *DatatypeProperty* and Figure 3.11 for RUBY class *ObjectProperty*) variable in case the corresponding property is stored in an *EquivalentPropertyValues* set.

4. In the fourth action *builder* takes all *ClassValues* objects from *template_values* and for every object iteratively invokes the method *newClass()* with the object as parameter. The details for this action are described in Subsection 3.6.3 (including the consideration of *Restriction* objects).

5. Next *builder* takes all *RestrictionValues* objects from *template_values* and iteratively invokes the method *newRestriction()* with the object as parameter for every object. The generated *Restriction* objects (see also Figure 3.8) are important for the meta-programming of the RUBY ontology classes described in subsequent action.

6. Here *builder* takes all *IntersectionValues* objects from *template_values* and iteratively invokes the method *newIntersection()* with the object as parameter for every object. In OWL LITE multiple domain and range assertions for a property constitute intersections of the included ontology classes.

7. In the seventh action *builder* takes all *SequenceValues* objects from *template_values* and for every object iteratively invokes the method *newSequence()* with the object as parameter. In OWL LITE sequences can be part of *DifferentIndividuals* constructs.

8. Now *builder* takes all *EquivalentClassesValues* and *DisjointClassesValues* objects from *template_values* and for every object iteratively invokes the method *alterClasses()* with object as parameter. *alterClasses()* extends as the case may be the @@*equivalent_classes*, @@*equivalent_bnode_classes*, @@*disjoint_classes* and/or @@*bnode_disjoint_classes* variables (see also Figure 3.8 for details of dynamically generated *?ClassLocalName?* RUBY classes).

9. Next *builder* invokes the method *determine_level_of_classes()*. This method computes for each ontology class its level in the class hierarchy – starting at zero for root classes. The level value of a class can be of use for example for printing out all ontology classes by level.

10. *builder* invokes the method *determine_level_of_properties()*. Analogous to *determine_level_of_classes()* does this method compute for each property the level in its hierarchy – starting at zero for root properties.

11. Next *builder* invokes the method *transformClassHierarchyReferencesForDILite()*. Transform sub-class-of and equivalent-class reference strings into their corresponding RUBY constructs.

12. In the twelfth action *builder* invokes the method *transformPropertyDomainAndRangeReferences()*. Like the previous method this one transforms the representations of references to domain and range values from strings to corresponding RUBY constructs.

13. At last *builder* invokes the method *deep_integrate()*. This method assembles the previously converted RUBY ontology classes and properties, and produces an integrated functional ontology model. The details for this action are described in Subsection 3.6.4.

### 3.6.1 Conversion of the OWL Ontology Definition into a RUBY Module

Figure 3.26 shows the activity diagram that illustrates the conversion of the ontology definition into a RUBY module. This RUBY module is thereby dynamically generated and has the structure shown in Figure 3.7. The first action of Figure 3.26 comprises *builder* invoking the method *buildOntology()* of the module *OntologyTemplate* (see also Figure 3.3) with the object *ontology_values* as parameter. *OntologyTemplate* then takes the ontology construct parameters saved in *ontology_values* and generates a string named *ontology_model_string* representing the code for the ontology model RUBY module. *builder* invokes the RUBY method *module_eval()* with the string *ontology_model_string* as parameter to convert the string into a RUBY module having the structure shown as *?OntologyLocalName?* in Figure 3.7. The return value of *module_eval()* is @*ontology_model* saving the not yet completely integrated ontology model.

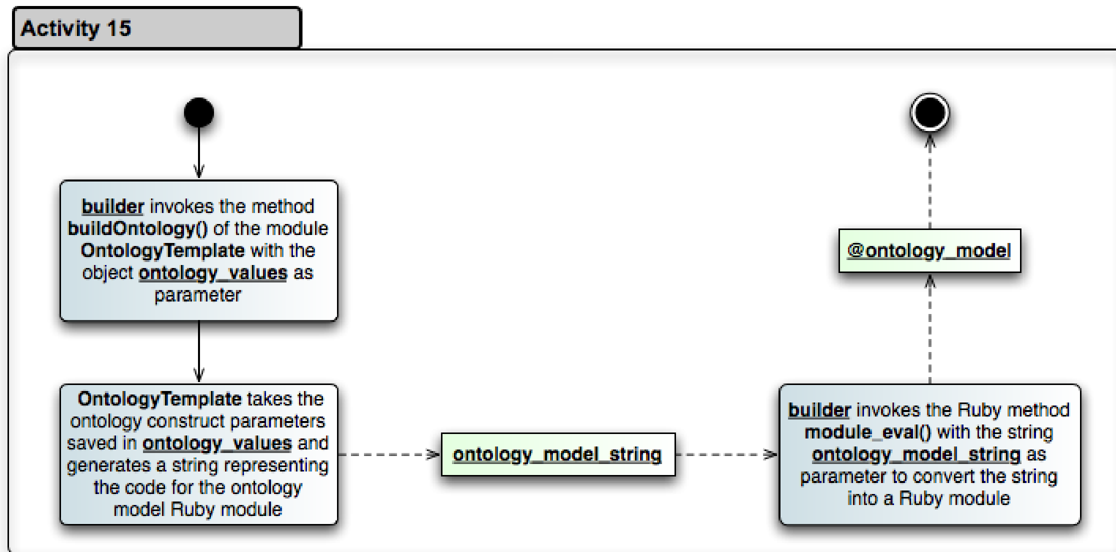


**Figure 3.26: Activity diagram 15: ontology module creation.** Conversion of the OWL ontology definitions into a RUBY module (actions are indicated as rounded rectangles in blue; variables as not rounded ones in green).

---

### 3.6.2 Conversion of OWL Properties into RUBY Objects

Figure 3.27 shows the conversion of ontology properties – both datatype properties and object properties – into RUBY objects. Illustrated is activity diagram 16 starting with a condition branching the process flow dependent on the type of property.

#### Conversion of Datatype Properties

For datatype properties, *builder* invokes the method *newDatatypeProperty()* of *@ontology_model* with the object *datatype_property_values* as parameter. Inside of *newDatatypeProperty()* the method *PropertyTemplate.buildDatatypeProperty()* is called which produces a RUBY instance of *DatatypeProperty* temporally stored in variable *new_datatype_property* (see also Figure 3.10). Next *@ontology_model* adds *new_datatype_property* to its array *@@datatype_properties*.

At the end of datatype property conversion, the corresponding *DatatypeProperty* instance includes variables and methods handling the local name of the property, the level of the property in the hierarchy, functionality to check if the property is deprecated or functional. Additional variables save super-properties, sub-properties and equivalent-properties of the described property. The domains and ranges can be retrieved as well by using the provided methods *domain* and *range*.

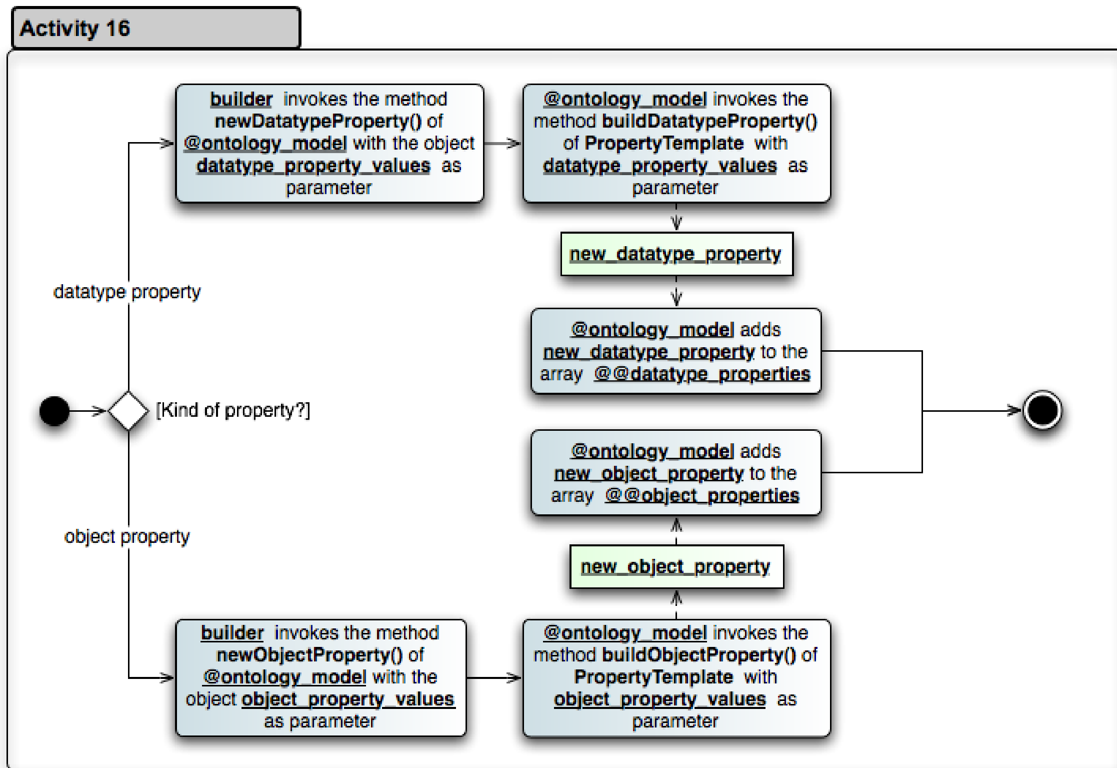


**Figure 3.27: Activity diagram 16: property object creation.** Conversion of the OWL properties into RUBY objects (actions are indicated as rounded rectangles in blue; variables as not rounded ones in green).

---

### Conversion of Object Properties

If the processed template values correspond to an object property *builder* invokes the method *newObjectProperty()* of *@ontology_model* with the object *object_property_values* as parameter. Next *@ontology_model* invokes the method *buildObjectProperty()* of *PropertyTemplate* with *object_property_values* as parameter. The generated *new_object_property* object is added by *@ontology_model* to its array *@@object_properties*.

Similar to the result of datatype property conversion does the corresponding *ObjectProperty* instance include the same variables and methods. However, *ObjectProperty* instances additionally include the variables *@inverse_functional*, *@transitive*, *@symmetric* and *@inverse_of*. Values of these variables can be retrieved with related *setter* methods (see also Figure 3.11).

## 3.6.3 Conversion of OWL Classes into RUBY Classes

Activity diagram 17 illustrated in Figure 3.28 describes the conversion of named OWL classes into RUBY classes. Firstly, *builder* invokes the method *newClass()* of *@ontology_model* with the object *class_values* as parameter. Secondly, *@ontology_model* invokes the method *buildClass()* of *ClassTemplate* with *class_values* as parameter. Thirdly, the newly created class

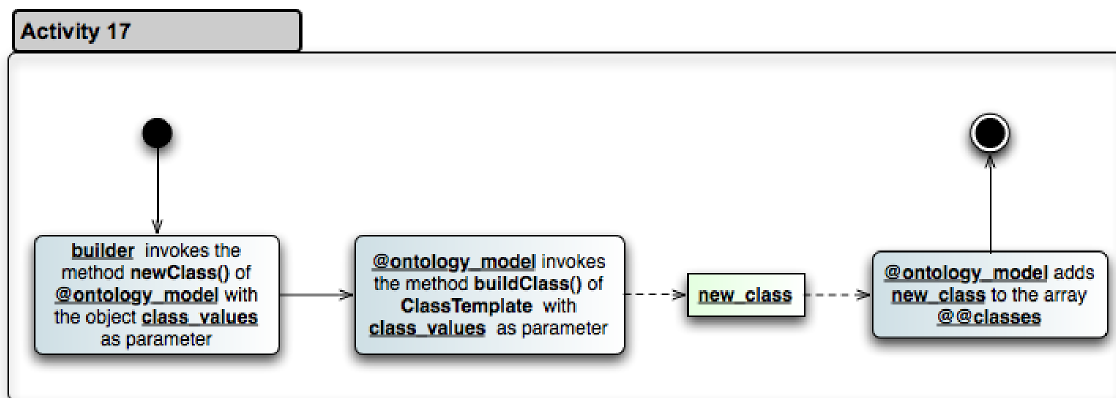


**Figure 3.28: Activity diagram 17: RUBY ontology class creation.** Conversion of the OWL classes into RUBY classes (actions are indicated as rounded rectangles in blue; variables as not rounded ones in green).

---

is stored in *new_class* and added by *@ontology_model* to the array *@@classes* of *@ontology_model*.

This conversion produces a meta-programmed RUBY class representing an ontology class. In Figure 3.8 such a class is modeled with the extended UML class diagram *?ClassLocalName?*. The RUBY ontology class comprises at this point of processing for example class and instance variables for local names as well as class variable *@@instances* which – in combination with *getter* and *setter* methods – allows the access of all instances of the ontology class.

### 3.6.4 TBox Deep Integration – Assembling of the Deep Integrated RUBY Properties and Classes into a consistent RUBY Representation of the Ontology

Figure 3.29 illustrates the processing steps for the assembling and deep integration of ontology classes and properties into functional ontology model in RUBY. This last step of the TBox deep integration is relatively complex including twelve actions as well as four conditions and one fork and join pair. The following list describes the sequence of actions in activity diagram 18:

1. *builder* invokes the method *process_datatype_properties_inheritance()*. *@ontology_model* uses its method *listDatatypePropertiesByLevel()* to iterate over datatype properties by hierarchy level. For each datatype property the following three methods are called:

   (a) *process_domain_inheritance(datatype_property)*: The sub-property inherits all domains of the parent property. If a domain of a parent property is a superclass of an existing domain of sub-property this class has not to be added to the sub-property's domain. Domain inheritance works identical for datatype and object properties.

   (b) *process_range_inheritance(datatype_property)*: The sub-property inherits all ranges of the parent property. As ranges for datatype properties are datatypes

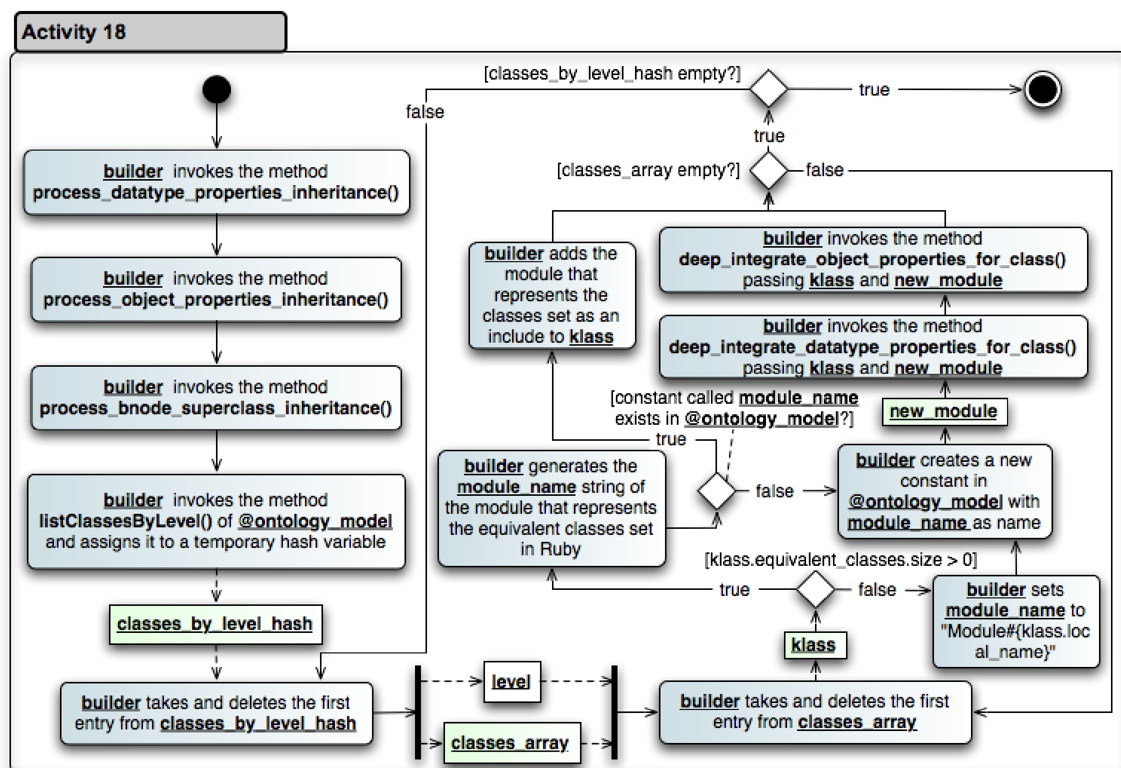


**Figure 3.29: Activity diagram 18: deep integration details 1.** Assembling and deep integration of the classes and properties into an ontology model in RUBY (actions are indicated as rounded rectangles in blue; variables as not rounded ones in green).

and the definition of multiple ranges for datatype properties can very easily lead to problems (e. g. *"What is the intersection of datatypes integer and boolean?"*), ontology designers should be very careful about what ranges they define.

(c) *process_global_restriction_inheritance(datatype_property)*: The sub-property inherits functional property membership of its super-property. Note that if a super-property is not defined to be functional but the sub-property is, than there is no inheritance – i. e. the child does not inherit the missing of a feature from its parent.

2. *builder* invokes the method *process_object_properties_inheritance()*. This method does basically the same on object properties as *process_datatype_properties_inheritance()* on datatype properties. However, there are two important differences:

   (a) *process_range_inheritance(object_property)*: For object properties range inheritance is important. The sub-property inherits all ranges of the parent property. If a range of a parent property is a superclass of an existing range of the sub-property this class has not to be added to the range of the sub-property.

   (b) *process_global_restriction_inheritance(object_property)*: For object properties global restriction inheritance additionally comprises consideration of inverse-functional statements.

3. *builder* invokes the method *process_bnode_superclass_inheritance()*. While inherited URI superclasses are made explicit by the PELLET reasoner the same is not the case for B-Node superclass inheritance. This method therefore determines all B-Node superclasses – that is restrictions – of every URI superclass of a class and adds them to *@@bnode_super_classes_values* of this particular class.

4. In the fourth action *builder* invokes the method *listClassesByLevel()* of *@ontology_model* and assigns it to a temporary hash variable *classes_by_level_hash*.

5. Next *builder* takes and deletes the first entry from *classes_by_level_hash*. The return values are *level* (the key of the hash) indicated the level of the classes stored in array *classes_array* (the value of the hash).

6. In the sixth action *builder* takes and deletes the first entry *klass* from *classes_array*. Which action is carried out next dependents on whether modeled ontology class in *klass* has recorded equivalent classes or not.

7. If *klass* does not contain equivalent classes statements then *builder* sets *module_name* to *"Module#klass.local_name"*. The successive processing step is action 9.

8. Otherwise *builder* generates the *module_name* string of the module that represents the equivalent classes set in RUBY. If a constant named like the string saved in *module_name* is not already existing in module *@ontology_model* then the successive processing step is action 9 otherwise 12.

9. *builder* creates a new constant pointing to the newly generated model *new_module* in *@ontology_model* named *module_name*.

10. Further progressing *builder* invokes the method *deep_integrate_datatype_properties_for_class()* passing parameters *klass* and *new_module*. This processing step is described in detail in *"Deep integration of properties into related ontology classes in* RUBY *"* below.

11. Progressing from previous action *builder* now invokes its method *deep_integrate_object_properties_for_c* passing *klass* and *new_module* as parameters. This processing step is also described in detail in *"Deep integration of properties into related ontology classes in* RUBY *"* below.

12. If a constant named like the string saved in *module_name* exists after action 8 then *builder* adds the module that represents the classes set as an RUBY include to *klass*.

In summary describing the actions in Figure 3.29 one can put on record how assembling and deep integration of converted classes and properties into a consistent and functional ontology model in RUBY functions. However, as consideration of property domain statements – regarding source code implementation – requires meta-programming of the corresponding ontology class implementations, the details of this deep integration of properties is described in detail next.

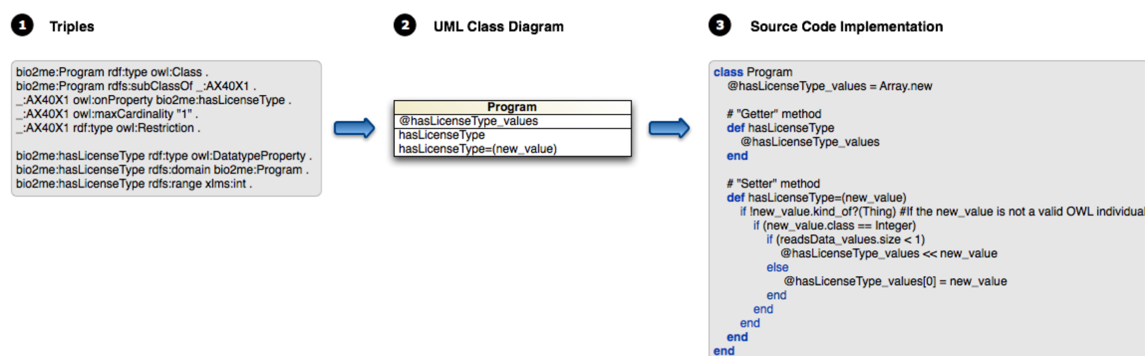


**Figure 3.30: Schematic illustration of the deep integration of an example property.** The namespace abbreviations are: xmls := http://www.w3.org/2001/XMLSchema#; rdf := http://www.w3.org/1999/02/22-rdf-syntax-ns#; rdfs := http://www.w3.org/2000/01/rdf-schema#; owl := http://www.w3.org/2002/07/owl#

---

### Deep integration of properties into related ontology classes in RUBY

Property deep integration into relevant ontology classes involves the generation of property related instance variables, *setter* and *getter* methods. Figure 3.30 shows an example from **BIO2ME** of such a deep integration process. On the left side **(1)** the class (*Program*) and property (*hasLicenseType*) defining triples are shown – the relevant namespaces are described in the caption. As displayed by a blue arrow these triples are mapped to RUBY implementation of ontology class *Program* – here shown as its UML class diagram representation **(2)**. The next blue arrow indicates mapping of this UML class diagram on its complying RUBY source code, dynamically generated using meta-programming by DEEP SEMANTICS. This implementation comprises the related instance variable *@hasLicenseType_values*, *setter* method *hasLicenseType=(new_value)* and *getter* method *hasLicenseType()*.

Implementation of *getter* methods is simple. They only return the value of the corresponding attribute variable. For *isUsedInProgram* from **BIO2ME**– as an additional example to the one shown in Figure 3.30 – the generated code by DEEP SEMANTICS is the following:

```
def isUsedInProgram
  return @isUsedInProgram_values
end
```

In contrast *setter* methods can be considerably more complex as cardinality and type constraints can be involved (compare the example in Figure 3.30 with a maximum cardinality on property *hasLicenseType* contained for instance). However, this complexity is required to ensure that DEEP SEMANTICS ABox operations do not lead to logical inconsistencies (that means DEEP SEMANTICS is consistency safe).

It is important to mention that some additional implementation complexity is omitted because DEEP SEMANTICS does not support additional inference functionality. Instead it should be used in tandem with a specialized OWL reasoner like PELLET. Chapter 4 describes the **IKEN** semantic application, which is an example for cooperative application of DEEP SEMANTICS with PELLET.

DEEP SEMANTICS considers any maximum or absolute cardinality and *some-values-from* as well as *all-values-from* restrictions. Minimum cardinality restrictions lead to the fact that the corresponding property cannot be used with the restriction-related ontology class anymore. Additionally, it integrates the global cardinality induced by functional property declarations and inverse functional restrictions. The described adjustment to the practical needs of Semantic Web developers – no inference support and consistency safeness – lead to the inclusion of OWL logic constraints as specified in Table 3.1 and Table 3.2 on page 88 and 89, respectively.

Concluding this subsection the following listing shows an example meta-programmed source code of a *setter* method for a *functional* and *inverse functional* object property with an additional local *all-values-from* restriction:

**Listing 3.1:** An concluding *setter* method example

```
1  def isMarriedTo=(new_value)
2    if new_value.kind_of?(Thing)
3      if !new_value.used_already_as_value?("isMarriedTo")
4        if (new_value.types.include?(Person) || new_value.class.super_classes.include?(Person))
             && (new_value.types.include?(Adult) || new_value.class.super_classes.include?(Adult)
             )
5          if @isMarriedTo_values
6            if (@isMarriedTo_values.size < 1)
7              @isMarriedTo_values_values << new_value
8              new_value.used_already_as_value("isMarriedTo")
9            else
10             @isMarriedTo_values[0] = new_value
11             @isMarriedTo_values[0].used_not_already_as_value("isMarriedTo")
12             new_value.used_already_as_value("isMarriedTo")
13           end
14         else
15           @isMarriedTo_values_values = Array.new
16           @isMarriedTo_values_values << new_value
17           new_value.used_already_as_value("isMarriedTo")
18         end
19           else
20             puts "#{new_value.local_name} cannot be used as value of isMarriedTo!"
21       end
22     else
23       puts "#{new_value.local_name} has already been used and cannot be used again with an
             inverse functional property!"
24     end
25   end
26 end
```

### 3.6.5 ABox Deep Integration – Conversion of Instances into RUBY Objects

Figure 3.31 illustrates the conversion of all ontology instances into RUBY objects and instances of the RUBY classes, dynamically generated during TBox deep integration. Conversion of the ABox constitutes the last step of DEEP SEMANTICS' meta-programming-based deep integration pipeline. Activity diagram 19 in Figure 3.31 starts with *builder* invoking the method *remove_superclass_types_and_transform_references()* with *template_values* as parameter and overwriting *template_values* with the method's return value. This method call *remove_superclass_types_and_transform_references()* reduces the number of type definitions in *IndividualValues* objects to include only direct types – not inherited superclasses.

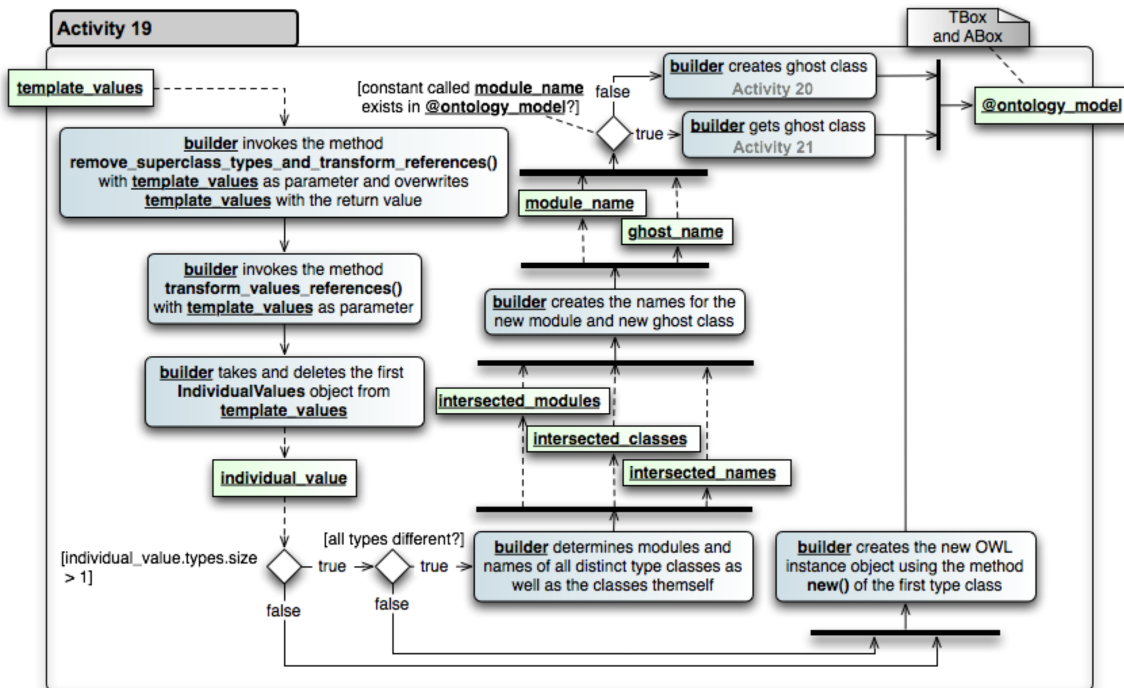


**Figure 3.31: Activity diagram 19: deep integration details 2.** Conversion of the OWL instances into RUBY objects (actions are indicated as rounded rectangles in blue; variables as not rounded ones in green).

---

The next action describes *builder* invoking method *transform_values_references()* with *template_values* as parameter. Next *builder* takes and deletes the first *IndividualValues* object from *template_values* and stores it in *individual_value*. If *IndividualValues* object stored in *individual_value* has more than one related ontology class in its *types* variable, and all of this classes are different, then the related instance belongs to a *Ghost* class. If *IndividualValues* object does only belong to one ontology class (or has multiple but mutually equal type-classes) then *builder* creates a new OWL instance object using the method *new()* of the first type-class – the corresponding ontology instance is therewith deep-integrated.

If an instance has multiple types *builder* determines RUBY modules and names of all distinct type-classes as well as the classes themselves. The return variables are *intersected_names*, *intersected_modules* and *intersected_classes*. Further progressing from the previous action *builder* creates names for the required module (*module_name*) and the corresponding required *Ghost* class (*ghost_name*). If a constant called *module_name* exists in *@ontology_model* then *builder* fetches the corresponding *Ghost* class (described in activity diagram 21 shown in Figure 3.33 on page 67), otherwise *builder* creates a new *Ghost* class (this is described in activity diagram 20 shown in Figure 3.32).

Figure 3.32 illustrates the creation of a new *Ghost* class and an instance of it. The six actions building up activity diagram 20 are:

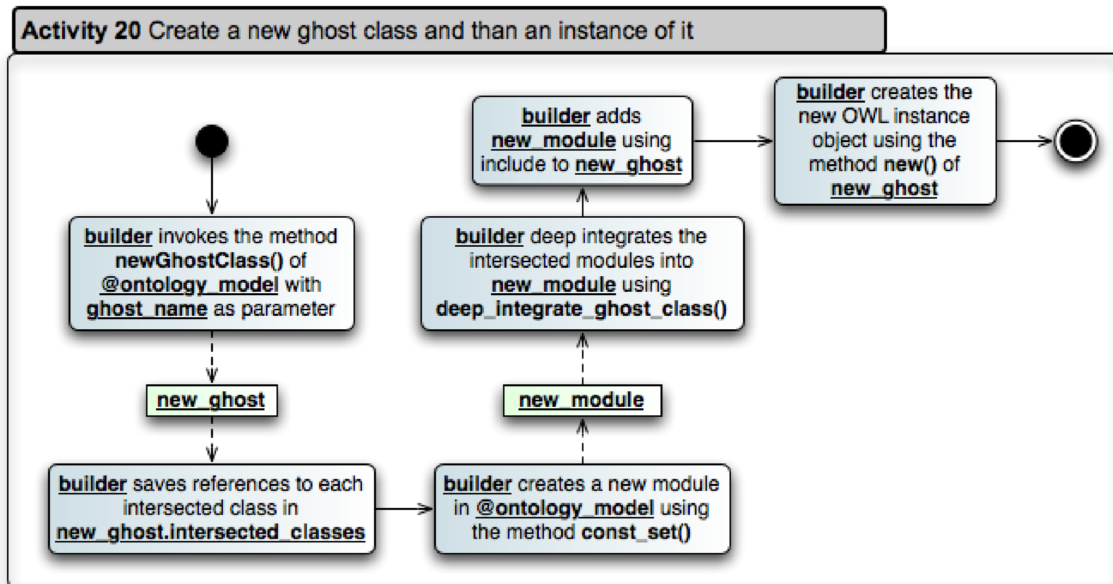


**Figure 3.32: Activity diagram 20: *Ghost* creation.** Creation of a new *Ghost* class followed by the initialization of an instance of it (actions are indicated as rounded rectangles in blue; variables as not rounded ones in green).

---

1. *builder* invokes method *newGhostClass()* of *@ontology_model* with *ghost_name* as parameter. The return value *new_ghost* references a meta-programmed RUBY class.

2. *builder* saves references to each intersected class in *new_ghost.intersected_classes*.

3. Next *builder* creates a new RUBY module in *@ontology_model* using the built-in method *const_set()* – the new module is temporarily stored in *new_module*.

4. Following up *builder* deep integrates the intersected modules into *new_module* using *deep_integrate_ghost_class()*.

5. Then *builder* adds *new_module* to *new_ghost* using RUBY's built-in *include*.

6. In a last step *builder* creates the new ontology instance RUBY object using method *new()* of *new_ghost*.

Fetching an existing *Ghost* class and then creating an instance of it is shown in Figure 3.33. The *builder* invokes method *getGhostClass()* of *@ontology_model* with *ghost_name* as parameter. The result is stored in *ghost* which is used next by *builder* to create the new OWL instance object using its *new* method.

In this chapter we have so far learned about DEEP SEMANTICS architecture, its conceptual relation to OWL LITE abstract syntax, and the complex details of how a set of RDF triples is read, parsed, converted and finally assembled into a functional ontology model. The remaining sections describe how a deep-integrated ontology can be utilized (Section 3.7) – including an example semantic application implemented from scratch – and how DEEP SEMANTICS compares to other Semantic Web frameworks.

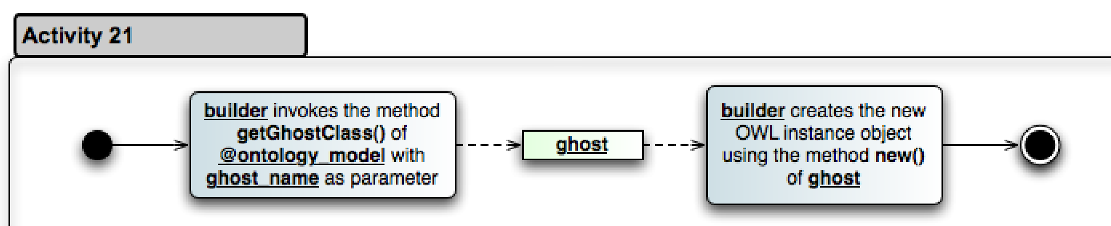


**Figure 3.33: Activity diagram 21: using an existing *Ghost* class.** Fetching of an existing *Ghost* class followed by the creation of an instance of this *Ghost* (actions are indicated as rounded rectangles in blue; variables as not rounded ones in green).

---

## 3.7 Utilization of the Deep Integrated Ontology

### 3.7.1 Using DEEP SEMANTICS to convert an Ontology into a RUBY Representation

Exploiting the benefits of DEEP SEMANTICS is straightforward. Assuming that an ontology is available the programmer in charge can incorporate a functional representation of this ontology using DEEP SEMANTICS with the following lines of RUBY code:

**Listing 3.2:** Using DEEP SEMANTICS to create a functional model of **IKEN**

```
1  require File.join(/DeepSemantics/active_semantics.rb')
2  require File.join(/DeepSemantics/Helper/namespaces.rb')
3
4  Namespaces.add_namespace('http://www.i-ken.de/iken3.owl#', 'iken:')
5
6  $deep_semantics = DeepSemantics.instance
7  $iken = $deep_semantics.set_director({'adapter' => 'FileAdapter', 'ontology_source' => '/
       iken_inferred3.nt', 'builder' => 'DeepIntegrationBuilderOWLLite'})
```

### 3.7.2 Working with OWL Classes, Properties and Instances in DEEP SEMANTICS

At first we have to include the required DEEP SEMANTICS files into our example script:

**Listing 3.3:** Including DEEP SEMANTICS into custom RUBY code

```
1  require File.join(/DeepSemantics/active_semantics.rb')
2  require File.join(/DeepSemantics/Helper/namespaces.rb')
```

Then we add the namespace of our example ontology in line 4 to the module *Namespaces* and create an instance of *DeepSemantics* that we pass to a variable *$deep_semantics*. Furthermore, we specify with *set_director()* the used director settings as follows (line 7): 1) use a file adapter to read the ontology triples from an N-Triple file that 2) can be accessed by the DEEP SEMANTICS director over the declared path and 3) choose the *DeepIntegrationBuilderOWLLite* builder to be used for the creation of the ontology. We get back the converted functional ontology representation in RUBY and save this model in variable *example*:

**Listing 3.4:** Adding the namespace of the ontology and creating an functional ontology model

```
1 Namespaces.add_namespace('http://www.ontoverse.org/example.owl#', 'example:')
2
3 deep_semantics = DeepSemantics.instance
4 example = $deep_semantics.set_director({'adapter' => 'FileAdapter', 'ontology_source' => '/
      example_inferred3.nt', 'builder' => 'DeepIntegrationBuilderOWLLite'})
```

Now that we can use the functional ontology model we access the ontology concept *Protein* and print out its rdfs:label assertions. What is of special interest here is that we can access the concept *Protein* just like any other RUBY class – ontology classes and RUBY classes are at this point functionally equal. First we print out the labels array and then using this array in a loop construct every single label on its own line:

**Listing 3.5:** Printing out the labels of class *Protein*

```
1 puts "1. Print out all rdfs:label values for Protein as array:"
2 puts example::Protein.rdfs_label
3 puts # This produces just an empty line in the output.
4 puts "2. Get the rdfs:label values array of Protein and print out each rdfs:label element
      separately:"
5 example::Protein.rdfs_label.each do |label|
6    puts label
7 end
```

We get the following output:

> *1. Print out all rdfs:label values for Protein as array:*
> *["protein", "Protein", "albumine"]*
>
> *2. Get the rdfs:label values array of Protein and print out each rdfs:label element separately:*
> *protein*
> *Protein*
> *albumine*

Next we want to see all labels of every instance of *Protein*. For this kind of tasks DEEP SEMANTICS offers the convenient method *instances()* for every ontology class. The corresponding listing is very straight forward:

**Listing 3.6:** Printing out the labels of every *Protein* instance

```
1 puts "Print for every protein instance all labels:"
2 example::Protein.instances.each do |protein_instance|
3    protein_instance.rdfs_label.each do |label|
4         puts label
5    end
6    puts
7    puts "--next protein--"
8    puts
9 end
```

The output we get looks like this:

> *Print for every protein instance all labels:*
> *hemoglobin*
> *haemoglobin*
>
> –next protein–
>
> *myoglobin*
>
> –next protein–
>
> *DNA polymerase*
>
> –next protein–
>
> *collagen*
>
> –next protein–
>
> *actin*
>

Now what about extending the ABox? Using DEEP SEMANTICS this is quite easy and does not require any workarounds. Like with any other RUBY class we use the constructor method *new()* of *Protein* and *BiologicalFunction* to create new instances of these classes. The ontology class *Protein* is the domain of the datatype property *hasMolecularWeight()*. DEEP SEMANTICS has incorporated this property into the code of *Protein* as RUBY instance method. We use this instance method to add the relevant molecular data to our new instance *insulin*. The code looks like this:

**Listing 3.7:** Creating a *Protein* and a *BiologicalFunction*

```
1 insulin = example::Protein.new("insulin")
2 insulin.hasMolecularWeight("5808") #in Dalton
3
4 glucoseRegulation = example::BiologicalFunction.new("GlucoseRegulation")
```

With the following statements we fetch the instance with the name *"insulin"* that we have just created and add to it the instance *glucoseRegulation* as biological function of insulin using the method *hasBiologicalFunction()*.

**Listing 3.8:** Adding information about its molecular function to insulin

```
1 insulin = iken.get_instance("insulin")
2
3 insulin.hasBiologicalFunction = glucoseRegulation
```

The next listing shows how to access information about molecular function of insulin and what happens if we want to add another molecular function to insulin assuming that *hasBiological-Function* is a functional property:

**Listing 3.9:** Trying to add a second molecular function to the instance *insulin*

```
1  puts "1. Insulin has the biological function:"
2  puts insulin.hasBiologicalFunction
3
4  puts # This produces just an empty line in the output.
5
6  glucoseTransportation = example::BiologicalFunction.new("GlucoseTransportation")
7  insulin.hasBiologicalFunction = glucoseTransportation
8
9  puts "2. Insulin has the biological function:"
10 puts insulin.hasBiologicalFunction
```

The output of listing 3.9:

> 1. Insulin has the biological function:
> #<DeepIntegrationBuilderOWLLite::Example::BiologicalFunction:0x137fff4 @local_name="GlucoseRegulation", @rdfs_comment_values=[] ...>
>
> 2. Insulin has the biological function:
> #<DeepIntegrationBuilderOWLLite::Example::BiologicalFunction:0x1354318 @local_name="GlucoseTransportation", @rdfs_comment_values=[] ...>

DEEP SEMANTICS has overridden the saved value of *hasBiologicalFunction* for *insulin* because *hasBiologicalFunction* is a functional object property and can therefore have only one value – in DEEP SEMANTICS this is always the latest added one. An interesting side note: the example of listing 3.9 demonstrates that using ontology languages does not avoid the emerging of semantic failures with respect to human interpretation capabilities. That insulin has the biological function to regulate the concentration of glucose in the blood and not to transport glucose has not been explicitly modeled in this example and therefore cannot be known by the system. A solution could be for example to create the *BiologicalFunction* subclasses *GlucoseTransportation* and *GlucoseRegulation*, the *Protein* subclass *InsulinFamily* and to constrain the *range* of *hasBiologicalFunction* with *all-values-from* to instances of the class *GlucoseRegulation* if used with instances of *InsulinFamily*. Then we could create an instance of *GlucoseRegulation* modeling *glucoseRegulation* and an instance of *InsulinFamily* modeling insulin. The assertion of *glucoseTransportation* as being the biological function of insulin would then not be allowed by DEEP SEMANTICS as it would lead to an inconsistency (the value of *hasBiologicalFunction* has to be an instance of *GlucoseRegulation*).

Ontology properties are stored in DEEP SEMANTICS as object types of the DEEP SEMANTICS classes *DatatypeProperty* or *ObjectProperty*. One can access these properties at runtime for example as shown in listing 3.10:

**Listing 3.10:** Access of a property RUBY object in DEEP SEMANTICS

```
1  hasBiologicalFunction_property = example.send("hasBiologicalFunction")
2
3  puts "Domain of the hasBiologicalFunction_property property is:"
4  puts hasBiologicalFunction_property.domain
```

The following is the output of listing 3.10:

> Domain of the hasBiologicalFunction property is:
> DeepIntegrationBuilderOWLLite::Example::Protein

### 3.7.3 XPERIMENTR– A Simple Semantic Application using DEEP SEMANTICS

In this section we will get an impression on how to use DEEP SEMANTICS to build a simple semantic application. This application called XPERIMENTR can be operated via the command line. The domain of the ontology we use as knowledge basis of the application is field of laboratory experiment planning. More precisely the domain ontology covers protocols, laboratory materials and equipment. The XPERIMENTR application can be applied to retrieve information about concepts and instances in the domain as well as to determine which protocols can be executed with a given set of laboratory materials. The XPERIMENTR ontology as well as the corresponding application were developed during this thesis. Information about protocols, laboratory equipment etc. were taken from the *Science 2.0* (Yoder & Shneiderman, 2008) website *OpenWetWare* [1]. Stated execution times for protocols are approximate estimates and do not origin from *OpenWetWare*.

#### The XPERIMENTR ontology

The detailed list of classes, local restrictions, properties and instances of the XPERIMENTR ontology are listed up in the appendix in Section 6.1 on page 135. The central concept of the XPERIMENTR ontology is *Protocol*. Covered types of protocols are currently restricted to buffers, growth media, in vivo and in vitro experiments. Included laboratory material concepts are *Antibiotic*, *Buffer*, *Chemical*, *Enzyme*, *Medium* (growth media like for example for bacterial cell cultures) and *Organism*. Furthermore, the classes *Person* and *LaboratoryEquipment* are included. For a future version one could extend the classes for example with the concept bioinformatics protocol or further subclass specifications protocol types already covered.

XPERIMENTR ontology includes the OWL LITE possibility to define inverse properties quite often. The benefits of this is that a reasoner can use these statements to deduce relations that therefore do not have to be stated explicitly. An example for an inverse property inference is:

$$isExpertOf(Indra, KnightColonyPCR)$$

$$\Rightarrow relatedExpert(KnightColonyPCR, Indra)$$

The used datatype properties are all constrained by the global restriction "functional property" as it would just lead to more complexity without a substantial quality gain to add for example more than one lists of procedures (property *"hasProcedure"*).

The ontology contains 57 instances. Included instances comprise laboratory equipment like Bunsen burner, micropipettes, incubators or PCR machines. Additionally, the ontology contains instances of class *LaboratoryMaterial* – for example acrylamide, TAE buffer, EDTA and

[1] http://openwetware.org/wiki/Main_Page

proteinase K. The instances are related to each other using defined object and datatype properties. The in vitro protocol for SDS-PAGE, for example, is related to the laboratory material acrylamide using the object property *hasRequiredMaterial*.

### The XPERIMENTR Application

The implementation of the XPERIMENTR example is intended to give an introduction into coding with deep-integrated ontologies as provided by DEEP SEMANTICS. Furthermore, will the described source code be used as reference material for framework comparison in Section 3.8 on page 81. The implemented functionalities include the retrieval of information about a concept or instance in the ontology, the search for protocols which include a list of user defined laboratory materials, and the search for protocols that can be executed in a given space of time. The implementation is described in detail in the following.

**Listing 3.11:** Using DEEP SEMANTICS to prepare the ontology knowledge base of XPERIMENTR

```
1  require File.join(File.dirname(__FILE__), 'active_semantics.rb')
2  require File.join(File.dirname(__FILE__), 'Helper/namespaces.rb')
3
4  Namespaces.add_namespace('http://www.ontovers.org/xperimentr.owl#', 'xperimentr:')
5
6  $deep_semantics = DeepSemantics.instance
7  xperimentr = $deep_semantics.set_director({'adapter' => 'FileAdapter', 'ontology_source' => '
       xperimentr_inferred.nt', 'builder' => 'DeepIntegrationBuilderOWLLite'})
8
9  puts "Welcome to Xperimentr your wetlab advisor. You may:"
10 puts "1. Enter \"?\" to retrieve information about a certain term."
11 puts "2. Enter \"find\" to find an experiment protocol for the materials you have available."
12 puts "3. Enter \"time\" to find an experiment protocol by execution time."
13 puts "4. Enter \"quit\" to exit."
```

Listing 3.11 shows the code for the preparation of the ontology knowledge base using DEEP SEMANTICS as well as our welcome message for the user which is the following one:

> "Welcome to Xperimentr your wetlab advisor. You may:"
> 1. Enter "?" to retrieve information about a certain term.
> 2. Enter "find" to find an experiment protocol for the laboratory materials you have available.
> 3. Enter "time" to find an experiment protocol by execution time.
> 4. Enter "quit" to exit.

The following listing 3.12 shows the basic control of the user input processing workflow. With the function *gets* one can get the latest user input from the command line. This input is then passed to the variable *line*. Depending on the user input being *"?"*, *"find"*, *"time"* or *"quit"* the corresponding code block is executed. If *line* does not match any of the compare strings an error message is prompted.

**Listing 3.12:** Accepting and processing user input

```
1  while line = gets
2  case line
```

```
3  when  "?\n"
4    # Call method for concept and instance information retrieval
5    retrieveInformation()
6  when "find\n"
7    # Call method to determine materials fitting protocols
8    findProtocolsByExperimentMaterials()
9  when "time\n"
10   # Call method to fetch protocols by execution time threshold
11   findProtocolsByExecTime()
12 when "quit\n"
13   break
14 else
15   puts
16   puts
17   puts "We are sorry but your input could not be processed. Please, try again:"
18   puts "1. Enter \"?\" to retrieve information about a certain term."
19   puts "2. Enter \"find\" to find an experiment protocol for the materials you have available.
       "
20   puts "3. Enter \"time\" to find an experiment protocol by execution time."
21   puts "4. Enter \"quit\" to exit."
22   puts
23 end
```

If the user selects *"?\n"* the controller calls the method *retrieveInformation()* that implements the functionality to search the ontology for information about a user requested search term. Listings 3.13, 3.14, 3.15 and 3.16 contain the implementation code of *retrieveInformation()*.

**Listing 3.13:** Implementation of the information retrieval method *retrieveInformation()*: search for matching classes segment

```
1  puts "Please enter the term you what to get information about:"
2
3  term = gets.chop # Cut off the next line character
4
5  # First we do look up any existing classes with the search term as label.
6  class_hits = xperimentr.find_classes_by_label(term)
7
8  if class_hits.size > 0
9    puts "Some information about #{term}:"
10   class_hits.each do |klass_hit|
11     puts klass_hit.rdfs_comment[0]
12     klass_hit.super_classes.each do |super_klass|
13       puts "#{term} is a #{super_klass.rdfs_label[0]}."
14     end
15   end
16 end
```

Listing 3.13 shows the implementation of the user input request as well as the searching for a class that has a *rdfs:label* matching the search term. To find these classes we use the DEEP SEMANTICS method *find_classes_by_label(term)* (line 6 in listing 3.13) provided by the functional ontology model *xperimentr* passing the current search term. The return value of *find_classes_by_label(term)* is an array (*class_hits*) of classes which satisfy the query.

If the size of this array is greater than 0 (line 8 in listing 3.13) the application prints out some information about each of these classes hits using the first *rdfs:comment* (line 11 in listing 3.13) and possibly available superclass names (lines 12 to 14 in the listing).

**Listing 3.14:** Implementation of the information retrieval method *retrieveInformation()*: search for matching laboratory material instances

```
1  # Then we do look up any existing instances with the search term as label. As we have three
       root classes in our ontology we can search the term in the labels of the instances for
       each of these root classes.
2
3  laboratory_material_hits = xperimentr::LaboratoryMaterial.find_instances_by_label(term)
4
5  if laboratory_material_hits.size > 0
6    puts "Some information about the laboratory material #{term}:"
7    laboratory_material_hits.each do |laboratory_material_hit|
8      if laboratory_material_hit.rdfs_comment.size > 0
9        puts laboratory_material_hit.rdfs_comment[0]
10     end
11     laboratory_material_hit.types.each do |type|
12       puts "#{term[0,1].capitalize+term[1,term.length]} is a kind of #{type.rdfs_label[0]}."
13       if type.super_classes.size > 0
14         type.super_classes.each do |superclass|
15           puts "#{term[0,1].capitalize+term[1,term.length]} is a kind of #{superclass.
                 rdfs_label[0]}."
16         end
17       end
18     end
19     puts "#{term[0,1].capitalize+term[1,term.length]} is used for the following protocols:"
20     laboratory_material_hit.isUsedForProtocol.each do |protocol|
21       puts protocol.rdfs_label[0]
22     end
23   end
24 end
```

Implementing the code of listing 3.14 XPERIMENTR provides the functionality to search for instances of the class *LaboratoryMaterial*. This is accomplished by calling the method *find_instances_by_label(term)* of the DEEP SEMANTICS integrated representation of the ontology class. The RUBY implementation of *LaboratoryMaterial* is accessible via the ontology's namespace *xperimentr* (line 3 in the listing). The return value is then passed to the variable *laboratory_material_hits*.

In the case that the size of the array *laboratory_material_hits* is greater than 0 the system prints out detailed information stored in the ontology about the found laboratory materials (lines 5 to 25). With method call *laboratory_material_hits.types* in line 11 one can fetch all the corresponding classes the instance belongs to. For the instance *ProteinaseK* for example the return value of *types* is an array containing the ontology classes *Enzyme* and *LaboratoryMaterial*. With *type.super_classes* (line 14) the superclasses of each *type* are retrieved and then printed out (line 15). The method call *laboratory_material_hit.isUsedForProtocol()* in line 21 returns an array of instances of class *Protocol* for which the laboratory material is required. Each of this returned protocols, respectively their first *rdfs:label* entries, is then printed out (line 22).

**Listing 3.15:** Implementation of the information retrieval method *retrieveInformation()*: search for matching laboratory equipment instances

```
1  laboratory_equipment_hits = xperimentr::LaboratoryEquipment.find_instances_by_label(term)
2
3  if laboratory_equipment_hits.size > 0
4    puts "Some information about the laboratory equipment #{term}:"
5    laboratory_equipment_hits.each do |laboratory_equipment_hit|
6      if laboratory_equipment_hit.rdfs_comment.size > 0
7        puts laboratory_equipment_hit.rdfs_comment[0]
8      end
9      laboratory_equipment_hit.types.each do |type|
10       puts "#{term[0,1].capitalize+term[1,term.length]} is a kind of #{type.rdfs_label[0]}."
```

```
11        if type.super_classes.size > 0
12          type.super_classes.each do |superclass|
13            puts "#{term[0,1].capitalize+term[1,term.length]} is a kind of #{superclass.
                  rdfs_label[0]}."
14          end
15        end
16      end
17      puts "#{term[0,1].capitalize+term[1,term.length]} is required for protocols:"
18      laboratory_equipment_hit.isRequiredForProtocol.each do |protocol|
19        puts protocol.rdfs_label[0]
20      end
21    end
22  end
```

The code of listing 3.15 implements the information retrieval workflow for laboratory equipment instances. This code is largely similar to the implementation shown in listing 3.14. Enabled through the transformations performed by DEEP SEMANTICS the searched instances of laboratory equipment are retrieved with *xperimentr::LaboratoryEquipment.find_instances_by_label(term)* (line 1). Further direct utilization of DEEP SEMANTICS can be seen for example in line 9 where the class types of the retrieved laboratory equipment instances are fetched using *laboratory_equipment_hit.types()* and in line 18 where protocols are retrieved that require the corresponding laboratory equipment (*laboratory_equipment_hit.isRequiredForProtocol()*).

**Listing 3.16:** Implementation of the information retrieval method *retrieveInformation()*: search for matching *Protocol* instances

```
1   protocol_hits = xperimentr::Protocol.find_instances_by_label(term)
2
3   if protocol_hits.size > 0
4     puts "Some information about the protocol #{term}:"
5     protocol_hits.each do |protocol_hit|
6       if protocol_hit.rdfs_comment.size > 0
7         puts protocol_hit.rdfs_comment[0]
8       end
9       protocol_hit.types.each do |type|
10        puts "#{term[0,1].capitalize+term[1,term.length]} is a kind of #{type.rdfs_label[0]}."
11        if type.super_classes.size > 0
12          type.super_classes.each do |superclass|
13            puts "#{term[0,1].capitalize+term[1,term.length]} is a kind of #{superclass.
                  rdfs_label[0]}."
14          end
15        end
16      end
17      if protocol_hit.hasRequiredLaboratoryEquipment && (protocol_hit.
           hasRequiredLaboratoryEquipment.size > 0)
18        puts "#{term[0,1].capitalize+term[1,term.length]} has required laboratory equipment:"
19        protocol_hit.hasRequiredLaboratoryEquipment.each do |equipment|
20          puts equipment.rdfs_label[0]
21        end
22      end
23      puts "#{term[0,1].capitalize+term[1,term.length]} requires material:"
24          if protocol_hit.hasRequiredMaterial && (protocol_hit.hasRequiredMaterial.size > 0)
25        protocol_hit.hasRequiredMaterial.each do |material|
26          puts material.rdfs_label[0]
27        end
28      end
29      puts "#{term[0,1].capitalize+term[1,term.length]} has the following procedure:"
30      puts protocol_hit.hasProcedure[0]
31      if protocol_hit.hasRequiredExecutionTime && (protocol_hit.hasRequiredExecutionTime.size >
           0)
```

```
32        puts "#{term[0,1].capitalize+term[1,term.length]} has required execution time in minutes
              : #{protocol_hit.hasRequiredExecutionTime}"
33      end
34      puts "The following experts are related to #{term[0,1].capitalize+term[1,term.length]}:"
35      protocol_hit.relatedExpert.each do |expert|
36        puts expert.rdfs_label[0]
37      end
38    end
39  end
```

The last code segment of the information retrieval method implementation is shown in listing 3.16. Analogous to listings 3.14 and 3.15 the DEEP SEMANTICS transformed methods *xperimentr::Protocol.find_instances_by_label(term)* (in line 1) and *protocol_hit.types()* (in line 9) are used to process the related ontology constructs. Additional uses of DEEP SEMANTICS in this listing are:

- Line 17: The method *protocol_hit.hasRequiredLaboratoryEquipment()* is used to fetch all required laboratory equipment instances for the selected protocol from the ontology.
- Line 24: In the ontology every protocol is related to zero or more laboratory materials using the object property *hasRequiredMaterial*.
- Line 30: With *protocol_hit.hasProcedure[0]* a string representing the experiments procedure is accessed. The return value is an array and because we have defined *hasProcedure* as being a functional property (that means it has at maximum one value) one can retrieve the unique property value with *"hasProcedure[0]"*.
- Line 31: The datatype property *hasRequiredExecutionTime* is used to save the experiment's execution time in minutes. Using the DEEP SEMANTICS generated method *protocol_hit.hasRequiredExecutionTime()* one can retrieve this execution time.

**Listing 3.17:** Implementation of the XPERIMENTR method *findProtocolsByExperimentMaterials()*

```
1   materials = Array.new
2   found_protocols = xperimentr::Protocol.instances
3
4   puts "Please enter the next laboratory material or \"end\" to stop:"
5
6   while material_input = gets
7     case material_input
8     when "end\n"
9       break
10    else
11      material_input = material_input.chop
12      if xperimentr::LaboratoryMaterial.find_instances_by_label(material_input).size > 0 #If a
            matching material exists in the ontology
13        materials << material_input
14        new_found_protocols = Array.new
15        found_protocols.each do |found_protocol|
16          if found_protocol.hasRequiredMaterial && (found_protocol.hasRequiredMaterial.size > 0)
17            found_protocol.hasRequiredMaterial.each do |material|
18              if material.rdfs_label.include?(material_input)
19                new_found_protocols << found_protocol
20              end
21            end
22          end
23        end
```

```
24        found_protocols = new_found_protocols
25        puts "You have entered the following materials:"
26        materials.each do |material|
27          puts material
28        end
29        puts "The following protocols include these materials:"
30        found_protocols.each do |found_protocol|
31          puts found_protocol.rdfs_label[0]
32        end
33      else
34        puts "#{material_input} is not a known laboratory material."
35      end
36      puts "Please enter the next material or \"end\" to stop:"
37    end
38  end
```

The second interaction mode offers the possibility to search for protocols that include a set of inputed laboratory materials. Listing 3.17 shows the code of method *findProtocolsByExperimentMaterials()* that implements this function. The array *materials* (line 1) saves the materials entered by the user during a search session. The statement *"found_protocols = xperimentr::Protocol.instances"* in the next line utilizes DEEP SEMANTICS to retrieve all instances of the class *Protocol* and stores the return values in *found_protocols*.

From lines 6 to 38 a while loop is used to capture user entered laboratory materials that are then used to determine appropriate protocols. The user command *"end"* tells the application to leave the current while loop. If the word that was entered by the user is not equal to *"end"* then the application tests whether or not this word can be associated to a known *LaboratoryMaterial* instance using *"if xperimentr::LaboratoryMaterial.find_instances_by_label(material_input).size > 0"* in line 12. If the input term can be mapped to at least one instance of *LaboratoryMaterial* then a) we save this term in the array *materials* (line 13), b) we look up if any protocol meets the required material conditions (lines 14 to 24) and c) we print out the set of entered laboratory materials (lines 25 to 28) as well as all matching protocols or their first *rdfs:label* entries (line 29 to 32), respectively.

**Listing 3.18:** Implementation of the XPERIMENTR method *findProtocolsByExecTime()*

```
1   puts "Please enter the maximum execution time in minutes acceptable for you:"
2
3   time = gets.chop
4
5   puts "Protocols that can be executed in #{time} minutes:"
6   xperimentr::Protocol.instances.each do |protocol|
7     if protocol.hasRequiredExecutionTime && (protocol.hasRequiredExecutionTime.size > 0)
8       if protocol.hasRequiredExecutionTime[0].to_i < time.to_i
9         puts "#{protocol.rdfs_label[0]} can be prepared in #{protocol.hasRequiredExecutionTime
              [0].to_i} minutes."
10      end
11    end
12  end
```

If the user enters the command *"time"* in the main program control loop, the method *findProtocolsByExecTime()* is called. The code of this method can be seen in listing 3.18. The processing steps of this method are:

- Line 3: Capturing the maximum execution time in minutes entered by a user.

- Lines 6 to 12: Iterate over all instances of class *Protocol* in order to ...
  - Line 7: ...to check with *"protocol.hasRequiredExecutionTime && (protocol.hasRequiredExecutionTime.size > 0)"* if the protocol has an execution time value and ...
  - Line 8: ...to test if an existing execution time value is less or equal to the user given threshold (if this test returns *"TRUE"* the protocol and its execution time is printed out).

Summing up, this subsection DEEP SEMANTICS has been successfully applied for a variety of ontology processing tasks. Deep-integrated RUBY representations of ontology classes were generated using DEEP SEMANTICS (see listing 3.11). These transformed ontology classes were then applied to access their corresponding instances (like for example using *laboratory_material_hits = xperimentr::LaboratoryMaterial.find_instances_by_label(term)* at line 3 of listing 3.14).

Beside applying deep-integrated classes, working with ontology instances was important for the realization of listings 3.14 to 3.18. DEEP SEMANTICS enables ontology instance objects that provide methods to access related object and datatype properties (for example the access of the deep-integrated datatype property *protocol.hasRequiredExecutionTime* at line 6 of listing 3.18 or the use of the integrated object property *found_protocol.hasRequiredMaterial* at line 16 of listing 3.17).

Additionally, DEEP SEMANTICS provides convenient methods to access ontology constructs using label matching. The method call *xperimentr::Protocol.instances* (line 2 in listing 3.17) for example returns all instances of the ontology class *Protocol*. Likewise ontology classes themselves were retrieved by matching a given search string with the class *rdfs:label* values (such as *xperimentr.find_classes_by_label(term)* in line 6 of listing 3.13).

### XPERIMENTR in action

In the previous subsection we learned about how XPERIMENTR has been implemented using DEEP SEMANTICS together with its required ontology knowledge base. In this subsection a typical user session with the XPERIMENTR application is described. Each session starts with a welcome message of the application:

> Welcome to Xperimentr your wetlab advisor. You may:
> 1. Enter "?" to retrieve information about a certain term.
> 2. Enter "find" to find an experiment protocol for the materials you have available.
> 3. Enter "time" to find an experiment protocol by execution time.
> 4. Enter "quit" to exit.

We choose to retrieve some information about the term *"SDS-PAGE"* and enter *"?"*. The next message of the system is:

> Please enter the term you what to get information about:

We enter *"SDS-PAGE"* and the system responses information stored in the ontology about the corresponding instance *SDSPAGE*:

> Some information about the protocol SDS-PAGE:
> "The pH of the separating gel in standard SDS-PAGE (a.k.a. Laemmli buffer system) is roughly 8-9 which is conducive to the deamination and alkylation of proteins, as well as reoxidation of reduced cysteines during electrophoresis. What this means is that your protein will form disulfide crosslinks during the stacking event because the protein migrates into the gel away from the reducing reagent in the sample buffer, and gets focused to a high concentration."
>
> SDS-PAGE is a kind of in vitro protocol.
> SDS-PAGE is a kind of protocol.
>
> SDS-PAGE requires material:
> sodium bisulfite
> 5x high-MW running buffer
> 5x low-MW running buffer
> acrylamide
> 3.5X bis-Tris gel buffer
>
> SDS-PAGE has the following procedure:
> Resolving:
Mix: 1/3.5 vol. of 3.5X bis-Tris gel buffer, acrylamide to 8% (30:2.0) or 12-15% (30:0.8), and water to final volume. I make 3.75 mLs for each Bio-Rad Protein gel, and use 3.5 mLs per gel. . . .
> SDS-PAGE has required execution time in minutes: 333
> The following experts are related to SDS-PAGE: Ilija Nahal Deniz Tim Indra

This informational response is followed by a reminder what the user can do next:

> What do you want to do next? You may:
> 1. Enter "?" to retrieve information about a certain term.
> 2. Enter "find" to find an experiment protocol for the materials you have available.
> 3. Enter "time" to find an experiment protocol by execution time.
> 4. Enter "quit" to exit.

We enter the command *"find"* and read the response:

> Please enter the next laboratory material or "end" to stop:

We enter the term *"sodium bisulfite"* and get the response:

> You have entered the following materials:
> sodium bisulfite
>
> The following protocols include these materials:
> 5X low-MW running buffer protocol
> 5X high-MW running buffer protocol
> SDS-PAGE
>
> Please enter the next material or "end" to stop:

Next we enter the term *"acrylamide"* and get the response:

> You have entered the following materials:
> sodium bisulfite
> acrylamide
>
> The following protocols include these materials:
> SDS-PAGE
>
> Please enter the next material or "end" to stop:

We enter *"end"* to stop this session and see once again the four basic selection possibilities of XPERIMENTR. This time we enter the *"time"* command to find protocols by their execution time. The system prompts:

> Please enter the maximum execution time in minutes acceptable for you:

After entering *"240"* (this could be useful for example if a biologist wants to know which protocols he can still execute in the residual labour time) we get the information:

> Protocols that can be executed in 240 minutes:
> Blackburn yeast colony PCR can be prepared in 70 minutes.
> mouse tissue lysis for genotyping can be prepared in 88 minutes.
> Knight Colony PCR can be prepared in 200 minutes.
> Affymetrix DNA labelling for gene expression arrays can be prepared in 50 minutes.
> PCR supermix protocol can be prepared in 20 minutes.

Now, that we have discussed every functionality of the system we enter *"quit"* to leave the program. The following section compares the most commonly used Semantic Web frameworks JENA2, OWL API and ACTIVERDF with DEEP SEMANTICS.

## 3.8 Comparison of DEEP SEMANTICS with other Semantic Web Frameworks

Specialized ontology processing frameworks build the foundation of the upcoming Semantic Web. Therefore, in recent years several frameworks have been published. Three of them are compared with DEEP SEMANTICS in this section. Two of these frameworks, JENA2 and OWL API, are implemented in JAVA, while the third one (ACTIVERDF) is developed using RUBY.

The JENA2 framework is open source and conceptionally centered on the RDF graph (see Subsection 2.7.1 on page 20). While offering similar functionalities, OWL API's architecture follows an axiomatic approach (introduced in Subsection 2.7.2 on page 21) regarding access and modification of ontology constructs. The third framework, ACTIVERDF (described in Subsection 2.7.3 on page 21), is most similar to DEEP SEMANTICS of the three compared frameworks. DEEP SEMANTICS and ACTIVERDF are developed in RUBY using the scripting language's metaprogramming features and do not offer support for TBox modification.

The comparisons are based on the XPERIMENTR implementations (for programming complexity comparison) and on two different reference tests, for which solutions were programmed for each framework. Each of these implemented solutions was then executed twice: firstly, using an inferred version of the **IKEN** ontology (described on page 102) and secondly, using an also inferred version of the **BIO2ME** ontology (Mainz, 2008). Ontology metrics are: 1) **IKEN** ontology comprising 366 named classes, 86 object properties, 23 datatype properties and 859 individuals. 2) **BIO2ME** ontology consisting of 213 named classes, 41 object properties, 29 datatype properties and 343 individuals.

The first reference test requires the corresponding solutions to list all classes of the ontology by their hierarchy level. Programming solutions for this type of test were realized for JENA2, OWL API and DEEP SEMANTICS, while ACTIVERDF does not provide necessary functionalities to determine the class hierarchy and therefore could not be used in this test. The second reference test is described as: "find all instances that match one of ten given search terms by using RDFS labels of instances". This test was performed in all four frameworks.

DEEP SEMANTICS was compared to the three other frameworks regarding programming as well as runtime and memory complexity. Programming complexity in this context refers to the amount of required code lines needed for an implementation. The different test implementations can be found in the appendix in Section 6.2. All tests were carried out on an Apple MacBook Pro© (Mac OS X© version 10.5.5) using a 2.33 GHz Intel Core 2 Duo© and 2 GB 667 MHz DDR2 SDRAM and RUBY in version 1.8.6.

### 3.8.1 DEEP SEMANTICS versus OWL API

While DEEP SEMANTICS is implemented in RUBY, and OWL API in JAVA, both frameworks are oriented to OWL's abstract syntax. A principal difference is the focusing of DEEP SEMANTICS on the editing of the ABox in contrast to ABox and TBox modifications in

OWL API. Summing up, the OWL API cannot be *consistency safe* in principal as this would not allow TBox modifications, which are necessary to further extend the set of classes and their interrelations.

### Programming Complexity Comparison

In this subsection DEEP SEMANTICS is compared to OWL API with respect to the programming complexity or coding complexity, respectively. The comparison was performed using framework specific implementations of the XPERIMENTR example introduced in Section 3.7.3 on page 71.

Listing 3.21 shows the required source code to print out the ontology class hierarchy to standard out using DEEP SEMANTICS. Listings 3.19 and 3.20 contain the corresponding implementation based on OWL API. The most obvious difference is the lack of convenient methods like DEEP SEMANTICS's *listClassesByLevel* in OWL API. This method returns a RUBY hash containing hierarchy levels as keys and corresponding classes of these levels as key related values. Lines 43 to 48 in listing 3.20 display the source code for traversing the class hierarchy using OWL API. As method *getSubClasses(this.ontology)* returns all subclasses (direct and indirect) including B-Nodes (restrictions), programmers have to implement the following checks: A) if the subclass is not equal to the calling class (*!child.equals(clazz)*), B) if the subclass is not a B-Node *!child.isAnonymous()* and C) if subclass is not equal to *OWL Nothing* (!child.isOWLNothing()). In contrast, in DEEP SEMANTICS only the class method *direct_sub_classes* has to be called, which returns all direct subclasses excluding any B-Nodes.

Another frequently required task is the computation of all direct instances of an ontology class as shown in lines 22 to 37 of listing 3.20. This is also significantly more complicated by using OWL API. This framework's class *OWLClass* provides the method *getIndividuals(this.ontology)*, which returns all instances of the corresponding ontology class. However, calling this method returns instances of possibly existing subclasses, too. Consequently, filtering out only direct instances becomes quite complicated and involves processing of subclasses (starting line 26). Using DEEP SEMANTICS the retrieving of all direct instances is provided via the convenient method *direct_instances* as shown in line 11 of listing 3.21.

**Listing 3.19:** XPERIMENTR implementation using OWL API: Print out the class hierarchy of the ontology.

```
1  // Print out all of the classes which are referenced in the
2  // ontology by hierarchy level
3  System.out.println("Known classes:");
4
5  Set<OWLClass> klasses = ontology.getReferencedClasses();
6  for (OWLClass klass : klasses) {
7          try {
8                  xperimentr.printHierarchy(ontology, klass);
9          } catch (OWLException e) {
10                 System.out.println("The class hierarchy could not be printed: " + e.getMessage
                       ());
11         }
12 }
```

**Listing 3.20:** XPERIMENTR implementation using OWL API: *printHierarchy()* methods.

```
1  /**
2   * Print the class hierarchy for the given ontology from this class
3   * down, assuming this class is at the given level.
4   * Makes no attempt to deal sensibly with multiple inheritance.
5  */
6   public void printHierarchy(OWLOntology ontology, OWLClass clazz) throws OWLException {
7       this.ontology = ontology;
8       printHierarchy( clazz, 0 );
9   }
10
11   /**
12    * Print the class hierarchy from this class down, assuming this class is at
13    * the given level. Makes no attempt to deal sensibly with multiple
14    * inheritance.
15    */
16  public void printHierarchy(OWLClass clazz, int level) throws OWLException {
17           System.out.println("Level: "+level);
18           System.out.println("  "+clazz);
19
20           /* Find this classes instances*/
21           System.out.println("  This classes direct instances:");
22           Set<OWLIndividual> instances = clazz.getIndividuals(this.ontology);
23           for (OWLIndividual instance : instances) {
24                   Boolean is_direct_instance = true;
25
26                   Set<OWLDescription> subclasses = clazz.getSubClasses(this.ontology);
27                   for (OWLDescription subclass : subclasses) {
28                           if (!subclass.isAnonymous() && !subclass.equals(clazz) && !subclass.
                                isOWLNothing()) {
29                                   if (instance.getTypes(this.ontology).contains(subclass)) {
30                                           is_direct_instance = false;
31                                   }
32                           }
33                   }
34                   if (is_direct_instance) {
35                          System.out.println("    "+instance);
36                   }
37           }
38           System.out.println();
39           System.out.println();
40           System.out.println("--------------------------");
41
42       /* Find the children and recurse */
43       Set<OWLDescription> children = clazz.getSubClasses(this.ontology);
44       for (OWLDescription child : children) {
45          if (!child.equals(clazz) && !child.isAnonymous() && !child.isOWLNothing() ) {
46             printHierarchy(child.asOWLClass(), level + 1);
47          }
48       }
49   }
```

**Listing 3.21:** XPERIMENTR implementation using DEEP SEMANTICS: Print out the class hierarchy of the ontology.

```
1  # Print out all of the classes which are referenced in the
2  # ontology by hierarchy level
3  puts "Known classes:"
4
5  xperimentr.listClassesByLevel.each_pair do |level, classes|
6    puts "Level: #{level}"
7    classes.each do |klass|
8      puts "  "+klass.local_name
9      puts
10     puts "  This classes direct instances:"
```

```
11      klass.direct_instances.each do |instance|
12        puts "    "+instance.local_name
13      end
14      puts "--------------------------"
15      puts
16    end
17  end
```

Listing 3.16 on page 75 shows the implementation of the information retrieval method *retrieveInformation()* in the DEEP SEMANTICS implementation of XPERIMENTR. Method *find_instances_by_label(term)* returns, analogous to the above mentioned method *find_classes_by_label(term)* (line 6 of listing 3.13), all instances that contain a RDFS label, which matches the provided search term. Listing 3.23 in contrast contains the realization with the OWL API, which is more complicated due to the lack of a convenient method. Method *getIndividuals(ontology)* in line 3 returns all instances of the corresponding class in the passed ontology and assigns these to variable *instances*. For each OWL individual stored in *instances* (line 5) all RDFS labels are iterated (line 6). If the method call *annotation.isAnnotationByConstant()* (line 7) returns the boolean value **TRUE** that means the annotation value is not an entity (class, property or individual), method *getAnnotationValueAsConstant()* of the annotation is invoked (line 8). Additionally, the literal value of variable *value* is compared with the passed search label (line 9) and, if this comparison returns **TRUE**, variable *instance* is added to the *OWLIndividual* containing set *instance_hits* (at line 10).

Lines 6 to 8 of listing 3.23 for example can be substituted by one line in DEEP SEMANTICS by calling method *instance.rdfs_label*. This method call returns all strings stored as labels of the instance. Additionally, the complete listing can be substituted by calling *klass.find_instances_by_label*.

OWL API-based listing 3.22 comprises the implementation of information retrieval functionality related to the discovery of instances of the XPERIMENTR class *Protocol*. An particularly mentionable programming issue is the access of property values of OWL individuals. Lines 19 to 27 for example realize the fetching of values of the object property *hasRequiredLaboratoryEquipment* for the current instance of *Protocol* which is stored in variable *protocolHit*.

**Listing 3.22:** XPERIMENTR implementation using OWL API: information retrieval implementation for instances of ontology class *Protocol*.

```
1  OWLClass protocolKlass = factory.getOWLClass(URI.create(base + "#Protocol"));
2
3  Set<OWLIndividual> protocolHits = findInstancesByLabel(ontology, protocolKlass, searchTerm);
4
5  if (protocolHits.size() > 0) {
6          System.out.println( "Some information about the protocol "+searchTerm+":" );
7          for (OWLIndividual protocolHit : protocolHits) {
8                  if (protocolHit.getAnnotations(ontology, OWLRDFVocabulary.RDFS_COMMENT.getURI
                       ()).size() > 0) {
9                          OWLAnnotation comment = (OWLAnnotation) protocolHit.getAnnotations(
                               ontology, OWLRDFVocabulary.RDFS_COMMENT.getURI()).toArray()[0];
10                         System.out.println( comment.getAnnotationValueAsConstant().getLiteral
                               () );
11                 }
12
13                 for (OWLDescription type :protocolHit.getTypes(ontology) ) {
14                         if (!type.isAnonymous() && !type.isOWLThing() ) {
```

```
15                     OWLAnnotation label = (OWLAnnotation) type.asOWLClass().
                           getAnnotations( ontology, OWLRDFVocabulary.RDFS_LABEL.
                           getURI() ).toArray()[0];
16                     System.out.println( searchTerm+" is a kind of "+label.
                           getAnnotationValueAsConstant().getLiteral() );
17                 }
18         }
19 
20         Map<OWLObjectPropertyExpression,java.util.Set<OWLIndividual>>
               protocolHitObjectPropertyMap = protocolHit.getObjectPropertyValues(
               ontology);
21         OWLObjectPropertyExpression requiredLaboratoryEquipment = factory.
               getOWLObjectProperty(URI.create(base + "#hasRequiredLaboratoryEquipment"))
               ;
22         if ( protocolHitObjectPropertyMap.containsKey(requiredLaboratoryEquipment) ) {
23                 System.out.println(searchTerm+" has required laboratory equipment:");
24                 for (OWLIndividual equipment :protocolHitObjectPropertyMap.get(
                       requiredLaboratoryEquipment) ) {
25                     OWLAnnotation equipmentLabel = (OWLAnnotation) equipment.
                           getAnnotations( ontology, OWLRDFVocabulary.RDFS_LABEL.
                           getURI() ).toArray()[0];
26                     System.out.println( equipmentLabel.
                           getAnnotationValueAsConstant().getLiteral() );
27                 }
28         }
29 
30         OWLObjectPropertyExpression requiredMaterial= factory.getOWLObjectProperty(URI
               .create(base + "#hasRequiredMaterial"));
31         if ( protocolHitObjectPropertyMap.containsKey(requiredMaterial) ) {
32                 System.out.println(searchTerm+" has required material:");
33                 for (OWLIndividual material :protocolHitObjectPropertyMap.get(
                       requiredMaterial) ) {
34                     OWLAnnotation materialLabel = (OWLAnnotation) material.
                           getAnnotations( ontology, OWLRDFVocabulary.RDFS_LABEL.
                           getURI() ).toArray()[0];
35                     System.out.println( materialLabel.
                           getAnnotationValueAsConstant().getLiteral() );
36                 }
37         }
38 
39         Map<OWLDataPropertyExpression, Set<OWLConstant>>
               protocolHitDatatypePropertyMap = protocolHit.getDataPropertyValues(
               ontology);
40         OWLDataPropertyExpression procedure = factory.getOWLDataProperty(URI.create(
               base + "#hasProcedure"));
41         if ( protocolHitDatatypePropertyMap.containsKey(procedure) ) {
42                 System.out.println(searchTerm+" has the following procedure:");
43                 OWLConstant procedureString = (OWLConstant)
                       protocolHitDatatypePropertyMap.get(procedure).toArray()[0];
44                 System.out.println( procedureString.getLiteral()  );
45         }
46 
47         OWLDataPropertyExpression hasRequiredExecutionTime = factory.
               getOWLDataProperty(URI.create(base + "#hasRequiredExecutionTime"));
48         if ( protocolHitDatatypePropertyMap.containsKey(hasRequiredExecutionTime) ) {
49                 OWLConstant executionTime = (OWLConstant)
                       protocolHitDatatypePropertyMap.get(hasRequiredExecutionTime).
                       toArray()[0];
50                 System.out.println( searchTerm+" has required execution time in
                       minutes: "+executionTime.getLiteral()  );
51         }
52 
53         OWLObjectPropertyExpression relatedExpert= factory.getOWLObjectProperty(URI.
               create(base + "#relatedExpert"));
54         if ( protocolHitObjectPropertyMap.containsKey(relatedExpert) ) {
55                 System.out.println();
```

```
56                          System.out.println("The following experts are related to "+searchTerm+
                                ":");
57                          for (OWLIndividual expert :protocolHitObjectPropertyMap.get(
                                relatedExpert) ) {
58                                  OWLAnnotation expertLabel = (OWLAnnotation) expert.
                                        getAnnotations( ontology, OWLRDFVocabulary.RDFS_LABEL.
                                        getURI() ).toArray()[0];
59                                  System.out.println( expertLabel.getAnnotationValueAsConstant
                                        ().getLiteral() );
60                          }
61                  }
62          }
63  }
```

**Listing 3.23:** XPERIMENTR implementation using OWL API: *findInstancesByLabel()* method.

```
1  public static Set<OWLIndividual> findInstancesByLabel(OWLOntology ontology, OWLClass klass,
       String searchLabel) {
2          Set<OWLIndividual> instance_hits = new HashSet<OWLIndividual>();
3          Set<OWLIndividual> instances = klass.getIndividuals(ontology);
4
5          for (OWLIndividual instance : instances) {
6                  for (OWLAnnotation annotation : instance.getAnnotations(ontology,
                       OWLRDFVocabulary.RDFS_LABEL.getURI())) {
7                          if (annotation.isAnnotationByConstant()) {
8                                  OWLConstant value = annotation.getAnnotationValueAsConstant();
9                                  if ( value.getLiteral().equals(searchLabel) ) {
10                                         instance_hits.add( instance );
11                                 }
12                         }
13                 }
14         }
15
16         return instance_hits;
17 }
```

**Listing 3.24:** DEEP SEMANTICS intern implementation of method *find_instances_by_label(label)*.

```
1 def self.find_instances_by_label(label)
2   results = Array.new
3   self.instances.each do |instance|
4     if instance.rdfs_label.include?(label)
5       results << instance
6     end
7   end
8   return results
9 end
```

In summary, comparing the programming complexity of both frameworks reveals that implementations based on DEEP SEMANTICS need considerably less lines of code. Counting only the functional lines (not comments or empty lines) comparison metrics are: *findInstancesByLabel* functionality 15 lines for OWL API (listing 3.23) compared to nine lines for the DEEP SEMANTICS intern realization of *find_instances_by_label(label)* (see listing 3.24) and one line when using *find_instances_by_label(label)* in a DEEP SEMANTICS based semantic application.

Likewise, the OWL API based implementation of listing 3.22 requires 51 lines in contrast to 32 lines for the DEEP SEMANTICS-based listing 3.16. The difference in the number of needed lines is even higher when considering the 15 lines of *findInstancesByLabel* implementation shown in listing 3.23.

#### Runtime and Memory Complexity Comparison

The results of the runtime and memory complexity comparison between DEEP SEMANTICS and OWL API are shown in Table 3.3. Noticeable for DEEP SEMANTICS, is the immense difference between the complete runtime and the required time for the corresponding operations. For example, the test *"List classes by level:* **IKEN** *ontology"* has a complete runtime of 47.080 seconds, while the list operation runtime is 0.021 seconds, meaning that the list operation is more than a 1,000 times faster than the complete runtime. This large difference in runtime is explained by the required deep integration processing steps.

In a direct comparison between DEEP SEMANTICS and OWL API, the complete runtimes are approximately 5 to 10 times longer for DEEP SEMANTICS. However, computing of the operations alone is around 20 to 200 times faster for DEEP SEMANTICS. This is especially interesting as in benchmarks[2] JAVA is still significantly faster than RUBY. Concerning peek main memory requirements DEEP SEMANTICS needs around three times more memory. In contrast the average main memory usage of DEEP SEMANTICS is almost equal to OWL API's peak memory consumption.

### 3.8.2 DEEP SEMANTICS versus JENA2

JENA2 also is developed in JAVA. Analogous to OWL API JENA2 supports ABox and TBox modifications. As strict *consistency safeness* would not allow TBox editing, JENA2 is therefore also not *consistency safe*.

#### Programming Complexity Comparison

Listing 3.21 on page 83 shows the source code for an output of the class hierarchy of the XPERIMENTR ontology. The easy access of the class hierarchy by calling the method *listClassesByLevel* in line five is characteristical for the DEEP SEMANTICS framework. A similar example of DEEP SEMANTICS' simplicity is the method *direct_instances* in line eleven. This method returns all direct instances of the calling ontology class.

Listings 3.25 and 3.26 show the implementation of an analogous functionality with JENA2. Listing 3.25 contains the source code for the iteration through all classes of the ontology. As JENA2 does not offer a direct way to determine the level of a class one has to check: 1) if the current class is a root class using *klass.isHierarchyRoot()* and 2) with *(klass.listSuperClasses(true).toSet().size() > 0)* that the current class is not equal to *owl:Thing*. Listing 3.26 comprises the required code for method *printHierarchy(OntClass klass, int level)*. This method prints out the class hierarchy from the passed class down.

Comparing the DEEP SEMANTICS implementation with JENA2-based listings 3.25 and 3.26, the major difference is JENA2's lack of the useful convenient function *listClassesByLevel*. Additionally, implementing *printHierarchy(OntClass klass, int level)* is comparatively complex

[2] http://shootout.alioth.debian.org/

**Table 3.1: Types of global property constraints that have to be considered by DEEP SEMANTICS**

<table>
<tr><th>Type of Constraint</th><th>Comments and Corresponding RUBY Code generated by DEEP SEMANTICS</th></tr>
<tr><td>Functional Property</td><td>

**Comment:** A functional property can have only one unique value **y** for each instance **x** that uses this property. If e. g. the functional property **p()** is used with instance **x** as subject of the statements, and there are instances **y1** and **y2** used in the statements **p(x, y1)** and **p(x, y2)** then **y1** and **y2** have to be equal or the ontology is inconsistent. To avoid such inconsistencies during runtime DEEP SEMANTICS checks in the deep integration process if a property or one of its super-properties is functional and if so sets the temporary variable *max* to 1. Afterwards *max* is used for generating the particular setter method.<br>
**Meta-Programming Code:** Please read the maximum cardinality description in Table 3.2 for details.

</td></tr>
<tr><td>Inverse Functional Property</td><td>

**Comment:** An inverse functional property is a property where the object of a property statement uniquely determines the subject's individual. If one states e. g. that **p()** is an inverse functional property, then this asserts that **y** can only be the value in a statement using **p()** for a single instance **x**. That means that if there would exist instances **x1** and **x2** and statements **p(x1, y)** and **p(x2, y)** under the given conditions then either **x1** and **x2** have to be equal instances or the ontology is inconsistent.<br>
**Meta-Programming Code:**<br>
*Setter* source code extension:

```
"def #{object_property_object.local_name}=(new_value)
  if new_value.kind_of?(Thing)
    if !new_value.used_already_as_value?("#{object_property_object.
        local_name}")
      ...
      new_value.used_already_as_value("#{object_property_object.
          local_name}")
      ...
    else
      puts '#{new_value.local_name} has already been used and cannot be
          used again with an inverse functional property!'
    end
  end
end"
```

Deletion *setter* method source code extension:

```
"value.used_not_already_as_value(#{object_property_object.local_name})"
```

</td></tr>
<tr><td>Symmetric Property</td><td>

**Comment:** Object properties defined as *"symmetric"* cannot lead to inconsistency problems during the application of a functional ontology model produced by DEEP SEMANTICS.<br>
**Meta-Programming Code:** Therefore there are no specific code fragments generated considering symmetry of an object property.

</td></tr>
<tr><td>Transitive</td><td>

**Comment:** Like *"symmetric"* constraints the *"transitive"* restriction cannot lead to inconsistencies during the application of a functional ontology model produced by DEEP SEMANTICS.<br>
**Meta-Programming Code:** Therefore there are no specific code fragments generated considering transitivity of an object property.

</td></tr>
</table>

**Table 3.2: Types of local property constraints that have to be consider by DEEP SEMANTICS**

<table>
<tr><th>Type of Constraint</th><th>Comments and Corresponding RUBY Code generated by DEEP SEMANTICS</th></tr>
<tr><td>All-Values-From Restriction</td><td>

**Comment:** An *all-values-from* restriction constraints the range of allowed values for a property to a particular ontology class or datatype.

**Meta-Programming Code (shown only for object properties):**

```
"def #{object_property_object.local_name}=(new_value)
  if new_value.kind_of?(Thing)
    if (new_value.types.include?(#{all_values_from_class1}) || new_value.
        class.super_classes.include?(#{all_values_from_class1})) && ...
      ...
    else
      puts '#{new_value.local_name} cannot be used as value of #{
          object_property_object.local_name}!'
    end
  end
end"
```

</td></tr>
<tr><td>Some-Values-From Restriction</td><td>

**Comment:** Some-Values-From restrictions on properties state that at least one value of the considered property is an instance of a given ontology class (for object properties) or datatype (for datatype properties).

**Meta-Programming Code (shown only for object properties):**

```
"def #{object_property_object.local_name}=(new_value)
  if new_value.kind_of?(Thing)
    if ... || (new_value.types.include?(#{some_value_class1}) ||
        new_value.class.super_classes.include?(#{some_value_class1})) ||
        ...
      ...
    else
      puts '#{new_value.local_name} cannot be used as value of #{
          object_property_object.local_name}!'
    end
  end
end"
```

</td></tr>
<tr><td>Minimum Cardinality Restriction</td><td>

**Comment:** A minimum cardinality has no effect on DEEP SEMANTICS generated functional models.

**Meta-Programming Code:** No meta-programming is needed.

</td></tr>
<tr><td>Maximum Cardinality Restriction</td><td>

**Comment:** A property having a maximum cardinality restriction of 1 can have only one unique value **y** for each instance **x** that uses this property. If a property is functional and/or has maximum cardinality of 1 the helper *max* is also set to 1.

**Meta-Programming Code (for *max* equals 1):**

```
"def #{object_property_object.local_name}=(new_value)
  if new_value.kind_of?(Thing)
    if (@#{object_property_object.local_name}_values.size < #{max})
      @#{object_property_object.local_name}_values << new_value
    else
      @#{object_property_object.local_name}_values[0] = new_value
    end
  end
end"
```

</td></tr>
<tr><td>Cardinality Restriction</td><td>

**Comment:** A cardinality restriction statement implies that the minimum and maximum cardinality values are equal.

**Meta-Programming Code:** Please read "maximum cardinality" above.

</td></tr>
</table>

**Table 3.3: Runtime and main memory complexity comparison between DEEP SEMANTICS and OWL API**

| Test | DEEP SEMANTICS | OWL API |
|---|---|---|
| List classes by level:<br><br>**IKEN** ontology | Runtime list operation: 0.021 seconds<br>Runtime complete: 47.080 seconds<br>Peak main memory complexity (average): 77 (23) megabyte | Runtime list operation: 4.088 seconds<br>Runtime complete: 9.271 seconds<br>Peak main memory complexity: 28 megabyte |
| List classes by level:<br><br>**BIO2ME** ontology | Runtime list operation: 0.016 seconds<br>Runtime complete: 15.391 seconds<br>Peak main memory complexity (average): 63 (21) megabyte | Runtime list operation: 2.604 seconds<br>Runtime complete: 3.480 seconds<br>Peak main memory complexity: 24 megabyte |
| Find all instances by their<br><br>RDFS labels: **IKEN** ontology | Runtime find operation: 0.007 seconds<br>Runtime complete: 47.066 seconds<br>Peak main memory complexity (average): 77 megabyte (23) | Runtime find operation: 0.235 seconds<br>Runtime complete: 4.930 seconds<br>Peak main memory complexity: 28 megabyte |
| Find all instances by their<br><br>RDFS labels: **BIO2ME** ontology | Runtime find operation: 0.004 seconds<br>Runtime complete: 15.372 seconds<br>Peak main memory complexity (average): 63 megabyte (21) | Runtime find operation: 0.109 seconds<br>Runtime complete: 0.962 seconds<br>Peak main memory complexity: 23 megabyte |

as for example iterating named subclasses requires six lines of code (lines 23 to 28) while a functional analogous implementation with DEEP SEMANTICS would require only three lines.

**Listing 3.25:** XPERIMENTR implementation using JENA2: Print out the class hierarchy of the ontology.

```
1  // Print out all of the classes which are referenced in the ontology by hierarchy level
2  System.out.println("Known classes:");
3  for (ExtendedIterator i = ontology.listNamedClasses(); i.hasNext(); ) {
4          OntClass klass = (OntClass) i.next();
5          if ( klass.isHierarchyRoot() && (klass.listSuperClasses(true).toSet().size() > 0) ) {
6                  //System.out.println( klass.getURI() );
7                  xperimentr.printHierarchy(klass, 0);
8          }
9  }
```

**Listing 3.26:** XPERIMENTR implementation using JENA2: *printHierarchy()* methods.

```
1  /**
2   * Print the class hierarchy from this class down, assuming this class is at
3   * the given level. Makes no attempt to deal sensibly with multiple
4   * inheritance.
5   */
6  public void printHierarchy(OntClass klass, int level) {
7
8    System.out.println("Level: "+level);
9    System.out.println("  "+klass.getLocalName());
10
11   /* Find this classes instances*/
12   System.out.println("  This classes direct instances:");
13   for (ExtendedIterator i = klass.listInstances(true); i.hasNext(); ) {
14         Individual instance = (Individual) i.next();
15         System.out.println("    "+instance.getLocalName());
16   }
17
18   System.out.println();
19   System.out.println();
20   System.out.println("--------------------------");
21
22   /* Find the children and recurse */
23   for (ExtendedIterator i = klass.listSubClasses(true); i.hasNext(); ) {
24           OntClass subClass = (OntClass) i.next();
25           if ( subClass.isURIResource() && (subClass.listSuperClasses(true).toSet().size() >
               0) && (subClass.listSubClasses(true).toSet().size() > 0)  ) {
26                printHierarchy(subClass, level + 1);
27           }
28   }
29 }
```

The DEEP SEMANTICS source code of listing 3.18 on page 77 is comparable in its functionality to the JENA2 code of listing 3.27. While listing 3.18 comprises eight lines of code, the JENA2 version requires 13 lines. Comparing both listings reveals one of the most significant differing features between the deep integration approach of DEEP SEMANTICS and the JAVA-based frameworks: the programatic handling of ontology classes (for example line six in listing 3.18 in contrast to lines seven to nine in listing 3.27).

**Listing 3.27:** XPERIMENTR implementation using JENA2: source code for the retrieval of protocols that can be executed in a specified duration.

```
1  String time = console.readLine();
2  Integer timeInteger = Integer.valueOf( time ).intValue();
3
4  System.out.println();
```

```
5  System.out.println("Protocols that can be executed in "+time+" minutes:");
6
7  OntClass protocolKlass = ontology.getOntClass("http://www.ontoverse.org/ontologies/2008/11/
       xperimentr.owl#Protocol");
8  for (ExtendedIterator i = protocolKlass.listInstances(); i.hasNext(); ) {
9          Individual protocol = (Individual) i.next();
10         if ( protocol.getPropertyValue( ontology.getDatatypeProperty("http://www.ontoverse.org
               /ontologies/2008/11/xperimentr.owl#hasRequiredExecutionTime") ) != null ) {
11                 Literal executionTime = (Literal) protocol.getPropertyValue( ontology.
                       getDatatypeProperty("http://www.ontovers.org/recipes.owl#
                       hasRequiredExecutionTime"));
12                 Integer executionTimeInteger = Integer.valueOf( (String) executionTime.
                       getValue() ).intValue();
13                 if (executionTimeInteger <= timeInteger) {
14                         System.out.println( protocol.getLabel(null)+" can be prepared in "+
                               executionTimeInteger+" minutes.");
15                 }
16         }
17 }
```

#### Runtime and Memory Complexity Comparison

Runtime and memory complexity comparisons between DEEP SEMANTICS and JENA2 were carried out for both tests and both ontologies. Table 3.4 shows the corresponding results. As JENA2 is developed based on JAVA, the total runtimes are significantly, that means approximately 5 to 10 times, shorter for this framework. However, computing of the operations alone is around 100 to 600 times faster for DEEP SEMANTICS.

The highest main memory requirements for JENA2 are higher than for OWL API (around 43 megabytes in contrast to approximately 26 megabytes), but still less than the 77 megabytes (both **IKEN** ontology application tests) and 63 megabytes (both **BIO2ME** ontology application tests). In contrast the average main memory usage of DEEP SEMANTICS is less than OWL API's peak memory consumption.

### 3.8.3 DEEP SEMANTICS versus ACTIVERDF

In this subsection both compared frameworks (DEEP SEMANTICS and ACTIVERDF) are based on RUBY and its metaprogramming features. While ACTIVERDF focuses on RDF, DEEP SEMANTICS additionally supports OWL language constructs (currently in particular OWL LITE). Most important, ACTIVERDF is not *consistency safe* for RDFS and OWL.

#### Programming Complexity Comparison

The determination of the ontology's class and property hierarchies is not practicable in ACTIVERDF, because no subclass (sub-property) to superclass (super-property) relations are stored. Hence, an output of the class hierarchy by level could not be implemented.

Listing 3.17 (DEEP SEMANTICS code) on page 76 can be functionally and programatically compared to listing 3.28 (ACTIVERDF code). Measuring programming complexity using the number of code lines to implement equivalent functionality, 33 lines of functional code are

**Table 3.4: Runtime and main memory complexity comparison between DEEP SEMANTICS and JENA2**

| Test | DEEP SEMANTICS | JENA2 |
|---|---|---|
| List classes by level:<br><br>**IKEN** ontology | Runtime list operation: 0.021 seconds<br>Runtime complete: 47.080 seconds<br>Peak main memory complexity (average): 77 (23) megabyte | Runtime list operation: 2.661 seconds<br>Runtime complete: 4.296 seconds<br>Peak main memory complexity: 44 megabyte |
| List classes by level:<br><br>**BIO2ME** ontology | Runtime list operation: 0.016 seconds<br>Runtime complete: 15.391 seconds<br>Peak main memory complexity (average): 63 (21) megabyte | Runtime list operation: 1.876 seconds<br>Runtime complete: 3.754 seconds<br>Peak main memory complexity: 43 megabyte |
| Find all instances by their<br><br>RDFS labels: **IKEN** ontology | Runtime find operation: 0.007 seconds<br>Runtime complete: 47.066 seconds<br>Peak main memory complexity (average): 77 (23) megabyte | Runtime find operation: 2.425 seconds<br>Runtime complete: 4.176 seconds<br>Peak main memory complexity: 41 megabyte |
| Find all instances by their<br><br>RDFS labels: **BIO2ME** ontology | Runtime find operation: 0.004 seconds<br>Runtime complete: 15.372 seconds<br>Peak main memory complexity (average): 63 (21) megabyte | Runtime find operation: 2.639 seconds<br>Runtime complete: 4.519 seconds<br>Peak main memory complexity: 43 megabyte |

required using DEEP SEMANTICS in constrast to 58 lines when using ACTIVERDF. This difference is a consequence of method *find_instances_by_label* (line twelve of listing 3.17), which is provided by DEEP SEMANTICS. In ACTIVERDF a similar functionality has to be additionally implemented as shown in listing 3.28 (lines 13 to 29).

**Listing 3.28:** KITCHEN MENTOR implementation using ACTIVERDF: retrieving a recipe for a set of included materials.

```
1  materials = Array.new
2  found_protocols = XPERIMENTR::Protocol.find_all
3
4  puts "Please enter the next laboratory material or \"end\" to stop:"
5
6  while material_input = gets
7    case material_input
8    when "end\n"
9      break
10   else
11     material_input = material_input.chop
12
13     material_in_ontology = false
14
```

```
15      XPERIMENTR::LaboratoryMaterial.find_all.each do |material|
16        if material.label.class != Array
17          if material.label.eql?(material_input)
18            material_in_ontology = true
19            break
20          end
21        else
22          material.label.each do |label|
23            if label.eql?(material_input)
24              material_in_ontology = true
25              break
26            end
27          end
28        end
29      end
30
31      if material_in_ontology
32        materials << material_input
33        new_found_protocols = Array.new
34        found_protocols.each do |found_protocol|
35          if found_protocol.hasRequiredMaterial
36            found_protocol.hasRequiredMaterial.each do |required_material|
37              if required_material.label.class != Array
38                if required_material.label.eql?(material_input)
39                  new_found_protocols << found_protocol
40                end
41              else
42                required_material.label.each do |label|
43                  if label.eql?(material_input)
44                    new_found_protocols << found_protocol
45                    break
46                  end
47                end
48              end
49            end
50          end
51        end
52
53        found_protocols = new_found_protocols
54
55        puts "You have entered the following materials:"
56        materials.each do |material|
57          puts material
58        end
59
60        puts "The following protocols include these materials:"
61        found_protocols.each do |found_protocol|
62          puts found_protocol.label
63        end
64
65      else
66        puts "#{material_input} is not a known laboratory material."
67      end
68      puts "Please enter the next material you have or \"end\" to stop:"
69    end
70  end
```

Considerably more important than the missing of certain convenient functions is the fact that ACTIVERDF is not *consistency safe*. Listing 3.29 shows a short example of how the usage of ACTIVERDF can produce inconsistencies. This example is based on the XPERIMENTR ontology (see Section 6.1 in the appendix) and involves the protocol *SDS-Page* as well as this instance's properties *relatedExpert* (an object property) and *hasRequiredExecutionTime* (a datatype property). Of primary importance are lines seven and eight, respectively, in which the

corresponding property value assignments are stated stated. The OWL property range of *relatedExpert* is the class *Person*. Therefore, the assignment in line seven should not be allowed. Likewise, the range of *hasRequiredExecutionTime* is the data type *String*, which in turn causes line eight to produce an inconsistency.

However, executing listing 3.29 produces the following output:

> **>** sds_page.relatedExpert [<http://www.ontoverse.org/ontologies/2008/11/xperimentr.owl#Ilija>,
> <http://www.ontoverse.org/ontologies/2008/11/xperimentr.owl#Nahal>,
> ... <http://www.ontoverse.org/ontologies/2008/11/xperimentr.owl#Indra>
> **>** sds_page.hasRequiredExecutionTime 333
> **>**
> **>** sds_page.relatedExpert [<http://www.ontoverse.org/ontologies/2008/11/xperimentr.owl#Ilija>,
> <http://www.ontoverse.org/ontologies/2008/11/xperimentr.owl#Nahal>,
> ... <http://www.ontoverse.org/ontologies/2008/11/xperimentr.owl#Indra>,
> <http://www.ontoverse.org/ontologies/2008/11/xperimentr.owl#SDSPAGE>
> **>** sds_page.hasRequiredExecutionTime
> <http://www.ontoverse.org/ontologies/2008/11/xperimentr.owl#SDSPAGE>

As one can see, in ACTIVERDF the ontology instance

*<http://www.ontoverse.org/ontologies/2008/11/xperimentr.owl#SDSPAGE>*

is assigned to both, the object property *relatedExpert* and the dataype property *hasRequiredExecutionTime*.

Reasoning on this ACTIVERDF modified version of the ontology would now generate an error message as instance *SDSPAGE* is inconsistent. Using DEEP SEMANTICS such kinds of modifications are not allowed, if they would result in logical inconsistencies. Instead, DEEP SEMANTICS provides warnings that state that the intended operations cannot be conducted to avoid inconsistencies. The statement *sds_page.relatedExpert « sds_page* in line seven for example would produce the warning *"An instance of 'Protocol' cannot be added as value to object property 'relatedExpert' because: the global range of 'relatedExpert' is 'Person'!"*.

**Listing 3.29:** Consistency problems using *setter* in ACTIVERDF.

```
1  XPERIMENTR::Protocol.find_all.each do |protocol|
2    if protocol.localname.eql?("SDSPAGE")
3      sds_page = protocol
4      puts "sds_page.relatedExpert "+sds_page.relatedExpert
5          puts "sds_page.hasRequiredExecutionTime "+sds_page.hasRequiredExecutionTime
6
7      sds_page.relatedExpert << sds_page
8          sds_page.hasRequiredExecutionTime = sds_page
9
10     puts
11     puts "sds_page.relatedExpert "+sds_page.relatedExpert
12         puts "sds_page.hasRequiredExecutionTime "+sds_page.hasRequiredExecutionTime
13   end
14 end
```

**Table 3.5: Runtime and main memory complexity comparison between DEEP SEMANTICS and ACTIVERDF**

| Test | DEEP SEMANTICS | ACTIVERDF |
|---|---|---|
| Find all instances by their<br><br>RDFS labels: **IKEN** ontology | Runtime find operation: 0.007 seconds<br>Runtime complete: 47.066 seconds<br>Peak main memory complexity (average): 77 (23) megabyte | Runtime: 60.849 seconds<br><br>Runtime: 66.770 seconds<br><br>Peak main memory complexity: 103 megabyte |
| Find all instances by their<br><br>RDFS labels: **BIO2ME** ontology | Runtime find operation: 0.004 seconds<br>Runtime complete: 15.372 seconds<br>Peak main memory complexity (average): 63 (21) megabyte | Runtime find operation: 33.630 seconds<br>Runtime complete: 34.734 seconds<br>Peak main memory complexity: 31 megabyte |

#### Runtime and Main Memory Complexity Comparison

As mentioned above, the determination of the ontology's class hierarchy is not possible using ACTIVERDF. Therefore, test *"List classes by level"* could not be implemented for this framework. Table 3.5 shows results of the runtime and memory complexity comparison between DEEP SEMANTICS and ACTIVERDF for the remaining test.

Runtime differences for complete test runs are considerably smaller. DEEP SEMANTICS is approximately 0.5 to 2 times faster. However, computing of the operation alone is around 1000 to 8000 times faster for DEEP SEMANTICS.

The highest main memory usage is higher in ACTIVERDF for the **IKEN** ontology (around 77 megabytes for DEEP SEMANTICS ion contrast to approximately 103 megabytes) and lower for the smaller **BIO2ME** ontology (approximately 63 megabytes for DEEP SEMANTICS in contrast to 31 megabytes). The average main memory usage of DEEP SEMANTICS is in both cases less than ACTIVERDF's peek main memory requirement.

## 3.9 Discussion

The decision to implement DEEP SEMANTICS in the dynamic programming language RUBY has proved to be effective for the deep integration of OWL ontologies. All aspects and constructs, respectively, of an OWL LITE ontology have been successfully converted into deep integrated RUBY counterparts. The presented version of DEEP SEMANTICS has the following deep integration features:

1. Conversion of named OWL classes into RUBY classes.

2. Conversion of datatype and object properties into:

   (a) RUBY objects of type *DatatypeProperty* and *ObjectProperty*, respectively, and
   (b) *getter* and *setter* instance methods of the deep integrated ontology classes.

3. Consideration of global and local property restrictions:
   (a) The local cardinality restrictions, as well as the global restriction *functional*, are deep integrated into the corresponding *setter* instance method implementations. These *setter* check if the maximum amount of allowed values is already used for the referred property.
   (b) The globally acting domain constraint causes the corresponding property to be integrated only into those ontology classes that have been stated in the domain assertion.
   (c) The globally acting range constraint and the local *allValuesFrom* restriction are deep integrated into the corresponding *setter* instance method implementations as type checks. The type checks lead to the *setter*'s behavior that only those passed values are excepted that belong to the stated ontology classes or data types.
   (d) Locally defined *someValuesFrom* restrictions extend the set of allowed value types.
4. Conversion of ontology individuals into RUBY instances of the converted ontology classes.

These integration features of DEEP SEMANTICS, and in particular those related to global or local property restrictions, make the framework the first *consistency safe* one for OWL LITE ontologies. Additionally, DEEP SEMANTICS even prevents the generation of inconsistencies whose potential occurrence is hidden in ontology's structure. One example is the declaration of a local restriction concerning a property $\mathrm{P}$ in the context of a class $\mathrm{C_1}$. If now $\mathrm{C_1}$ is not part of the set of classes stated as domain for $\mathrm{P}$ and if $\mathrm{C_1}$ is disjoint with at least one domain class $\mathrm{C}$, than using $\mathrm{P}$ for an instance $\mathrm{i}$ of $\mathrm{C_1}$ would generate an inconsistency because of the following inference rules (?i, ?j are variables concerning OWL individuals; $\mathrm{C_1}$ and $\mathrm{C_2}$ are OWL classes; $\mathrm{P}$ represents and object property):

$$1.\ type(?i, C_1)\ \textbf{AND}\ P(?i, ?j)\textbf{AND}\ domain(P, C_2)$$

$$\Rightarrow type(?i, C_2)$$

$$2.\ type(?i, C_1)\ \textbf{AND}\ type(?i, C_2)\ \textbf{AND}\ disjointWith(C_1, C_2))$$

$$\Rightarrow \text{"}The\ ontology\ is\ inconsistent\ as\ indivual\ ?i\ cannot\ be\ of\ type\ C_1\ and\ its\ disjoint\ C_2.\text{"}$$

DEEP SEMANTICS detects such potential causes for inconsistencies during the deep integration process and makes sure that these inconsistencies cannot be generated using deep integrated ontology models. In stated example object property $\mathrm{P}$ would not be integrated as instance method into the source code of $\mathrm{C_1}$.

Another important topic to discuss is DEEP SEMANTICS current focus on OWL LITE and therefore lacking of OWL DL support. While OWL LITE is the least expressive sublanguage of OWL it is still expressive enough to capture – using syntactical tricks – all of OWL DL except of descriptions containing either individuals (for example *hasValue*) or cardinalities greater

than 1 (Bechhofer *et al.*, 2004; Horrocks *et al.*, 2003a). From this it follows that the exclusive support for OWL LITE is not critical for practical implementations using DEEP SEMANTICS. However, for the development of semantic applications that utilize existing OWL DL ontologies a conversion of these into OWL LITE is at least impractical and in worst cases not possible (for example if cardinalities higher than 1 are required).

To sum up, DEEP SEMANTICS's support for OWL LITE already makes it useful for a wide range of possible Semantic Web projects – even ones with higher semantic complexity requirements – while development of semantic applications based on existing OWL DL ontologies require an extension of DEEP SEMANTICS towards description logic support (this issue is further discussed in the conclusion in Subsection 3.9.4 below).

### 3.9.1 Programming Complexity

Section 3.8 contains comparisons between DEEP SEMANTICS and three other Semantic Web frameworks - OWL API, JENA2 and ACTIVERDF. Although there is no applicable formal definition of programming complexity as basis for these comparisons, the used example implementations show DEEP SEMANTICS to require significantly less lines of code for the same functionality than the other frameworks.

While RUBY is described as being an efficient way to achieve solutions with smaller amounts of code (Geer, 2006) than in most other programming languages, part of DEEP SEMANTICS's source code efficiency is due to its systems design emphasize on programmer, rather than computer, needs. Instead of solely focussing on runtime and main memory requirements, I designed DEEP SEMANTICS with a focus on usability concerning the requirements of semantic application developers. Related features of DEEP SEMANTICS are the provided convenient methods like *find_instances_by_label* (line 12 of listing 3.17 page 76) and *find_instance_by_local_name(local_name)*.

### 3.9.2 Runtime and Main Memory Complexity

While DEEP SEMANTICS is very efficient relating its source code complexity, the required initial deep integration process makes it the longest running framework compared in Section 3.8. In all mutual comparisons – for which the results are shown in Table 3.3, Table 3.4 and Table 3.5 – the completed runtime is always higher for DEEP SEMANTICS. Beginning from a twofold longer runtime compared to ACTIVERDF up to a considerable 600 times longer runtime when comparing with JENA2. Although these runtime comparisons suggest that DEEP SEMANTICS is slow in general, a closer look reveals that it was actually faster than all other frameworks when considering the tested operations only. In the extremest, test 2 – find all instances matching one of ten given search terms – runs 8000 times faster for DEEP SEMANTICS than for ACTIVERDF. The operational slowness of ACTIVERDF can easily be explained by its so-called lazy loading access of the underlying ontology data from the used triple store. Lazy loading means that the required ontology data is fetched from the triple store just in time when a search query on this triples is carried out. In contrast, explaining that DEEP SEMANTICS is faster than the JAVA

based frameworks JENA2 and OWL API is more difficult. A straightforward explanation is the computing overhead that results from the more abstract architecture of these frameworks. In other words: the computational cost of the initial deep integration results in a speed gain when using the integrated ontology model compared to processing the ontology data using an abstract model as in JENA2 and OWL API.

Considering DEEP SEMANTICS's memory complexity an interesting point is the difference between the average and peak memory usage. For the **IKEN** ontology the difference is 54 megabyte (77 megabyte peak and 23 megabyte average memory consumption) and for the **BIO2ME** ontology 42 megabyte (63 megabyte peak and 21 megabyte average memory usage). The peak memory demand is related to the deep integration process, while the resulting functional ontology model requires significantly less memory – about the amount of the average memory consumption.

### 3.9.3 Handling Multiple Ontologies in DEEP SEMANTICS

DEEP SEMANTICS has fundamental support for usage with multiple ontologies. However, to realize this support it requires appropriate reasoners during pre-processing. Principle problems of current ontology tools with respect to ontology import and referencing have been discussed in the literature (Liebig *et al.*, 2005). Although the vision of the Semantic Web heavily builds on sharing and re-using ontologies, both OWL provided corresponding functionalities – ontology import and referencing – are not perceived as being appropriate for this task.

While importing requires to include also all transitively imported ontologies into reasoning referencing of ontologies does not come with any semantic implications whatsoever. In consequence the import process can easily lead to computation and memory complexities during reasoning that render application of import functionality impractical while referencing does not change the ontology semantically (Grau *et al.*, 2004). From this it follows that the application of multiple ontologies is currently primarily dependent on the used reasoner.

However, future enhancement of OWL's multiple ontology support will certainly have to be considered in order to enable DEEP SEMANTICS to become a key technology of the Semantic Web. Furthermore, a straightforward possible pro-active solution could be the extension of the DEEP SEMANTICS framework towards an integrated network of DEEP SEMANTICS modules each covering exactly one ontology. Interconnections between these ontologies – for example mutual concept equivalence across ontology boundaries – could then be used to retrieve additional information stored in an other ontology and DEEP SEMANTICS module, respectively, whenever needed.

### 3.9.4 Conclusions and Outlook

DEEP SEMANTICS has proved to be efficient with respect to its programming complexity and operational runtime performance. Its architecture delivers an open approach that supports further extensions in future versions. One such an extension will be the implementation of a deep

integrating model builder supporting OWL DL in order to assist developers implementing applications that make use of description logic ontologies. Other useful extensions could be additional adapters like one to access ontologies stored in specialized triples stores like SESAME (Broekstra *et al.*, 2002) and BOCA (Feigenbaum *et al.*, 2007).

As described in Section 3.8 on page 81, is the current deep integration process computational complex. Starting points for improvements are:

- Optimization of the used triple parsing algorithm. For example by initially sorting the set of input triples into ABox and TBox related triples and not till then starting the parsing process.
- A straightforward approach would be breaking up the deep integration process from the actual application implementation. In particular the TBox – as long as it is not altered – can be pre-processed (that means previously deep integrated) and included into the application when needed.

The first recommendation of OWL is already more than four years old. At the moment of this writing initiatives for the definition of OWL 2 (Grau *et al.*, 2008) are underway. In order to support DEEP SEMANTICS's sustainability new model builder implementations for OWL 2 should be provided for future versions. Relating to matters of sustainability I currently intend to continue DEEP SEMANTICS's further development as an Open Source project.

# 4

# DEEP SEMANTICS in Action: IKEN and the BIO2ME

This chapter covers two reference applications based on DEEP SEMANTICS. The first one, **IKEN**, provides a novel type of semantics enabled web-interface developed during this doctoral thesis. Figure 4.1 shows a screenshot of two snippets of this web-interface. The second semantic application using DEEP SEMANTICS for ontology processing is the **BIO2ME** Information System developed by (Mainz, 2008). This application is described in section 4.2 on page 120.

## 4.1 IKEN

The project **IKEN**[1] is an applied research project lead-managed by VARION GmbH, an Germany based communication design agency. My area of responsibility was the design and implementation of the prototype application as well as the supporting ontology. A German description can be found on the project's website including a screencast showing the prototype of the **IKEN** system.

**IKEN** is at the moment of this writing a prototype platform for image retrieval enhanced by semantic annotations. **IKEN** is a proof-of-concept on how ontologies can support annotation and search in a photo collection. Recently, folksonomies have established themselves as popular means for indexing large document collections, with FLICKR[2] being the most popular example for social tagging in photo collections. Yet, user-generated annotations cause some problems regarding inconsistent vocabularies, varieties of synonyms, spelling variants, misspellings and language variants. In the long run, the user centric approach of social tagging will have to be combined with richer semantics of ontologies.

[1] http://www.i-ken.de

[2] http://www.flickr.com/

In summary the main aims of **IKEN** are:

- to provide a usable application which makes benefits of ontologies visible for internet users;
- to enable easily usable functionalities for ontology-based photo annotations;
- to allow browsing a document collection based on domain semantics.

**IKEN**'s novel type of semantics enabled web-interface has the following distinguishing features (numbers relate to the numbering used in Figure 4.1):

1. Terms and their related ontology instances are visualized in an innovative and user-friendly style: ontology instances are represented as blue rectangles and relations between two instances as larger and more grayish blue rectangles. The small green rectangles at the right side of an ontology instance named *"Verfeinern" (Refine)* are buttons that opens a refinement interface for the respective instance – the refinement interaction is described in the next bullet point. Below the small green button there is also an orange one. This button named *"Details"* invokes a pop-up window, showing details about the referenced instance utilizing the related knowledge from the ontology.

2. **IKEN** semantic interface offers a first of its kind context sensitive annotation and search refinement interface based on the semantics modeled in the **IKEN** ontology. If the user for example clicks on *"Verfeinern" (Refine)* for the search term *"Frau" (Woman)* he will see the interface shown in Figure 4.1. Based on the semantics about *"Frau" (Woman)* modeled in the **IKEN** ontology, the application offers a list of refinement choices. The user can then choose for example the hair color or the age of the woman. Beside offering helpful support during search, this interface also optimizes annotation tasks as it helps the annotator to identify relevant term refinements.

3. The *"Details"* interface displays information about the search term *"T-Shirt"*. **IKEN** uses the modeled semantic context to generate a view containing term specific information. For the term *"T-Shirt"* these are information about synonyms and *is-a* relations (*"T-Shirt is an outerwear. T-Shirt is an apparel."*).

The **IKEN** system is designed to enable users, who are not experts in knowledge engineering and ontology modeling, to apply semantic annotations and to use semantic information retrieval in image collections. As a first use case it focuses on applications in the tourism sector. This mainly effects the choice of the images included in the database and the ontology in use; the underlying principles can be applied to other and less-specified domains as well.

### 4.1.1 The IKEN ontology

The **IKEN** ontology is modeled in OWL LITE. It was developed from scratch to exactly fit into the new system. The ontology comprises three main types of concepts: a) geographical concepts

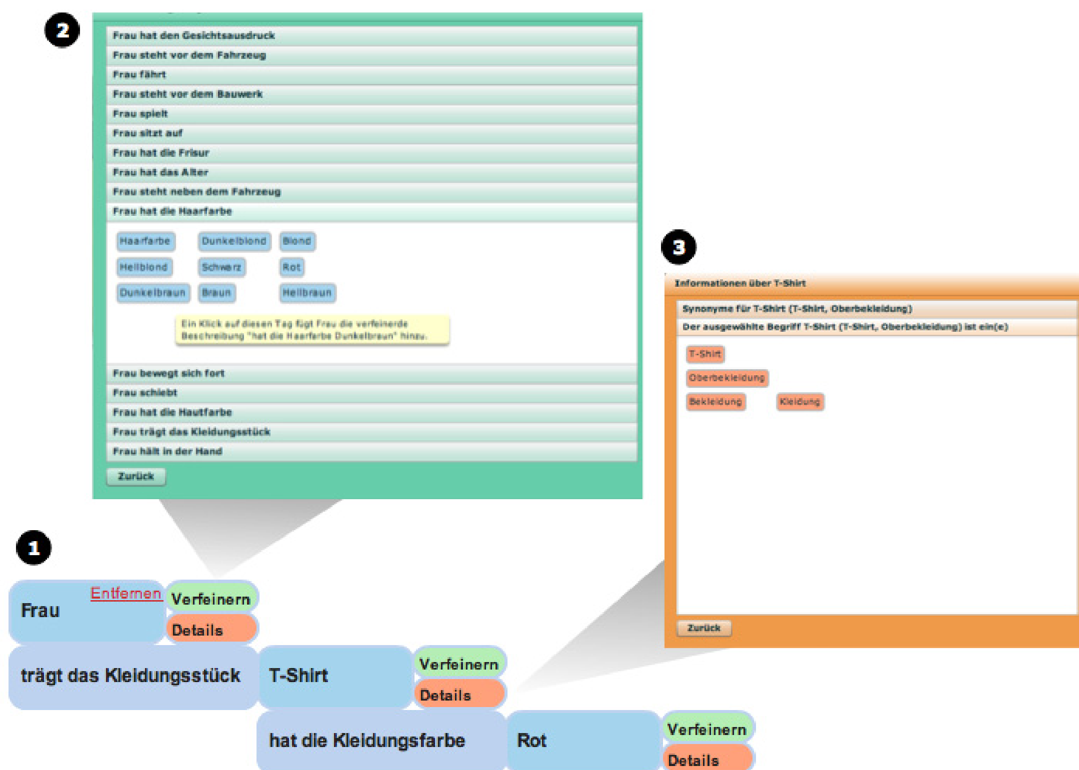


**Figure 4.1: Screenshot of the novel semantics enabled web-interface of the IKEN application.** As IKEN is completely in German, interface snippets in this screenshot are in German, also. **1)** The smaller rectangles highlighted in blue represent ontology instances and the bigger ones represent relations between two instances. **2)**

---

like *"Country"*, *"City"* and *"District"* as well as *"Street"*, *"Building"* and *"Park"*; b) concepts that directly relate to objects visible in the pictures like *"Building"*, *"Vehicle"*, *"Clothing"* or *"Animal"*; c) others are used to capture the sets of individuals that can be used as values of property axioms that themselves define attributes of the visual objects – examples are: *"Colour"* and *"FacialExpression"*.

The annotations of a photo are stored using instances of the concept *"Photo"* (see part 2) of Figure 4.2 for an example). For each photo one instance of *"Photo"* is created. Annotations belonging to a photo are asserted using the object property *"hatAnnotation" (has annotation)* with its sub properties *"hatGeographischerRaumAnnotation" (has geographical area annotation)*, *"hatEmotionAnnotation" (hasEmotionAnnotation)* and *"hatTourismuskategorieAnnotation" (has tourism category annotation)* which all have *"Photo"* as their domain. The property *"hatGeographischerRaumAnnotation" (has geographical area annotation)* is functional and has the concept *"GeographischerRaum" (Geographic Area)* as its range. It is used to unambiguously identify where a certain photo has been taken. With *"hatEmotionAnnotation" (hasEmotionAnnotation)* general emotional features of a photo – like *"Friede" (Peace)* or *"Einsamkeit" (Loneliness)* - can be added to the description. Likewise *"hatTourismuskategorieAnnotation"*

*(has tourism category annotation)* is used to assert the photo to certain related categories like *"Natur" (Nature)*, *"Sports"* or *"Erholung" (Recreation).*

Metrics of the **IKEN** ontology (25nd of July): 360 concepts/classes, 70 object properties, 6 datatype properties, 222 initial instances and additionally 8 SWRL rules.

### The Class Hierarchy

This is a list of the top-level classes of the **IKEN** ontology:

- **Alter (Age)**: This concept is used to enable the annotation of the age of people visible in a photo.
- **Bauteil (Structural Element)**: Subclasses of *"Bauteil" (Structural Element)* are for example *"Wand" (Wall)*, *"Tuer" (Door)* and *"Treppe" (Staircase)*. This class is used for the specification of buildings.
- **Bauwerk (Building)**: This concept is used to support the annotation of a *"Bauwerk" (Building)* that can be seen in a photo. Subclasses are for example *"Burg" (Castle)*, *"Turm" (Tower)* and *"Wohnhaus" (Apartment Building).*
- **Bewoelkung (Cloudiness)**: *"Bewoelkung" (Cloudiness)* is used to enable the annotation of the cloudiness of the visible sky in a photo.
- **Emotion**: Subclasses of *"Emotion"* are *"NegativeEmotion"*, *"PositiveEmotion"* and *"NeutraleEmotion" (Neutral Emotion)*. This concept is used to model the subjective impression a photo has on the person that annotates it.
- **Fahrzeug (Vehicle)**: The concept *"Fahrzeug" (Vehicle)* is the root class for all kinds of vehicles that can occur in photos related to the tourism domain of **IKEN** like for example *"Auto" (Car)*, *"Bus"*, *"Strassenbahn" (Tram)* and *"Fahrrad" (Bicycle).*
- **Farbe (Colour)**: The declaration of colours of objects that can be seen in photos is very important. Therefore the class *"Farbe" (Colour)* is used as the root class for more specialized concepts like *"Kleidungsfarbe" (Colour of Clothing)*, *"Hautfarbe" (Skin Colour)*, and *"Haarfarbe" (Hair Colour).*
- **Fortbewegungsart (Type of Movement)**: This concept currently comprises four instances: *"gehen" (walking)*, *"joggend" (walking)*, *"laufend" (running)* and *"rennend" (sprinting)*. It is used as range for the property *"bewegtSichFort" (moves).*
- **Frisur (Hairstyle)**: This concept currently comprises seven instances: *"Glatze" (Bald Head)* , *"Kurzhaarschnitt" (Short Haircut)* , *"Langhaarschnitt" (Long Haircut)*, *"Locken" (Curls)*, *"Pferdeschwanz" (Ponytail)*, *"Pony" (Fringe)*, and *"Zopf" (Pigtail)*. It is used as range for the property *"hatFrisur" (has hairstyle).*

- **Gegenstand (Artefact)**: The concept *"Gegenstand" (Artefact)* comprises all kinds of artificial objects, potentially visible in photo that are not part of either *"Bauteil" (Structural Element)*, *"Bauwerk" (Building)*, *"Fahrzeug" (Vehicle)*, *"Kleidung" (Clothing)* and/or *"Moebel" (Piece of Furniture)*. In particular, it is the superclass of *"InDerHandHaltbarerGegenstand" (In the Hand holdable Artefact)* and *"SchiebbarerGegenstand" (Pushable Artefact)* which are important classes for the detailed description of people (e.g. *A man that holds an umbrella.*). Examples for *"InDerHandHaltbarerGegenstand" (In the Hand holdable Artefact)* are: *"Buch" (Book)*, *"Regenschirm" (Umbrella)* and *"Glas"*. Examples for *"SchiebbarerGegenstand" (Pushable Artefact)* are: *"Kinderwagen" (Perambulator)*, *"Einkaufswagen" (Shopping Trolley)* and *"Gepaeckwagen" (Baggage Cart)*.

- **GeographischerRaum (Geographic Area)**: This concept is used as a root for all geographical annotations that can be used to identify the location where a photo has been taken, excluding buildings that are modeled separately. Subclasses of *"GeographischerRaum" (Geographic Area)* are for example *"Stadt" (Town)*, *"Stadtteil" (District)* and *"Strasse" (Street)*. For the annotation process only initial instances of these classes are allowed. The initial instances for *"Stadt" (Town)* in the current proof-of-concept ontology version are for example: *Duesseldorf* and *Krefeld* (other city names have to be added in order to be applicable for annotation). Terms like an unspecified *"Gruenflaeche" (Green Area)* are not included in the definition of *"GeographischerRaum" (Geographic Area))*.

- **Gesichtsausdruck (Facial Expression)**: This concept currently comprises six instances: *"Aengstlich" (Anxious)* , *"Grinsen" (Grining)* , *"Lachen" (Laughing)*, *"Laecheln" (Smiling)*, *"Weinen" (Crying)* and *"Wuetend" (Angry)*. It is used as range for the property *"hatGesichtsausdruck" (has facial expression)*.

- **Gruenflaeche (Green Area)**: This concept is used a root for all green area annotations that can not be used to identify the location where a photo has been taken – the subclasses are: *"Rasen" (Lawn)*, *"Weide" (Grazing Land)* and *"Wiese" (Grassland)*.

- **Himmel (Sky)**: This concept is used to enable the annotation of the sky visible in a photo. Annotation instances of this concept can be used as subject for *"hatBewoelkung" (has cloudiness)* statements.

- **Kleidung (Clothing)**: Subclasses of *"Kleidung" (Clothing)* are for example *"Hose" (Trousers)*, *"Mantel" (Coat)* and *"Herrenanzug" (Gentleman Suit)*. This class is used for the annotation of the clothing of people visible in a photo.

- **Lebewesen (Creature)**: The concept *"Lebewesen" (Creature)* comprises all kinds of creatures, potentially visible in photos like for example: *"Mensch" (Human)*, *"Pflanze" (Plant)* and the more specific *"Hund" (Dog)*.

- **Moebel (Piece of Furniture)**: *"Moebel" (Piece of Furniture)* is used to enable the annotation of furniture visible in a photo.

- **Monat (Month)**: Is used to define the month when an event takes place – *"Monat" (Month)* is the range of *"findetStattImMonat" (takes place in month)*.

- **Photo**: Instances of the concept *"Photo"* are generated whenever a photo is annotated. Annotation statements are then related to these instances using for example the properties *"hatAnnotation" (has annotation)*, *"hatTag" (has tag)* and *"createdAt"*.

- **Sehenswuerdigkeit (Tourist Feature)**: This class is used for the specification of points of interest in a geographical area. Subclasses of *"Sehenswuerdigkeit" (Tourist Feature)* are for example *"Baudenkmal" (Monument)*, *"BotanischerGarten" (Botanical Garden)* and *"Museum"*. For the annotations process only initial instances of these classes are allowed. The initial instances for *"Baudenkmal" (Monument)* in the current proof-of-concept ontology version are for example: *"KaiserWilhelmStandbild"* (a statue), *"Reiterstandbild-JanWellems"* (an equestrian sculpture) and *"SeidenweberDenkmal"* (a statue).

- **Sitzgelegenheit (Seating)**: *"Sitzgelegenheit" (Seating)* is used to model all kinds of instances that can be used as values for the *"sitztAuf" (sits on)* relation (for example *"Woman sits on a picnic blanket."*).

- **Spiel (Game)**: *"Spiel" (Game)* covers all kinds of games that can be played by humans like: *"Ballspiel" (Ball Game)*, *"Brettspiel" (Board Game)* and *"Kartenspiel" (Card Game)*.

- **Tourismuskategorie (Tourism Category)**: This concept currently comprises six instances: *"Erholung" (Recreation)*, *"Freizeit" (Leisure Time)*, *"Gastronomie" (Gastronomy)*, *"Kulturtourismus" (Cultural Tourism)*, *"Natur" (Nature)* and *"Sport"*. It is used to assort the annotated photos into certain categories that are related with the tourism domain.

- **Unternehmen (Company)**: This is the superclass for all kinds of companies and business related terms that can occur as part of the semantic annotations. For the proof-of-concept data set there have only two instances of *"Unternehmen" (Company)* been used: the restaurant *"StadtwaldhausKrefeld"* and its beer garden *"BiergartenStadtwaldhausKrefeld"*.

- **Veranstaltung (Event)**: This class is used for the specification of special events in a geographical area. Subclasses of *"Veranstaltung" (Event)* are for example *"Kirmes" (Fun Fair)*, *"Messe" (Trade Fair)* and *"Weihnachtsmarkt" (Christmas Market)*. For the annotations process only initial instances of these classes are allowed. The initial instances for *"Weihnachtsmarkt" (Christmas Market))* in the current proof-of-concept ontology version are for example: *"WeihnachtsmarktDuesseldorf"* (the christmas market in Düsseldorf city) and *"WeihnachtsmarktKrefeld"* (the christmas market in Krefeld city).

**The Annotation Properties**

During the implementation process of the **IKEN** prototype new requirements were encountered concerning the annotation capabilities of the **IKEN** ontology. This lead to the definition of four new custom annotation properties:

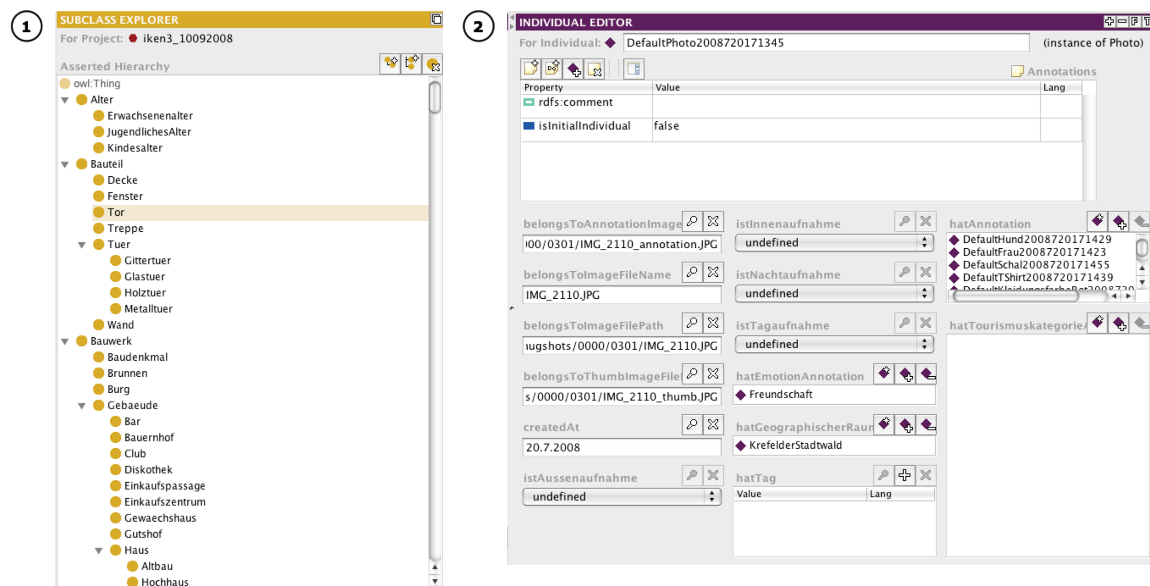


**Figure 4.2: Extract of the IKEN class hierarchy and a sample photo annotation.** 1) A screenshot of a part of the **IKEN** class hierarchy in PROTÉGÉ. 2) Screenshot of the PROTÉGÉ individuals view of a photo instance. It becomes apparent here that the corresponding photo is for example annotated with the instances *"DefaultHund2008720171429"* (a dog) and *"DefaultFrau2008720171423"* (a woman).

---

- **primaryLabel**: The property *"primaryLabel"* is an extension for the built-in *rdfs:label* annotation property. It is used to store the label of a concept, instance or property that should be used by default for graphical user interface.

- **isInitialIndividual**: This annotation property is used on individuals only with value **TRUE** to indicate that the individual has been in the initial ontology model as defined by the ontology designer and can be used for the annotation of multiple photos.

- **isUsableForSpecificationGUI**: *"isUsableForSpecificationGUI"* is applied for the annotation of object and data type properties. The value of this annotation property can either be **TRUE** or **FALSE**. If the value is **TRUE** then the annotated property can be utilised to further specify an already added annotation term. In Figure 4.6 for example one can see only those specification options in the refine view for which the corresponding properties have the *"isUsableForSpecificationGUI"* value **TRUE**.

- **hasSearchSpecificationMode**: If *"isUsableForSpecificationGUI"* has the value **TRUE** in the annotation of a given object or data type property then possible type or mode of the specification has to be indicated with *"hasSearchSpecificationMode"*. Allowed values of *"hasSearchSpecificationMode"* are *"Update"* and *"Substitution"*. If the specification mode is *"Update"* then the specification process leads to an additional statement about the specified term - for example: 1. An user searches for *"a man with blond hair."* 2. She clicks on *"Verfeinern" (Refine)* for the search term *"man"*. 3. She gets a list of properties that are usable for this specification interface and clicks on *"Man drives car"*. 4. The search term *"man"* has now been updated and the new query is *"a man with blond hair and that drives a car."*. If in contrast the specification mode is *"Substitution"* then the

specification process leads to an exchange of the old term with a new one - for example: 1. An user searches for *"a man with blond hair"*. 2. She clicks on *"Verfeinern" (Refine)* for the search term *"man"*. 3. She gets a list of properties that are usable for this specification interface and clicks on *"woman"* in the context of *Other terms that are sub-categories of the concept "Human"* 4. The search term *"man"* has now been substituted by *"woman"* and the new query is *a woman with blond hair."*.

### The Object Property Hierarchy

The object properties in the **IKEN** ontology are predominantly used for the specification of content objects like *"traegtKleidung" (wears clothing)* which can be used in the system to annotate the type of clothing of a person in the photo. Besides object properties are for example applied to relate photo instances to annotation terms with *"hatAnnotation" (has annotation)* and to describe part-of relations between geographical areas with *"liegtIn" (is situated in)*. The following list of top-level **IKEN** object properties gives an overview about their purpose and formal realization.

- **hatRegelmaessigeVeranstaltung (has regular event)**: This property connects locations with events (for example *"Düsseldorf has regular event Japanese Day."*). Its inverse property is *"findetStattIn" (takes place in)*.
  - **Domain**: *"Ort" (Location)*
  - **Range**: *"Veranstaltung" (Event)*
  - **isUsableForSpecificationGUI**: True
  - **hasSearchSpecificationMode**: Substitution
- **liegtIn (is situated in)**: This property connects smaller geographical areas with larger ones (for example *"Düsseldorf is situated in Germany."*). Its inverse property is *"umfasst" (comprises)*.
  - **Domain**: *"GeographischerRaum" (Geographic Area)*
  - **Range**: *"GeographischerRaum" (Geographic Area)*
  - **isUsableForSpecificationGUI**: False
  - **hasSearchSpecificationMode**: None
- **umfasst (comprises)**: This property connects larger geographical areas with smaller ones (for example *"Germany comprises Düsseldorf."*). Its inverse property is *"liegtIn" (is situated in)*.
  - **Domain**: *"GeographischerRaum" (Geographic Area)*
  - **Range**: *"GeographischerRaum" (Geographic Area)*
  - **isUsableForSpecificationGUI**: True
  - **hasSearchSpecificationMode**: Substitution

- **istFarbeVon (is colour of)**: This property connects a certain colour with some instance that is of this colour (for example *"Yellow is the colour of a banana."*).
  - **Domain**:*"Farbe" (Colour)*
  - **Range**: OWL Thing
  - **isUsableForSpecificationGUI**: False
  - **hasSearchSpecificationMode**: None
- **traegtKleidung (wears clothing)**: *"traegtKleidung" (wears clothing)* connects a person to the clothing he is wearing in the photo (for example *"The man wears blue trousers."*).
  - **Domain**: *"Mensch" (Human)*
  - **Range**: *"Kleidung" (Clothing)*
  - **isUsableForSpecificationGUI**: True
  - **hasSearchSpecificationMode**: Update
- **hatFarbe (has colour)**: This property connects some instance with a certain colour this instance has (for example *"The banana is yellow."*).
  - **Domain**: OWL Thing
  - **Range**: *"Farbe" (Colour)*
  - **isUsableForSpecificationGUI**: False
  - **hasSearchSpecificationMode**: None
- **findetStattImMonat (takes place in month)**: *"findetStattImMonat" (takes place in month)* connects an event with month this event takes place.
  - **Domain**: *"Veranstaltung" (Event)*
  - **Range**: *"Monat" (Month)*
  - **isUsableForSpecificationGUI**: False
  - **hasSearchSpecificationMode**: None
- **wirdGefahrenVon (is driven by)**: This property connects a vehicle with its driver (for example *"The black car is driven by a woman with wearing a blue T-shirt."*). Its inverse property is *"faehrt" (drives)*.
  - **Domain**: *"Fahrzeug" (Vehicle)*
  - **Range**: *"Mensch" (Human)*
  - **isUsableForSpecificationGUI**: True
  - **hasSearchSpecificationMode**: Update
- **faehrt (drives)**: *"faehrt" (drives)* connects a driver with the vehicle she drives (for example *"The woman wearing a blue T-shirt drives a motorbike."*). Its inverse property is *"wirdGefahrenVon" (is driven by)*.

  - **Domain**: *"Mensch" (Human)*
  - **Range**: *"Fahrzeug" (Vehicle)*
  - **isUsableForSpecificationGUI**: True
  - **hasSearchSpecificationMode**: Update

- **hatAnnotation (has annotation)**: Is the most important property as it connects the photo instances with the annotations describing the visible content. *"hatAnnotation" (has annotation)* comprises three sub-properties *"hatTourismuskategorieAnnotation" (has tourism category annotation)*, *"hatGeographischerRaumAnnotation" (has geographical area annotation)* and *"hatEmotionAnnotation" (hasEmotionAnnotation).*
  - **Domain**:*"Photo"*
  - **Range**: OWL Thing
  - **isUsableForSpecificationGUI**: False
  - **hasSearchSpecificationMode**: None

- **stehtNeben (stands beside of)**: A person can be specified as standing beside of a visible object in the photo (for example *"The woman stands beside of a motorbike."*).
  - **Domain**: *"Mensch" (Human)*
  - **Range**: OWL Thing
  - **isUsableForSpecificationGUI**: False
  - **hasSearchSpecificationMode**: None

- **haeltInderHand (holds in hand)**: *"haeltInderHand" (holds in hand)* connects a person with an in the hand holdable artefact (for example *"The woman holds in hand a glas."*).
  - **Domain**: *"Mensch" (Human)*
  - **Range**: *"InDerHandHaltbarerGegenstand" (In the Hand holdable Artefact)*
  - **isUsableForSpecificationGUI**: True
  - **hasSearchSpecificationMode**: Update

- **hatFrisur (has hairstyle)**: A person can be specified as having a certain hairstyle (for example *"The man has the hairstyle curls."*).
  - **Domain**: *"Mensch" (Human)*
  - **Range**: *"Frisur" (Hairstyle)*
  - **isUsableForSpecificationGUI**: True
  - **hasSearchSpecificationMode**: Update

- **spielt (plays)**: A person can be specified as playing a certain game (for example *"The boy plays soccer."*).

- **Domain**: *"Mensch" (Human)*
- **Range**: *"Spiel" (Game)*
- **isUsableForSpecificationGUI**: True
- **hasSearchSpecificationMode**: Update

- **findetStattIn (takes place in)**: This property connects events with locations (for example *The Japanese Day takes place in "Düsseldorf."*). Its inverse property is *"hatRegelmaessigeVeranstaltung (has regular event).*
  - **Domain**: *"Veranstaltung" (Event)*
  - **Range**: *"Ort" (Location)*
  - **isUsableForSpecificationGUI**: True
  - **hasSearchSpecificationMode**: Substitution

- **sitztAuf (sits on)**: *"sitztAuf" (sits on)* connects a person with an object this person is sitting on (for example *"The woman sits on a couch."*).
  - **Domain**: *"Mensch" (Human)*
  - **Range**: *"Sitzgelegenheit" (Seating)*
  - **isUsableForSpecificationGUI**: True
  - **hasSearchSpecificationMode**: Update

- **stehtVor (stands in front of)**: *"stehtVor" (stands in front of)* connects a person with an object this person is standing in front of (for example *"The man stands in front of a house."*).
  - **Domain**: *"Mensch" (Human)*
  - **Range**: OWL Thing
  - **isUsableForSpecificationGUI**: False
  - **hasSearchSpecificationMode**: None

- **schiebt (pushes)**: A person can be specified as pushing an object (for example *"The girl pushes a perambulator."*).
  - **Domain**: *"Mensch" (Human)*
  - **Range**: *"SchiebbarerGegenstand" (Pushable Artefact)*
  - **isUsableForSpecificationGUI**: True
  - **hasSearchSpecificationMode**: Update

- **hatBewoelkung (has cloudiness)**: The cloudiness of the sky can be described with *"hatBewoelkung" (has cloudiness).*
  - **Domain**: *"Himmel" (Sky)*

- **Range**: *"Bewoelkung" (Cloudiness)*
- **isUsableForSpecificationGUI**: True
- **hasSearchSpecificationMode**: Update

- **bewegtSichFort (moves)**: *"bewegtSichFort" (moves)* connects a person to her type of movement (for example *"The woman is running."*).
  - **Domain**: *"Mensch" (Human)*
  - **Range**: *"Fortbewegungsart" (Type of Movement)*
  - **isUsableForSpecificationGUI**: True
  - **hasSearchSpecificationMode**: Update

- **hatAlter (has age)**: A person can be annotated as having a certain age (for example *"The man has the age 42."*).
  - **Domain**: *"Mensch" (Human)*
  - **Range**: *"Alter" (Age)*
  - **isUsableForSpecificationGUI**: False
  - **hasSearchSpecificationMode**: None

- **hatGesichtsausdruck (has facial expression)**: *"hatGesichtsausdruck" (has facial expression)* connects a person with her facial expression (for example *"The woman has facial expression smiling."*).
  - **Domain**: *"Mensch" (Human)*
  - **Range**: *"Gesichtsausdruck" (Facial Expression)*
  - **isUsableForSpecificationGUI**: True
  - **hasSearchSpecificationMode**: Update

#### The Data Type Property Hierarchy

Data type properties are applied in the **IKEN** ontology when the possible values for these properties do not have to be further specified. *hatTag (has tag)* for example relates photo instances to non semantic tags. According their nature nNon semantic tags are not modeled in the ontology and are therefore connected via a data type property. Top-level **IKEN** data type properties are:

- **hatTag (has tag)**: *hatTag (has tag)* is applied to all annotations terms, that can not be mapped to the ontology. An example would be *"the photo has the tag Barack Obama"* as the **IKEN** ontology does not comprise the term *"Barack Obama"*.
  - **Domain**:*"Photo"*
  - **Range**: String

  - **isUsableForSpecificationGUI**: False
  - **hasSearchSpecificationMode**: None

- **istNachtaufnahme (is night photograph)**: This data type property prepares the possibility to annotate if a photo has been taken at night. However, this property is not supported by the prototype at the moment.
  - **Domain**:*"Photo"*
  - **Range**: Boolean
  - **isUsableForSpecificationGUI**: False
  - **hasSearchSpecificationMode**: None

- **createdAt**: *"createdAt"* is applied to all photo instances and saves the exact time when the related photo has been uploaded into the system.
  - **Domain**:*"Photo"*
  - **Range**: String
  - **isUsableForSpecificationGUI**: False
  - **hasSearchSpecificationMode**: None

- **belongsToImageFileName**: Like *"createdAt" "belongsToImageFileName"* is applied to all photo instances as well. Using this property the name of the photo file is related to its photo annotation instance.
  - **Domain**:*"Photo"*
  - **Range**: String
  - **isUsableForSpecificationGUI**: False
  - **hasSearchSpecificationMode**: None

- **istAussenaufnahme (is location shot)**: The data type property *"istAussenaufnahme" (is location shot)* prepares the possibility to annotate if a photo has been taken outside. It is not supported by the **IKEN** prototype at the moment.
  - **Domain**:*"Photo"*
  - **Range**: Boolean
  - **isUsableForSpecificationGUI**: False
  - **hasSearchSpecificationMode**: None

- **belongsToThumbImageFilePath**: This data type property links every photo annotation instance to the file path of the thumb image of the corresponding photo.
  - **Domain**:*"Photo"*
  - **Range**: String

  - **isUsableForSpecificationGUI**: False
  - **hasSearchSpecificationMode**: None

- **istTagaufnahme (is day photograph))**: Like *"istNachtaufnahme" (is night photograph)* this data type property is not supported by the prototype at the moment. It will be used in future versions to annotate if a photo has been taken by day.
  - **Domain**:*"Photo"*
  - **Range**: Boolean
  - **isUsableForSpecificationGUI**: False
  - **hasSearchSpecificationMode**: None

- **belongsToImageFilePath**: *"belongsToImageFilePath"* links every photo annotation instance to the path of the corresponding photo file.
  - **Domain**:*"Photo"*
  - **Range**: String
  - **isUsableForSpecificationGUI**: False
  - **hasSearchSpecificationMode**: None

- **belongsToAnnotationImageFilePath**: The annotation image file is a copy of the original photo with reduced resolution which is used in the graphical user interface. *"belongsToAnnotationImageFilePath"* links every photo annotation instance to the path of this annotation image file.
  - **Domain**:*"Photo"*
  - **Range**: String
  - **isUsableForSpecificationGUI**: False
  - **hasSearchSpecificationMode**: None

#### The used SWRL Rules

A series of annotation assertions are automatically added to the photos using PELLET with the following SWRL rules:

- **Rule 1 – Colour annotation**

$$Photo(?x) \;\wedge\; hatAnnotation(?x, ?y) \;\wedge\; hatFarbe(?y, ?z)$$

$$\Rightarrow hatAnnotation(?x, ?z)$$

  Explanation: If an instance **x** of *"Photo"* is annotated with an individual **y** which has a *has colour ("hatFarbe")* attribute with colour **z** as value then **x** is also annotated with this colour **z**.

- **Rule 2 – Is situated in annotation 1**

$$Photo(?x) \ \wedge \ hatGeographischerRaumAnnotation(?x,?y) \ \wedge \ liegtIn(?y,?z)$$

$$\Rightarrow hatAnnotation(?x,?z)$$

Explanation: If an instance **x** of *"Photo"* is annotated with an instance **y** of *Geographic Area ("GeographischerRaum")* and if **y** *is situated in ("liegtIn")* **z** then **x** is also annotated with **z**.

- **Rule 3 – Is situated in annotation 2**

$$Photo(?x) \ \wedge \ hatAnnotation(?x,?y) \ \wedge \ liegtIn(?y,?z)$$

$$\Rightarrow hatAnnotation(?x,?z)$$

Explanation: If an instance **x** of *"Photo"* is annotated with an individual **y** and if **y** *is situated in ("liegtIn")* **z** then **x** is also annotated with **z**.

- **Rule 4 – Tourism category annotation 1**

$$Photo(?x) \ \wedge \ hatAnnotation(?x,?y) \ \wedge \ Sehenswuerdigkeit(?y)$$

$$\Rightarrow hatTourismuskategorieAnnotation(?x,Freizeit)$$

Explanation: If an instance **x** of *"Photo"* is annotated with an instance **y** of *Tourist Feature ("Sehenswuerdigkeit")* then **x** is annotated (*"hatTourismuskategorieAnnotation"*) with the tourism category *Leisure ("Freizeit")*.

- **Rule 5 – Tourism category annotation 2**

$$Photo(?x) \ \wedge \ hatAnnotation(?x,?y) \ \wedge \ Gewaesser(?y)$$

$$\Rightarrow hatTourismuskategorieAnnotation(?x,Natur)$$

Explanation: If an instance **x** of *"Photo"* is annotated with an instance **y** of *Body of Water ("Gewaesser")* then **x** is annotated (*"hatTourismuskategorieAnnotation"*) with the tourism category *Nature ("Natur")*.

- **Rule 6 – Tourism category annotation 3**

$$Photo(?x) \ \wedge \ hatAnnotation(?x,?y) \ \wedge \ Gruenanlage(?y)$$

$$\Rightarrow hatTourismuskategorieAnnotation(?x,Natur)$$

Explanation: If an instance **x** of *"Photo"* is annotated with an instance **y** of *Recreation Area ("Gruenanlage")* then **x** is annotated (*"hatTourismuskategorieAnnotation"*) with the tourism category *Nature ("Natur")*.

- **Rule 7 – Tourism category annotation 4**

$$Photo(?x) \ \wedge \ hatAnnotation(?x, ?y) \ \wedge \ Pflanze(?y)$$

$$\Rightarrow hatTourismuskategorieAnnotation(?x, Natur)$$

Explanation: If an instance **x** of *"Photo"* is annotated with an instance **y** of *Plant ("Pflanze")* then **x** is annotated (*"hatTourismuskategorieAnnotation"*) with the tourism category *Nature ("Natur")*.

- **Rule 8 – Event annotation**

$$Photo(?x) \wedge Veranstaltung(?y) \wedge findetStattImMonat(?y, ?z) \wedge hatAnnotation(?x, ?y)$$

$$\Rightarrow hatAnnotation(?x, ?z)$$

Explanation: If an instance **x** of *"Photo"* is annotated with an instance **y** of *Event ("Veranstaltung")* which has a *takes place in month ("findetStattImMonat")* attribute with month **z** as value then **x** is also annotated with this month **z**.

### 4.1.2 The IKEN Architecture

The **IKEN** architecture can be divided into two main parts: 1.) the ontology data pre-processing and 2.) the architecture of the functional web-application. Figure 4.3 shows the ontology data pre-processing workflow that can be divided into four major steps:

1. **Consistency check**: During the development of the **IKEN** prototype the reasoner PELLET (version 1.5.1) was used for consistency checks and inference. This consistency check of the **IKEN** ontology has to be executed after every extension or alteration of the ontology – especially the TBox.

2. **Inference**: The consistent **IKEN** ontology base model is then further processed with PELLET to infer the implicit constructs in the ontology and thus making them explicit. The result of this pre-processing step is the inferred **IKEN** ontology model.

3. **Serialization**: This consistent and inferred ontology version is then serialized into N-Triples format using JENA2 that can be used as input format for the next processing step.

4. **Deep integration**: In the last pre-processing step the OWL model is then transfered from N-Triple into a deep integrated Ruby representation.

I used the RUBY ON RAILS web development framework for the implementation of the **IKEN** prototype. RUBY ON RAILS supports the MVC architectural pattern that I further extended with semantic web capabilities using DEEP SEMANTICS. The MVC pattern consists of three units Model, View and Controller. The main parts of the semantic extended **IKEN** architecture as shown in Figure 4.4 are:

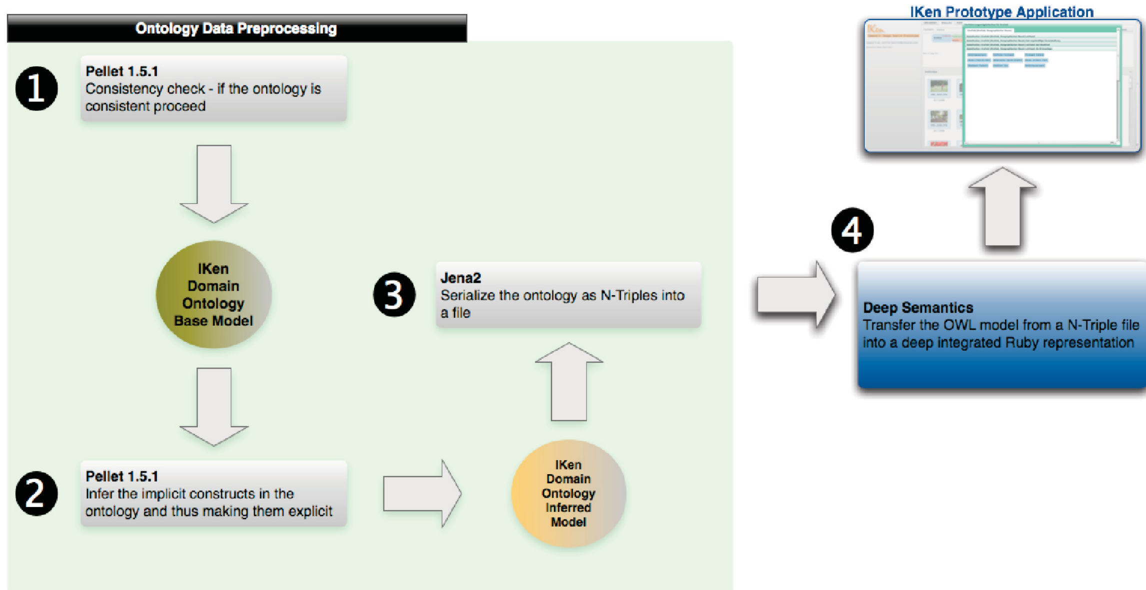


**Figure 4.3: Ontology data pre-processing for IKEN.** Ontology data pre-processing workflow divided into four major steps: 1) consistency check, 2) inference, 3) serialization and 4) deep integration.

---

1. **Client request**: The **IKEN** application running in the users web-browser as a FLEX3 process sends a request to the **IKEN** web server.

2. **Database access**: The controller which is responsible for this kind of request fetches the required data for the intended response from the database. It fetches all required data from the database except of the functional ontology data. The database is thereby running on a database server.

3. **DEEP SEMANTICS access**: If necessary the same controller retrieves the required ontological data saved in the **IKEN** ontology using DEEP SEMANTICS– required ontological data and constructs, respectively, can be for example certain classes or instances. The DEEP SEMANTICS process runs on the application server.

4. **Response preparation**: The fetched and processed data is passed from the controller to the respective view.

5. **Server response**: The view in turn combines this dynamic data with the page template that together constitute the visual appearance of the response that is send from the web server to the client.

6. **Ontology data processing**: Initially – that means at the start up of the **IKEN** application – the used ontology data is read by the DEEP SEMANTICS process either from the database or from an OWL file. Likewise is new annotation data in the ABox saved in the database when triggered by the administrator.

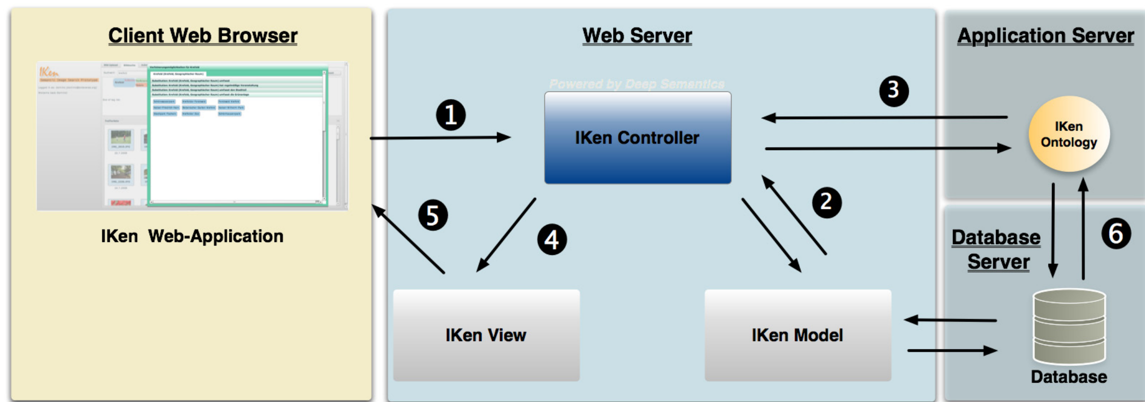


**Figure 4.4: IKEN architecture and implementation set-up.** The architecture of **IKEN** is basically a semantic extension of the Mode-View-Controller architectural pattern. Besides the web server the implementation set-up consist of an application server running the DEEP SEMANTICS process which is responsible for the ontology processing and the database server.

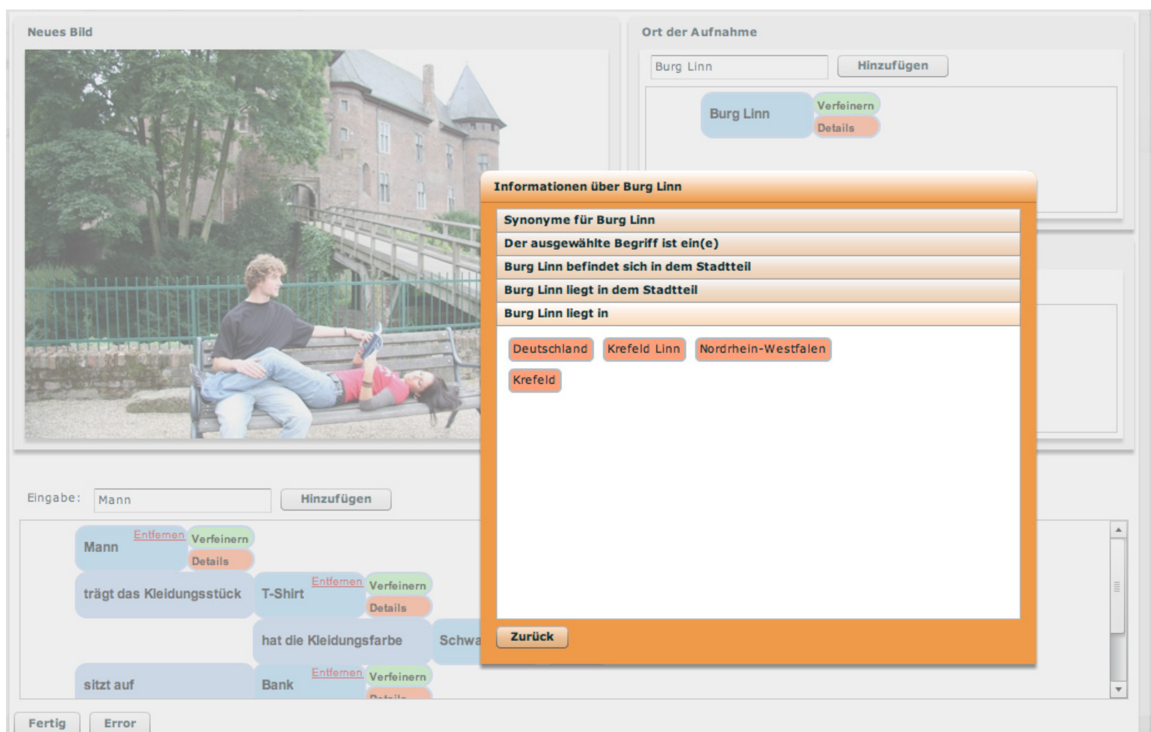


**Figure 4.5: IKEN details view in the annotation interface.** The details view of the term *"Burg Linn"*. The details about where this castle is situated are: *"Krefeld Linn"*, *"Krefeld"*, *"North Rhine-Westphalia"* and *"Germany"*.

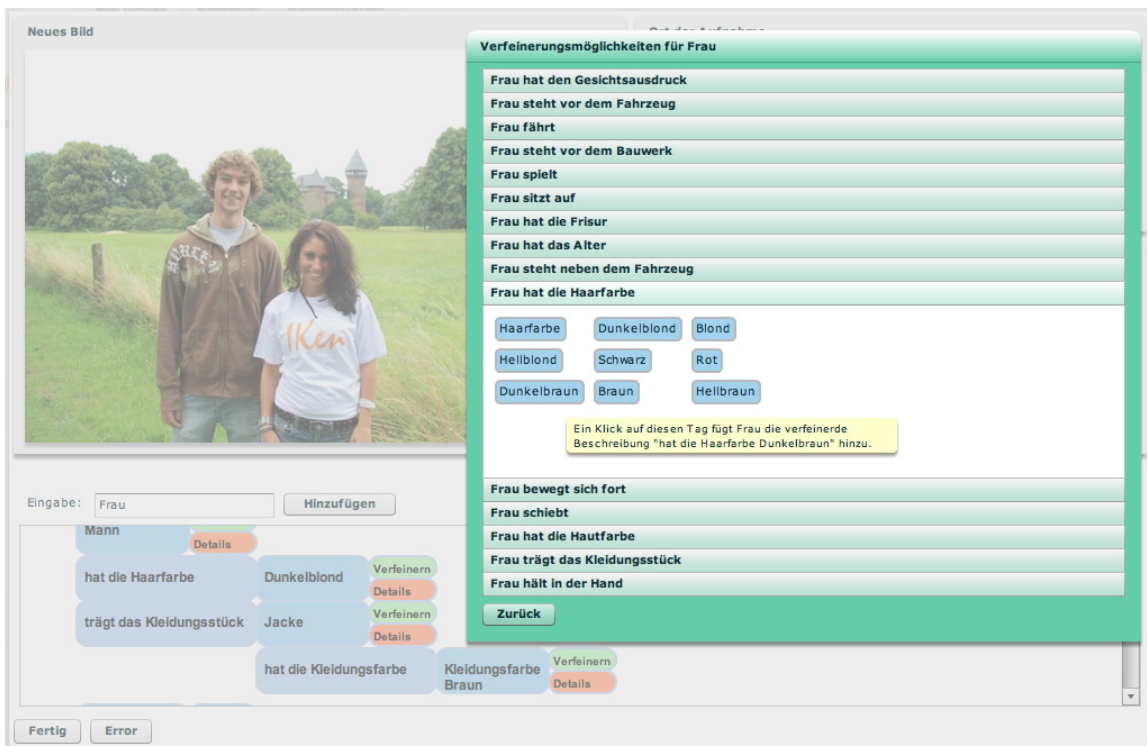


**Figure 4.6: IKEN refinement view in the annotation interface.** This screenshot of the systems refinement view shows the specification possibilities for the semantic tag *"Frau" (woman)*. Currently selected is the possibility to indicate the colour of the woman's hair.

### 4.1.3 The IKEN Graphical User Interface

In **IKEN** the concept interrelations defined in the underlying ontology can be exploited through the annotation user interface. Separated fields have been created for the annotation of the location where the image has been taken and for emotional impressions. The main field captures the content-descriptive annotations. In a first step, concepts can be added to a photo by entering a word – the system will look for a matching ontology concept and add this to the photo. This matching process uses the *rdfs:label* assertions in the ontology which also include synonyms and spelling variants. If no matching concept is found, the annotation term is added as a simple tag (in a future version these unrelated tags can be used as suggestions for new ontology concepts). In a next step, details for this concept can be displayed and the annotation may be refined. For example, the concept *"Burg Linn"* (a castle) has been entered. One may now choose the *"Details"* button to see how this concept is related to other aspects in the ontology, for example that it is situated in: *Deutschland* (Germany), *Nordrhein-Westfalen* (a german state), *Krefeld* (a city), *Krefeld Linn* (a district of Krefeld) (see Figure 4.5). Clicking on *"Verfeinern (refine)"* opens the option to directly specify the nature of a concept – in the case shown in Figure 4.6 for example one may choose to specify the *"hair colour"* of a *woman* as being *"dark brown"*.

An ontology-based retrieval system like **IKEN** can select only those pictures, for which the annotation concepts have been specified to have certain property values. Figure 4.7 for example shows the search for *"a woman (Frau) that wears the piece of clothing (trägt das Klei-*

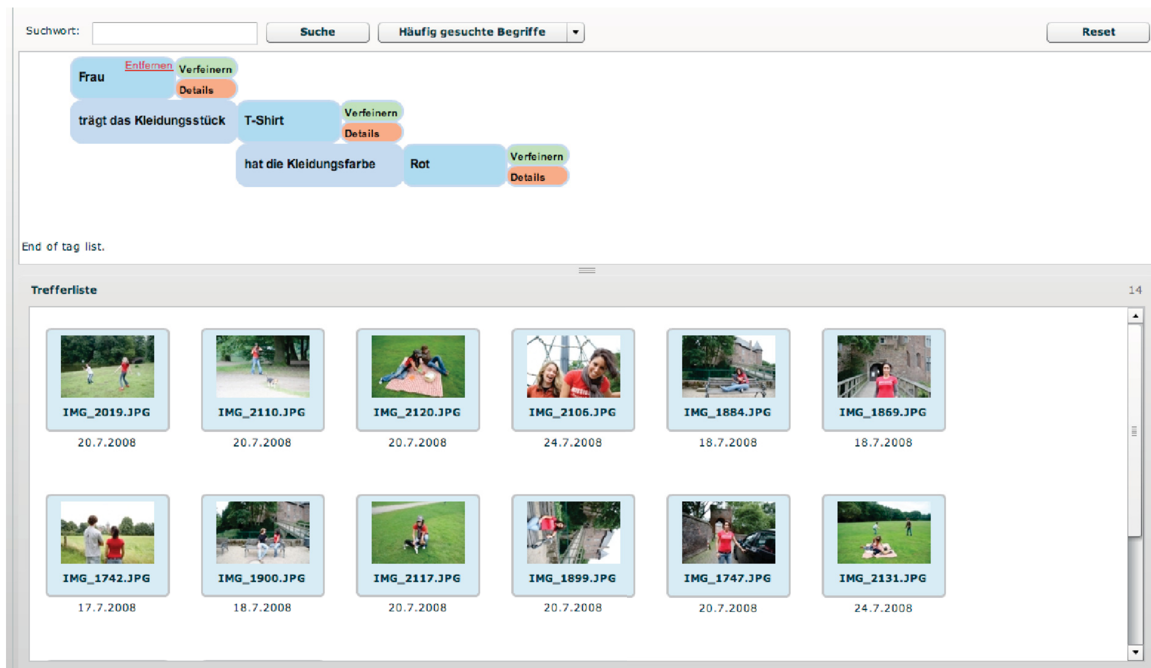


**Figure 4.7: IKEN search interface.** The screenshot shows the search results for the following query: *"A woman who wears a red T-shirt"*.

*dungsstück) T-shirt which has the colour red (Rot)"* using the **IKEN** prototype. To make use of the semantic annotations for precise document retrieval in the **IKEN** user interface, search terms can be added in the same way as annotation concepts. The user may enter several search terms which will again by replaced by corresponding ontology concepts and may refine each of them with the available properties. Details regarding the interrelations to other aspects can be shown for every concept. This may offer suggestions for manual query reformulation and refinement as well as it constitutes a navigation tool in the domain knowledge space.

## 4.2 The BIO2ME Ontology and Information System

### 4.2.1 BIO2ME Ontology

The **B**io**I**nformatics **O**ntology for **T**ools and **M**ethods (**BIO2ME**) is an ontology about the domain of bioinformatics tools and methods (Mainz, 2006). It was motivated by a work ((Wilm *et al.*, 2006)) which was done by members of the **ONTOVERSE** project. In this work, several multiple sequence alignment tools were compared regarding selected input data. One conclusion of this work was that the most popular and mostly employed program in this field, CLUSTALW, provided not necessarily the best results for each data set. The example of CLUSTALW (Chenna *et al.*, 2003) exemplifies the current situation in bioinformatics: There exist a variety of programs, packages, databases etc. dealing with various problems like the efficient processing of experimental data, sequence analysis and protein structure prediction and visualization. The problem for biologists currently is, that the domain of bioinformatics methods cannot be sur-

veyed with reasonable effort. Even for experts in a specific domain of bioinformatics it is sometimes difficult to decide which tool fits the given requirements best. Moreover, there is a plethora of programs which incorporate miscellaneous computational, mathematical, and biological approaches, respectively, solving the same problems.

The **BIO2ME** ontology is a detailed collection of information about bioinformatics tools, which are categorized according to their application ranges. Supported biological tasks, utilized computational methods, processed data formats and support information of tools are captured, too. The ontology provides a basis of a search for tools that meet an users needs and also provides information about certain tools and computational methods. It will answer questions as for example: Which tools and methods exist, that deal with given problems? Which data output do they provide? etc.

### 4.2.2 BIO2ME Information System

Based on the **BIO2ME** ontology, and using DEEP SEMANTICS for ontology processing, an information system has been implemented by Mainz (2008). The **BIO2ME** information system provides functionalities to search for required bioinformatic tools and methods using the ontology defined semantics for query extension as well as query formulation.

Beside semantic information retrieval, the **BIO2ME** application also supports the extension of the ontology ABox. While this level of functionality does not comprise schema modification, as for example the ontology editors SWOOP (Kalyanpur *et al.*, 2005), PROTÉGÉ (Knublauch *et al.*, 2004) or **ONTOVERSE**, it does provide fast and consistency safe ontology instance extension and editing.

## 4.3 Discussion

In this chapter two semantic applications were described that both use DEEP SEMANTICS as ontology processing framework. The first one, the **IKEN** semantic image management application (Mainz *et al.*, 2008), provides functionalities to annotate images based on an underlying ontology. These ontology-based annotations can then be used to offer a semantical search engine over the images. The **IKEN** system was designed to enable users, who are not experts in knowledge engineering and ontology modelling, to apply semantic annotations and make use of semantic information retrieval in image collections. In the context of this application DEEP SEMANTICS was successfully applied to access the TBox of the ontology (as well as initially stored instances) to implement the semantic image annotation and retrieval functionalities.

The second described semantic application is called the **BIO2ME** information system. While this application was not developed during this thesis, it is discussed here as a further reference application using DEEP SEMANTICS for ontology access and manipulation.

### 4.3.1 Semantic Image Management with IKEN

Besides being a test implementation for DEEP SEMANTICS, the initial aims of **IKEN** were: A) to provide a usable application which makes benefits of ontologies visible for internet users; B) to enable easily usable functionalities for ontology-based photo annotations; C) to allow browsing a document collection based on domain semantics. First test runs with a small set of reference users showed that ontology supported image annotation and retrieval with **IKEN** can significantly improve user experience. To empirically support these first results, tests to provide the evaluation of precision, recall and usability are currently prepared. However, present user responses indicate that **IKEN**'s user interface highlights advantages of semantic interfaces and therefore successfully accomplish aim A). Additionally, users reported DEEP SEMANTICS to have easier usable functionalities for ontology-based photo annotations as compared to PHOTOSTUFF (Halaschek-Wiener *et al.*, 2005).

During the first test runs it turned out that trying to describe every visual object in the picture in detail is neither a feasible nor a desirable approach. For the presented version the concept *Person* was identified as being most important for the **IKEN** application. Consequently, a large part of used properties is related to class *Person* like *wearsClothing*, *hasHairColour* and *holdsInHand*. Furthermore, the applied SWRL rules proved to be beneficial for automatically determining additional facts and annotations, respectively. However, the use of reasoner PELLET with SWRL rules turned out to be ineffective regarding its runtime as it extended the required inference runtime fiftyfold from approximately 30 minutes to more than 24 hours.

An important result was the finding that terms used for photo annotations which cannot be mapped to the ontology constitute a valuable resource for ontology extension. Using these tags I was able to extend for example the subclasses for the concept *Clothing* with classes *Coat* and *Gentleman Suit*. Future work on the **IKEN** system will include functionalities to incorporate unrelated tags into the used ontology.

### 4.3.2 Experiences applying DEEP SEMANTICS

The primary purpose of the implementation of the semantic image annotation system **IKEN**, with respect to this thesis, was to build a reference application based on DEEP SEMANTICS. Experiences during the development of **IKEN** support the results discussed in Subsection 3.9.1 of Chapter 3. Deep integrated classes and instances were easily accessible and modifiable. For example the following *getter* methods for a class's set of instances were useful in a wide range of implemented functionalities: *direct_instances*, *initial_instances* and *instances*. Similar experiences were reported by the developer of the **BIO2ME** information system.

The usage of DEEP SEMANTICS for the conversion of an OWL LITE ontology into RUBY classes and objects turned out to require the consideration of some special rules. Ontology class names for example have to begin with an uppercase character. This rule accounts for RUBY's naming conventions for classes. Additionally, RUBY classes and therefore ontology classes processed with DEEP SEMANTICS may consist of any combination of letters, numbers and underscores.

Furthermore, the following three best-practices were defined during the development of **IKEN** and **BIO2ME** regarding ontology preparation for DEEP SEMANTICS:

1. Every class, individual and property in the ontology should provide at least one natural language label – this recommendation among other things fosters the usefulness of convenient methods like *find_instance_by_label(label)*.

2. Each property should provide a short description that can be displayed in the search form to help the user understanding the search field.

3. *rdfs:comments* should be defined HTML conform when used to display in a browser.

Before the development of both reference applications convenient methods like *find_instance_by_label(label)* were not part of DEEP SEMANTICS. These convenient methods were added considering gathered experiences regarding commonly used ontology processing functionalities. The current version of DEEP SEMANTICS comprises convenient methods like:

- *find_instance_by_label(label)*: returns all instances of a class matching the passed label.

- *find_instance_by_local_name(local_name)*: returns all instances of a class matching the passed local name.

- *find_subclasses_by_label(label)*: returns all subclasses of a class matching the passed label.

- *incompleted_object_properties*: returns those object properties that can still be used to state additional facts about an instance without overwriting existing values.

- *find_classes_by_superclass(superclass)*: returns those classes in the ontology that have the defined superclass.

- *listClassesByLevel* respectively *list_classes_by_level*: lists classes by their highest levels (for example if a class occurs on the levels 2 and 3 in the hierarchy it is saved for level 3).

The discussed convenient methods support DEEP SEMANTICS focus on developer requirements and provide a sophisticated usability layer on top of the deep integrated ontology model. Additional types of convenient or helper methods are discussed next in the conclusions and outlook section.

### 4.3.3 Conclusions and Outlook

The implementations of both reference applications proofed DEEP SEMANTICS to be an easy to use *Semantic Web* framework. Additional helper and convenient methods, respectively, were identified that further extended DEEP SEMANTICS potential to become the framework of choice for rapid *Semantic Web* development within the next few years.

Runtime and memory complexity were unproblematic and did not provide any practical obstacles at all. However, handling of the deep integration process should be optimized in future version. For example by enabling the deep integration of the TBox to be performed once including storing of the resulting functional model. Thus, allowing this model to be included into an application source code without having to convert the related ontology yet another time.

Besides improvements of the DEEP SEMANTICS's core functionalities (as discussed in Section 3.9 of the previous chapter), the implementation experiences described suggest a future extension of the provided set of convenient methods. Relevant candidates are new methods for fetching instances by their property values.

# 5

# Summary

The arising Semantic Web is an important component of the Internet of the future. Semantic applications provide encouraging possibilities to control the exponentially growing data and information amount. Despite its considerable application potential, the idea of the Semantic Web is not enforced on a broad range yet. Thereby there are two problems of particular relevance: existing Semantic Web frameworks are either difficult to learn or cause problems because they do not avoid the generation of logical inconsistencies.

The in this work developed, fundamentally new Semantic Web framework DEEP SEMANTICS first provides an efficient and quickly to learn approach to edit ontologies, second integrated consistency checks and third a fast processing of the data. Through the algorithmic realization of the so called deep integration approach, with DEEP SEMANTICS it is possible to convert OWL LITE ontologies in RUBY code. As a result a functional model of the ontology is generated in RUBY. The editing of ontologies is more efficient because less lines of source code achieve the same functionality as source code of conventional API-based Semantic Web frameworks. Moreover, the access to ontologies using DEEP SEMANTICS is much more intuitive and therefore easier to learn. The integrated consistency check corresponds to an important feature of DEEP SEMANTICS: consistency safeness. DEEP SEMANTICS is the first Semantic Web framework of its kind, which guarantees the logic consistency of the modified ontology. As the framework processes only those operations which cannot cause logical inconsistencies, it provides the developer of semantic applications a security in handling semantic data whose importance should not be underestimated. Thereby, DEEP SEMANTICS reveals an excellent operation time within the ontology editing.

Further on, DEEP SEMANTICS was successfully applied for the implementation of the semantic image management application **IKEN**. Using **IKEN**, images can be annotated and searched on the basis of an ontology. Thereby, the application offers a novel approach for the utilization of semantic context information in graphical user interfaces. Depending on the entered term, **IKEN** presents the user additional information stored in the ontology. During the implementation of **IKEN** all three above mentioned advantages could be affirmed with respect

to DEEP SEMANTICS. Additionally, new suggestions for the extension of DEEP SEMANTICS have been motivated, that lead to the implementation of novel, efficiency improving convenient functions.

Furthermore, DEEP SEMANTICS promises to become a valuable building block for semantic applications in bioinformatics. In this thesis it was possible to discuss this on the basis of the results of the **BIO2ME** project. In summary of the presented work results it appears to be possible that DEEP SEMANTICS could become an important part of the Semantic Web.

# 6

# Zusammenfassung

Das entstehende Semantic Web ist ein wichtiger Bestandteil des Internet der Zukunft. Semantische Anwendungen bieten vielversprechende Möglichkeiten um die exponentiell wachsenden Daten- und Informationsbestände zu beherrschen. Trotz seines bedeutenden Anwendungspotenzials hat sich die Idee des Semantic Web allerdings bis heute nicht auf breiter Basis durchsetzen können. Dabei sind zwei Probleme von besonderer Relevanz: bestehende Semantic Web Frameworks sind entweder für Entwickler schwer zu erlernen, oder bereiten Probleme da sie die Erzeugung von logischen Inkonsistenzen nicht verhindern.

Das in dieser Arbeit entwickelte, fundamental neue Semantic Web Framework DEEP SEMANTICS bietet erstens einen effizienten und schnell zu erlernende Ansatz Ontologien zu bearbeiten, zweitens integrierte Konsistenzüberprüfungen und drittens eine schnelle Datenprozessierung. Durch die algorithmische Umsetzung des sogenannte Deep Integration Ansatzes ist es mit DEEP SEMANTICS möglich, OWL LITE Ontologien in RUBY Programmcode zu übersetzen. Als Resultat entsteht ein funktionales Modell der Ontologie in RUBY. Die Ontologiebearbeitung gestaltet sich damit wesentlich effizienter, da weniger Zeilen Programmcode die gleichen Funktionalitäten erreichen als Programmcode herkömmlicher API-basierter Semantic Web Frameworks. Zudem ist der Zugriff auf die Ontologiedaten über DEEP SEMANTICS wesentlich intuitiver und damit leichter zu erlernen. Die integrierte Konsistenzüberprüfung bezieht sich auf ein wichtiges Merkmal von DEEP SEMANTICS: Konsistenzsicherheit. DEEP SEMANTICS ist das erste Semantic Web Framework seiner Art, das die logische Konsistenz der bearbeiteten Ontologie sicherstellt. Indem das Framework nur solche Operationen ausführt, die keine logischen Inkonsistenzen hervorrufen können, gibt es den Entwicklern von semantischen Applikationen eine nicht zu unterschätzendes neue Sicherheit im Umgang mit semantischen Daten. Dabei zeigt DEEP SEMANTICS zudem ein hervorragendes Laufzeitverhalten bei der Ontologiedatenbearbeitung.

Weiterhin wurde DEEP SEMANTICS in dieser Arbeit erfolgreich für die Implementierung der semantischen Bildverwaltungsanwendung **IKEN** eingesetzt. Bilder können über **IKEN** anhand einer Ontologie annotiert und gesucht werden. Dabei bietet die Anwendung einen

neuartigen Ansatz für die Verwendung semantischer Kontextinformationen in graphischen Benutzerschnittstellen. Je nachdem welches Schlagwort in das System eingegeben wird, präsentiert **IKEN** dem Anwender in der Ontologie hinterlegte Zusatzinformationen. In Bezug auf DEEP SEMANTICS bestätigten sich bei der Umsetzung von **IKEN** die drei oben aufgeführten Vorteile. Zusätzlich konnten neue Anregungen für die Erweiterung von DEEP SEMANTICS gewonnen werden, die zur Implementierung von neuen, effizienzsteigernden Hilfsfunktionen führten.

DEEP SEMANTICS verspricht zudem ein wertvoller Baustein für semantische Anwendungen in der Bioinformatik zu werden. Dies konnte in dieser Arbeit anhand der Ergebnisse des **BIO2ME** Projektes diskuttiert werden. Zusammenfassend lassen es die in dieser Arbeit präsentierten Ergebnisse möglich erscheinen, dass DEEP SEMANTICS zu einem wichtigen Bestandteil des Semantic Web werden wird.

# Bibliography

(2004). RDF Vocabulary Description Language 1.0: RDF Schema. http://www.w3.org/TR/rdf-schema/. 1.2.1

Ahmad, Ashraf M. A. (2007). Multimedia content and the semantic web: Methods, standards and tools: Book Reviews. *J. Am. Soc. Inf. Sci. Technol.*, 58(3), 457–458. 1.1, 3

Ashburner, M., Ball, C. A., Blake, J. A., Botstein, D., Butler, H., Cherry, J. M., Davis, A. P., Dolinski, K., Dwight, S. S., Eppig, J. T., Harris, M. A., Hill, D. P., Issel-Tarver, L., Kasarskis, A., Lewis, S., Matese, J. C., Richardson, J. E., Ringwald, M., Rubin, G. M. & Sherlock, G. (2000). Gene ontology: tool for the unification of biology. The Gene Ontology Consortium. *Nat Genet*, 25(1), 25–29. 2, 2.4.1

Avraham, S., Tung, C. W., Ilic, K., Jaiswal, P., Kellogg, E. A., McCouch, S., Pujar, A., Reiser, L., Rhee, S. Y., Sachs, M. M., Schaeffer, M., Stein, L., Stevens, P., Vincent, L., Zapata, F. & Ware, D. (2008). The Plant Ontology Database: a community resource for plant structure and developmental stages controlled vocabulary and annotations. *Nucleic acids research*, 36(Database issue). 2.4

Baader, Franz, Calvanese, Diego, McGuinness, Deborah L., Nardi, Daniele & Patel-Schneider, Peter F. (2003). *The description logic handbook: Theory, implementation, and applications*. Cambridge University Press, Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, United Kingdom. 2.2

Babik, Marian & Hluchy, Ladislav (2006). Deep integration of Python with Web Ontology Language. In *Proc. of 2nd Workshop on Scripting for the Semantic Web*. 1.2.1

Baxevanis, Andreas D. (2000). The Molecular Biology Database Collection: an online compilation of relevant database resources. *Nucl. Acids Res.*, 28(1), 1–7. 1.2

Baxevanis, Andreas D. (2001). The Molecular Biology Database Collection: an updated compilation of biological database resources. *Nucl. Acids Res.*, 29(1), 1–10. 1.2

Baxevanis, Andreas D. (2002). The Molecular Biology Database Collection: 2002 update. *Nucl. Acids Res.*, 30(1), 1–12. 1.2

Baxevanis, Andreas D. (2003). The Molecular Biology Database Collection: 2003 update. *Nucl. Acids Res.*, 31(1), 1–12. 1.2

Bechhofer, S., van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D.L., Patel-Schneider, P. & Stein, L.A. (2004). OWL Web Ontology Language Reference. W3C Recommendation. 1.2, 3.9

Bechhofer, Sean, Volz, Raphael & Lord, Phillip (2003). Cooking the Semantic Web with the OWL API. In *In Proc. 2nd International Semantic Web Conference (ISWC), Sanibel Island (FL US*. Springer, pp. 659–675. 2.7.2

Benson, Dennis A., Karsch-mizrachi, Ilene, Lipman, David J., Ostell, James & Wheeler, David L. (2004). GenBank: update. *Nucleic Acids Res*, 32, 23–26. 1.1

Blackburn, S. (2007). *The Oxford Dictionary of Philosophy*. Oxford University Press. 2.1.1

Bodenreider, Olivier & Stevens, Robert (2006). Bio-ontologies: current trends and future directions. *Brief Bioinform*, 7, 256–274. 2.4

Broekstra, Jeen, Kampman, Arjohn & van Harmelen, Frank (2002). Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. pp. 54+. 3.9.4

Carroll, Jeremy J., Dickinson, Ian, Dollin, Chris, Seaborne, Andy, Wilkinson, Kevin, Reynolds, Dave & Reynolds, Dave (2004). Jena: Implementing the semantic web recommendations. pp. 74–83. 1.2.1, 2.7.1

Cheng, Xu, Dale, Cameron & Liu, Jiangchuan (2007). Understanding the Characteristics of Internet Short Video Sharing: YouTube as a Case Study. 1.1

Chenna, Ramu, Sugawara, Hideaki, Koike, Tadashi, Lopez, Rodrigo, Gibson, Toby J., Higgins, Desmond G. & Thompson, Julie D. (2003). Multiple sequence alignment with the Clustal series of programs. *Nucl. Acids Res.*, **31**(13), 3497–3500. 4.2.1

Cheung, Kei-Hoi, Smith, Andrew, Yip, Kevin, Baker, Christopher & Gerstein, Mark (2007). Semantic Web Approach to Database Integration in the Life Sciences. pp. 11–30. 1.2

Christiansen, Tom & Torkington, Nathan (2003). *Perl Cookbook*. O'Reilly, second edition. 2.8

Delfs, Ralph, Doms, Andreas, Kozlenkov, Er & Schroeder, Michael (2004). GoPubMed: ontology-based literature search applied to GeneOntology and PubMed. In *In Proceedings of German Bioinformatics Conference. LNBI*. Springer, pp. 169–178. 3

Esmaili, K. Sheykh & Abolhassani, H. (2006). A Categorization Scheme for Semantic Web Search Engines. In *4th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA-06)*. 3

Feigenbaum, Lee, Martin, Sean, Roy, Matthew N., Szekely, Benjamin & Yung, Wing C. (2007). Boca: an open-source RDF store for building Semantic Web applications. *Brief Bioinform*, 8(3), 195–200. 3.9.4

Fernandez, O. (2005). Deep Integration of Ruby with Semantic Web Ontologies. 2.9

Finin, T., Mayfield, J., Joshi, A., Cost, R. S. & Fink, C. (2005). Information Retrieval and the Semantic Web. p. 113a. 3

Freeman, Elisabeth, Freeman, Eric, Bates, Bert & Sierra, Kathy (2004). *Head First Design Patterns*. O' Reilly & Associates, Inc. 3.2

Galperin, Michael Y. (2004). The Molecular Biology Database Collection: 2004 update. *Nucl. Acids Res.*, 32(1), D3–22. 1.2

Galperin, Michael Y. (2005). The Molecular Biology Database Collection: 2005 update. *Nucl. Acids Res.*, 33(1), D5–24. 1.2

Galperin, Michael Y. (2006). The Molecular Biology Database Collection: 2006 update. *Nucl. Acids Res.*, 34(1), D3–5. 1.2

Galperin, Michael Y. (2007). The Molecular Biology Database Collection: 2007 update. *Nucl. Acids Res.*, 35(1), D3–4. 1.2

Galperin, Michael Y. (2008). The Molecular Biology Database Collection: 2008 update. *Nucl. Acids Res.*, 36(1), D2–4. 1.2

Geer, D. (2006). Will software developers ride ruby on rails to success? *Computer*, 39(2), 18–20. 3.9.1

Goble, Carole Anne & Roure, David Charles De (2007). myExperiment: social networking for workflow-using e-scientists. In *WORKS '07: Proceedings of the 2nd workshop on Workflows in support of large-scale science*. ACM, New York, NY, USA, pp. 1–2. 1.3

Golbreich, Christine, Horridge, Matthew, Horrocks, Ian, Motik, Boris & Shearer, Rob (2007). OBO and OWL: Leveraging Semantic Web Technologies for the Life Sciences. In *ISWC/ASWC*. pp. 169–182. 2.4.1

Goldberg, Adele & Robson, David (1989). *Smalltalk 80 : The Language*. Addison-Wesley Series in Computer Science. Addison-Wesley Professional. 2.8

Graham, Paul (1995). *ANSI Common LISP*. Prentice Hall. 2.8

Grau, B, Horrocks, I., Motik, B., Parsia, B., Patel-Schneider, P. & Sattler, U. (2008). OWL 2: The next step for OWL. *Web Semantics: Science, Services and Agents on the World Wide Web*. 3.9.4

Grau, Bernardo Cuenca, Parsia, Bijan & Sirin, Evren (2004). Working with multiple ontologies on the semantic web. In *In International Semantic Web Conference*. Springer, pp. 620–634. 3.9.3

Grosof, Benjamin N. (2003). Description logic programs: Combining logic programs with description logic. ACM, pp. 48–57. 2.9

Gruber, Thomas R. & Gruber, Thomas R. (1993). A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5, 199–220. 1

Guarino, Nicola, Poli, Roberto, Publishers, Kluwer Academic, Substantial, In Press & Gruber, Thomas R. (1993). Toward Principles for the Design of Ontologies Used for Knowledge Sharing. In *In Formal Ontology in Conceptual Analysis and Knowledge Representation, Kluwer Academic Publishers, in press. Substantial revision of paper presented at the International Workshop on Formal Ontology*. 2.1.1

Haarslev, Volker & Moller, Ralf (2001). RACER system description. Springer, pp. 701–705. 2.6

Haarslev, Volker & Móller, Ralf (2003). Racer: A Core Inference Engine for the Semantic Web. In *In 2nd International Workshop on Evaluation of Ontology-based Tools (EON-2003), Sanibel Island, FL*. pp. 27–36. 1.2.1

Halaschek-Wiener, Christian, Golbeck, Jennifer, Schain, Andrew, Grove, Michael, Parsia, Bijan & Hendler, James (2005). PhotoStuff âĂŤ An Image Annotation Tool for the Semantic Web. In *Poster Proceedings of the 4th International Semantic Web Conference* (Gil, Yolanda, Motta, Enrico, Benjamins, Richard V. & Musen, Mark A., eds). 4.3.1

Halaschek-wiener, Christian, Schain, Andrew, Golbeck, Jennifer, Parsia, Bijan & Hendler, Jim (2005). A flexible approach for managing digital images on the semantic web. In *In Proceedings of the Fifth International Workshop on Knowledge Markup and Semantic Annotation (SemAnnot)*. Springer, pp. 49–58. 2.5.1

Hare, Jonathon S., Lewis, Paul H., Enser, Peter G. B. & Sandom, Christine J. (2006). Mind the gap: another look at the problem of the semantic gap in image retrieval. volume 6073. SPIE. 1.1

Hendler, James (2001). The semantic web. *Scientific American*, 284, 34–43. 1, 2.1

Hollink, Laura, Schreiber, Guus, Wielemaker, Jan & Wielinga, Bob (2003). Semantic annotation of image collections. In *In Workshop on Knowledge Markup and Semantic Annotation, KCAPâĂŹ03, 2003. Available athttp://www.cs.vu.nl/ËIJguus*. pp. 0–3. 1.1

Horridge, Matthew & Bechhofer, Sean (2007). Igniting the OWL 1.1 Touch Paper: The OWL API. In *In Proc. OWL-ED 2007, volume 258 of CEUR*. 1.2.1, 2.7.2

Horrocks, Ian, Patel-schneider, Peter F. & Harmelen, Frank Van (2003). From SHIQ and RDF to OWL: The Making of a Web Ontology Language. *Journal of Web Semantics*, 1, 2003. 3.9

Horrocks, Ian, Peter, Boley, Harold, Tabet, Said, Grosof, Benjamin & Dean, Mike (2003). SWRL: A Semantic Web Rule Language Combining OWL and RuleML. 2.1.5

Hu, Xiaohua, Lin, T. Y., Song, Il Y., Lin, Xia, Yoo, Illhoi, Lechner, Mark & Song, Min (2004). Ontology-Based Scalable and Portable Information Extraction System to Extract Biological Knowledge from Huge Collection of Biomedical Web Documents. In *WI '04: Proceedings of the 2004 IEEE/WIC/ACM International Conference on Web Intelligence*. IEEE Computer Society, pp. 77–83. 1.2

International (1995). *Ada 95 Reference Manual - ISO/8652-1995*. ISO. 2.8

Jasper, Robert & Uschold, Mike (1999). A framework for understanding and classifying ontology applications. pp. 16–21. 1.2

Kalyanpur, Aditya, Parsia, Bijan, Sirin, Evren, Grau, Bernardo Cuenca & Hendler, James (2005). Swoop: A Web Ontology Editing Browser. *Journal of Web Semantics*, 4, 2005. 4.2.2

Kalyanpur, Aditya, Parsia, Bijan, Sirin, Evren, Grau, Bernardo C. & Hendler, James (2006). Swoop: A Web Ontology Editing Browser. *Web Semantics: Science, Services and Agents on the World Wide Web*, 4(2), 144–153. 2.7.2

Kanellopoulos, Dimitris N. & Kotsiantis, Sotiris B. (2007). Semantic Web: A state of the art survey. 2.1

Klyne, Graham & Carroll, Jeremy J. (2004). Resource Description Framework (RDF): Concepts and Abstract Syntax. 1.2.1

Knublauch, H., Musen, M. & Rector, A. (2004). Editing description logics ontologies with the Protégé OWL plugin. 2.7.2, 4.2.2

Lacy, Lee W. (2005). *Owl: Representing Information Using the Web Ontology Language*. Trafford Publishing. 1.2.1

Liebig, Thorsten, Luther, Marko, Paolucci, Massimo, Wagner, Matthias & Henke, Friedrich Von (2005). Building Applications and Tools for OWL – Experiences and Suggestions. 3.9.3

Lu, Zhiyong, Cohen, K. Bretonnel & Hunter, Lawrence (2006). Finding GeneRIFs via Gene Ontology Annotations. In *in Pacific Symposium on Biocomputing. 2006. Maui*. World Scientific Publishing Co. Pte. Ltd, pp. 52–63. 3

Mainz, Dominic, Weller, Katrin & Mainz, Jürgen (2008). Semantic Image Annotation and Retrieval with IKen. In *International Semantic Web Conference (Posters & Demos)*. 4.3

Mainz, Indra (2006). Development of a prototype ontology for bioinformatics tools. (in german). Bachelor thesis, Heinrich-Heine-Universität Düsseldorf. 4.2.1

Mainz, Indra (2008). Development and Implementation of Techniques for Ontology Engineering and an Ontology-based Search for Bioinformatics Tools and Methods. 1.3, 3.8, 4, 4.2.2

Meyer, B. (1992). *Eiffel: The Language*. Prentice Hall, Englewood Cliffs, NJ, USA. 2.8

Oren, Eyal & Delbru, Renaud (2006). ActiveRDF: Object-oriented RDF in Ruby. In *In Scripting for Semantic Web (ESWC)*. 1.2.1, 2.7.3

Oren, E., Haller, A., Hauswirth, M., Heitmann, B., Decker, S. & Mesnage, C. (2007). A flexible integration framework for semantic web 2.0 applications. *IEEE Software*, 24(5), 64–71. 1.3

Ousterhout, John K. (1998). Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31, 23–30. 1.2.1

Parsia, Bijan & Sirin, Evren (2004). Pellet: An owl dl reasoner. In *In Third International Semantic Web Conference - Poster*. p. 2003. 1.2.1, 2.6, 3.3

Pesole, Graziano (2008). What is a gene? An updated operational definition. *Gene*, 417(1-2), 1–4. 1.1

Peter, Hayes, Patrick & Horrocks, Ian (2004). OWL Web Ontology Language – Semantics and Abstract Syntax.
urlhttp://www.w3.org/TR/owl-semantics/. 2.7.2, 3.3, 6.1

Roure, David De & Goble, Carole (2007). myexperiment ? a web 2.0 virtual research environment. In *International Workshop on Virtual Research Environments and Collaborative Work Environments*. 1.3

Sahoo, Satya S., Thomas, Christopher & Sheth, Amit (2006). Knowledge modeling and its application in life sciences: A tale of two ontologies. In *In Proceedings of WWW*. p. 2006. 1.1

Smith, Barry, Ashburner, Michael, Rosse, Cornelius, Bard, Jonathan, Bug, William, Ceusters, Werner & Eilbeck, Louis J Goldberg Karen (2007). The OBO Foundry: coordinated evolution of ontologies to support biomedical data integration. *Nature Biotechnology*, 25, 1251–1255. 2.4.1

Soldatova, Larisa & King, Ross (2007). Ontology Engineering for Biological Applications. pp. 121–137. 2.4

Soman, Sunil & Krintz, Chandra (2007). Application-specific garbage collection. *J. Syst. Softw.*, 80(7), 1037–1056. 1.2.1

Stevens, Robert, Aranguren, Mikel, Wolstencroft, Katy, Sattler, Ulrike, Drummond, Nick, Horridge, Matthew & Rector, Alan (2007). Using OWL to model biological knowledge. *Int. J. Hum.-Comput. Stud.*, 65(7), 583–594. 2.4

Stevens, Robert, Goble, Carole A. & Bechhofer, Sean (2000). Ontology-based knowledge representation for bioinformatics. *Brief Bioinform*, 1(4), 398–414. 1.1

Thomas, Dave, Fowler, Chad & Hunt, Andy (2004). *Programming Ruby: The Pragmatic Programmers' Guide, Second Edition*. Pragmatic Bookshelf. 1.2.1

Thomas, Dave, Hansson, David, Breedt, Leon, Clark, Mike, Fuchs, Thomas & Schwarz, Andrea (2005). *Agile Web Development with Rails (The Facets of Ruby Series)*. Pragmatic Bookshelf. 1.3

Tsarkov, Dmitry & Horrocks, Ian (2006). FaCT++ description logic reasoner: System description. In *In Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006)*. Springer, pp. 292–297. 1.2.1, 2.6

Van Rossum, G. (2003). *The Python Language Reference Manual*. Network Theory Ltd. 1.2.1

van Zwol, Roelof (2007). Flickr: Who is Looking? In *WI '07: Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence*. IEEE Computer Society, Washington, DC, USA, pp. 184–190. 1.1

Vrandecic, Denny (2005). Deep integration of scripting language and semantic web technologies. In *In Proceedings of the ESWC Workshop on Scripting for the Semantic Web*. 1.2.1

Wilm, Andreas, Mainz, Indra & Steger, Gerhard (2006). An enhanced RNA alignment benchmark for sequence alignment programs. *Algorithms Mol Biol*, 1, 19. 4.2.1

Yoder, Jeremy B. & Shneiderman, Ben (2008). Science 2.0: Not So New? *Science*, 320(5881), 1290–1291. 3.7.3

# Appendix

## 6.1 XPERIMENTR Ontology: Classes, Properties and Instances

**The XPERIMENTR classes:**

$$
\begin{aligned}
LaboratoryEquipment &\sqsubseteq Thing\\
LaboratoryElectronicEquipment &\sqsubseteq LaboratoryEquipment\\
LaboratoryGlassware &\sqsubseteq LaboratoryEquipment\\
LaboratoryPlasticEquipment &\sqsubseteq LaboratoryEquipment\\
LaboratoryMaterial &\sqsubseteq Thing\\
Antibiotic &\sqsubseteq LaboratoryMaterial\\
Buffer &\sqsubseteq LaboratoryMaterial\\
Chemical &\sqsubseteq LaboratoryMaterial\\
Enzyme &\sqsubseteq LaboratoryMaterial\\
Medium &\sqsubseteq LaboratoryMaterial\\
Organism &\sqsubseteq LaboratoryMaterial\\
Person &\sqsubseteq Thing\\
Protocol &\sqsubseteq Thing\\
BufferProtocol &\sqsubseteq Protocol\\
InVitroProtocol &\sqsubseteq Protocol\\
InVivoProtocol &\sqsubseteq Protocol\\
MediumProtocol &\sqsubseteq Protocol
\end{aligned}
$$

**The XPERIMENTR local restrictions:**

$$
\begin{aligned}
Buffer &\sqsubseteq \forall\, hasProtocol.BufferProtocol\\
Medium &\sqsubseteq \forall\, hasProtocol.MediumProtocol
\end{aligned}
$$

**The XPERIMENTR object properties:**

$$hasRequiredLaboratoryEquipment \sqsubseteq ObjectProperty$$
$$\exists hasRequiredLaboratoryEquipment.Thing \sqsubseteq \delta(Protocol)$$
$$Thing \sqsubseteq \forall\delta(hasRequiredLaboratoryEquipment.LaboratoryEquipment)$$
$$hasProtocol \sqsubseteq ObjectProperty$$
$$\exists hasProtocol.Thing \sqsubseteq \delta(LaboratoryMaterial)$$
$$Thing \sqsubseteq \forall\delta(hasProtocol.Protocol)$$
$$hasMediumProtocol \sqsubseteq hasProtocol$$
$$\exists hasMediumProtocol.Thing \sqsubseteq \delta(Medium)$$
$$Thing \sqsubseteq \forall\delta(hasMediumProtocol.MediumProtocol)$$
$$hasBufferProtocol \sqsubseteq hasProtocol$$
$$\exists hasBufferProtocol.Thing \sqsubseteq \delta(Buffer)$$
$$Thing \sqsubseteq \forall\delta(hasBufferProtocol.BufferProtocol)$$
$$hasRequiredMaterial \sqsubseteq ObjectProperty$$
$$\exists hasRequiredMaterial.Thing \sqsubseteq \delta(Protocol)$$
$$Thing \sqsubseteq \forall\delta(hasRequiredMaterial.LaboratoryMaterial)$$
$$hasBeenUploadedBy \sqsubseteq ObjectProperty$$
$$\exists hasBeenUploadedBy.Thing \sqsubseteq \delta(Protocol)$$
$$Thing \sqsubseteq \forall\delta(hasBeenUploadedBy.Person)$$
$$isExpertOf \sqsubseteq ObjectProperty$$
$$\exists isExpertOf.Thing \sqsubseteq \delta(Person)$$
$$Thing \sqsubseteq \forall\delta(isExpertOf.Protocol)$$
$$hasUploaded \sqsubseteq isExpertOf$$
$$\exists hasUploaded.Thing \sqsubseteq \delta(Person)$$
$$Thing \sqsubseteq \forall\delta(hasUploaded.Protocol)$$
$$isProtocolOfBuffer \sqsubseteq ObjectProperty$$
$$\exists isProtocolOfBuffer.Thing \sqsubseteq \delta(BufferProtocol)$$
$$Thing \sqsubseteq \forall\delta(isProtocolOfBuffer.Buffer)$$
$$isProtocolOfMedium \sqsubseteq ObjectProperty$$
$$\exists isProtocolOfMedium.Thing \sqsubseteq \delta(MediumProtocol)$$
$$Thing \sqsubseteq \forall\delta(isProtocolOfMedium.Medium)$$
$$isRequiredForProtocol \sqsubseteq ObjectProperty$$
$$\exists isRequiredForProtocol.Thing \sqsubseteq \delta(LaboratoryEquipment)$$
$$Thing \sqsubseteq \forall\delta(isRequiredForProtocol.Protocol)$$
$$isUsedForProtocol \sqsubseteq ObjectProperty$$
$$\exists isUsedForProtocol.Thing \sqsubseteq \delta(LaboratoryMaterial)$$
$$Thing \sqsubseteq \forall\delta(isUsedForProtocol.Protocol)$$

$$relatedExpert \sqsubseteq ObjectProperty$$
$$\exists relatedExpert.Thing \sqsubseteq \delta(Protocol)$$
$$Thing \sqsubseteq \forall\delta(relatedExpert.Person)$$
$$hasRequiredLaboratoryEquipment \equiv isRequiredForProtocol^{-}$$
$$hasRequiredMaterial \equiv isUsedForProtocol^{-}$$
$$isExpertOf \equiv relatedExpert^{-}$$
$$hasUploaded \equiv hasBeenUploadedBy^{-}$$

**The XPERIMENTR datatype properties:**

$$hasProcedure \sqsubseteq DatatypeProperty$$
$$\exists hasProcedure.Thing \sqsubseteq \delta(Protocol)$$
$$Thing \sqsubseteq \leq 1\ hasProcedure$$
$$hasRequiredExecutionTime \sqsubseteq DatatypeProperty$$
$$\exists hasRequiredExecutionTime.Thing \sqsubseteq \delta(Protocol)$$
$$Thing \sqsubseteq \leq 1\ hasRequiredExecutionTime$$

**The XPERIMENTR instances:**

Details about instances like property values and annotations have been omitted because the related list would have been to large.

Instances of *LaboratoryEquipment*:

$$BunsenBurner : LaboratoryEquipment$$
$$LoadingDye6x : LaboratoryEquipment$$
$$Micropipette : LaboratoryEquipment$$
$$SterileTip : LaboratoryEquipment$$
$$Incubator : LaboratoryElectronicEquipment$$
$$MicrowaveOven : LaboratoryElectronicEquipment$$
$$MotorizedPipette : LaboratoryElectronicEquipment$$
$$PCRMachine : LaboratoryElectronicEquipment$$
$$UVBox : LaboratoryElectronicEquipment$$
$$Vortexer : LaboratoryElectronicEquipment$$
$$GlassCultureTubes : LaboratoryGlassware$$
$$GlassPipetteTubes : LaboratoryGlassware$$
$$PlasticTube : LaboratoryPlasticEquipment$$
$$Sterile06mLPlasticTube : LaboratoryPlasticEquipment$$

Instances of *LaboratoryMaterial*:

$$Acrylamide : LaboratoryMaterial$$
$$BioPrimeRandomGenomicDNALabelingKit : LaboratoryMaterial$$
$$Parafilm : LaboratoryMaterial$$
$$PCRSupermix : LaboratoryMaterial$$
$$Primer : LaboratoryMaterial$$
$$QiagenTaqPolymeraseKit : LaboratoryMaterial$$
$$HighMWRunningBuffer5x : Buffer$$
$$LowMWRunningBuffer5x : Buffer$$
$$TAEBuffer : Buffer$$
$$Taq10xbuffer : Buffer$$
$$GelBuffer3P5X : Buffer$$
$$Acetate : Chemical$$
$$EDTA : Chemical$$
$$EthidiumBromide : Chemical$$
$$NaOH02M : Chemical$$
$$SDS : Chemical$$
$$SodiumBisulfite : Chemical$$
$$Tris : Chemical$$
$$ProteinaseK : Enzyme$$
$$ABMedium : Medium$$
$$GrowthMedium : Medium$$
$$YeastColony : Organism$$

Instances of *Person*:

$$Arndt : Person$$
$$Barry : Person$$
$$Deniz : Person$$
$$Dominic : Person$$
$$Ilija : Person$$
$$Indra : Person$$
$$Ingo : Person$$
$$Jochen : Person$$
$$Nahal : Person$$
$$Tim : Person$$

Instances of *Protocol*:

$$PCRSupermixProtocol : Protocol$$
$$HighMWRunningBuffer5xProtocol : BufferProtocol$$
$$LowMWRunningBuffer5xProtocol : BufferProtocol$$
$$TAEBufferProtocol : BufferProtocol$$
$$GelBuffer3P5XProtocol : BufferProtocol$$
$$AffymetrixDNALabellingForGeneExpressionArrays : InVitroProtocol$$
$$AgaroseGelElectrophoresis : InVitroProtocol$$
$$SDSPAGE : InVitroProtocol$$
$$BacterialCellCulture : InVivoProtocol$$
$$BlackburnYeastColonyPCR : InVivoProtocol$$
$$KnightColonyPCR : InVivoProtocol$$
$$MouseTissueLysisForGenotyping : InVivoProtocol$$
$$ABMediumProtocol : MediumProtocol$$

## 6.2 Abstract syntax of OWL LITE

**Ontologies**

```
ontology ::= 'Ontology(' [ ontologyID ] { directive } ')'
directive ::= 'Annotation(' ontologyPropertyID ontologyID ')'
            | 'Annotation(' annotationPropertyID URIreference ')'
            | 'Annotation(' annotationPropertyID dataLiteral ')'
            | 'Annotation(' annotationPropertyID individual ')'
            | axiom
            | fact

datatypeID ::= URIreference
classID ::= URIreference
individualID ::= URIreference
ontologyID ::= URIreference
datavaluedPropertyID ::= URIreference
individualvaluedPropertyID ::= URIreference
annotationPropertyID ::= URIreference
ontologyPropertyID ::= URIreference

annotation ::= 'annotation(' annotationPropertyID URIreference ')'
             | 'annotation(' annotationPropertyID dataLiteral ')'
             | 'annotation(' annotationPropertyID individual ')'
```

**Facts**

```
fact ::= individual
       | 'SameIndividual(' individualID individualID {individualID} ')'
       | 'DifferentIndividuals(' individualID individualID {individualID} ')'
individual ::= 'Individual(' [ individualID ] { annotation } { 'type(' type ')' } { value } ')'
value ::= 'value(' individualvaluedPropertyID individualID ')'
        | 'value(' individualvaluedPropertyID  individual ')'
        | 'value(' datavaluedPropertyID  dataLiteral ')'
type ::= classID | restriction

dataLiteral ::= typedLiteral | plainLiteral
typedLiteral ::= lexicalForm^^URIreference
plainLiteral ::= lexicalForm | lexicalForm@languageTag
lexicalForm ::= as in RDF, a unicode string in normal form C
languageTag ::= as in RDF, an XML language tag
```

**OWL Lite Class Axioms**

```
axiom ::= 'Class(' classID ['Deprecated'] modality { annotation } { super } ')'
        | 'EquivalentClasses(' classID classID { classID } ')'
        | 'Datatype(' datatypeID ['Deprecated']  { annotation } )'
modality ::= 'complete' | 'partial'
super ::= classID | restriction
```

**OWL Lite Restrictions**

```
restriction ::= 'restriction(' datavaluedPropertyID dataRestrictionComponent ')'
              | 'restriction(' individualvaluedPropertyID individualRestrictionComponent ')'
dataRestrictionComponent ::= 'allValuesFrom(' dataRange ')'
                           | 'someValuesFrom(' dataRange ')'
                           | cardinality
individualRestrictionComponent ::= 'allValuesFrom(' classID ')'
                                 | 'someValuesFrom(' classID ')'
                                 | cardinality
cardinality ::= 'minCardinality(0)' | 'minCardinality(1)'
              | 'maxCardinality(0)' | 'maxCardinality(1)'
              | 'cardinality(0)'    | 'cardinality(1)'
```

**OWL Lite Property Axioms**

```
axiom ::= 'DatatypeProperty(' datavaluedPropertyID ['Deprecated'] { annotation }
          { 'super(' datavaluedPropertyID ')' } ['Functional']
          { 'domain(' classID' ')' } { 'range(' dataRange ')' } ')'
        | 'ObjectProperty(' individualvaluedPropertyID ['Deprecated'] { annotation }
          { 'super(' individualvaluedPropertyID ')' }
          [ 'inverseOf(' individualvaluedPropertyID ')' ] [ 'Symmetric' ]
          [ 'Functional' | 'InverseFunctional' | 'Functional' 'InverseFunctional' | 'Transitive' ]
          { 'domain(' classID ')' } { 'range(' classID ')' } ')'
        | 'AnnotationProperty(' annotationPropertyID { annotation } ')'
        | 'OntologyProperty(' ontologyPropertyID { annotation } ')'
        | 'EquivalentProperties(' datavaluedPropertyID datavaluedPropertyID
          { datavaluedPropertyID } ')'
        | 'SubPropertyOf(' datavaluedPropertyID  datavaluedPropertyID ')'
        | 'EquivalentProperties(' individualvaluedPropertyID
          individualvaluedPropertyID { individualvaluedPropertyID } ')'
        | 'SubPropertyOf(' individualvaluedPropertyID  individualvaluedPropertyID ')'
dataRange ::= datatypeID | 'rdfs:Literal'
```

**Figure 6.1: EBNF of the abstract syntax of OWL LITE.** This representation of the abstract syntax of OWL LITE (Peter *et al.*, 2004) is specified by means of a version of Extended BNF. Terminals are quoted; non-terminals are **bold** and not quoted.

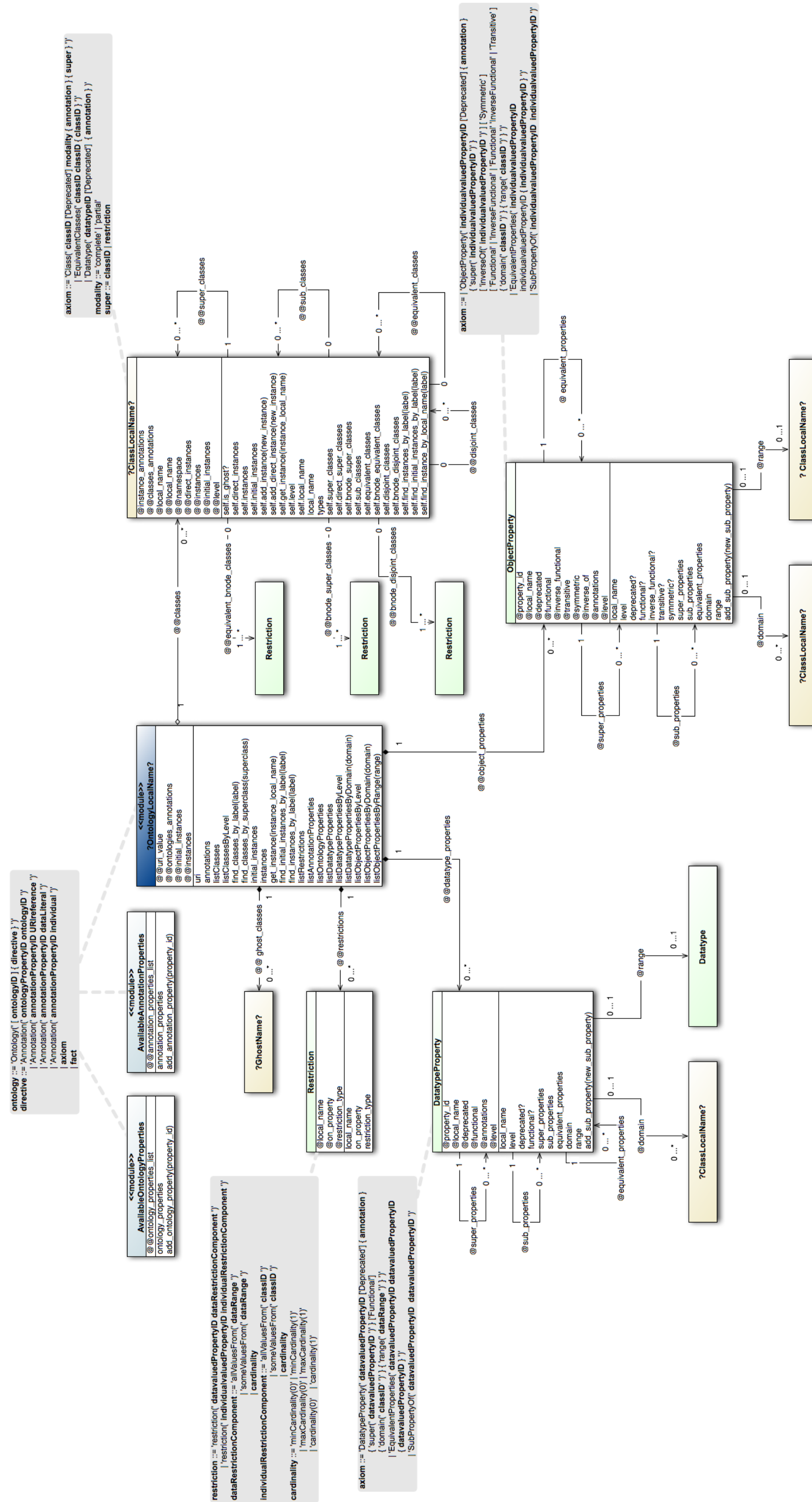

**Figure 6.2: Extended UML class diagram: EBNF of OWL LITE constructs in relation to their DEEP SEMANTICS classes and modules counterparts.**

## 6.3 Reference Test Implementations

### 6.3.1 Test 1: list all classes of the ontology

**Listing 6.1:** Test 1 implementation using OWL API.

```
1  package dissertation.xperimentr;
2
3  import org.semanticweb.owl.apibinding.OWLManager;
4  import org.semanticweb.owl.model.OWLClass;
5  import org.semanticweb.owl.model.OWLDescription;
6  import org.semanticweb.owl.model.OWLException;
7  import org.semanticweb.owl.model.OWLIndividual;
8  import org.semanticweb.owl.model.OWLOntology;
9  import org.semanticweb.owl.model.OWLOntologyCreationException;
10 import org.semanticweb.owl.model.OWLOntologyManager;
11
12 import java.net.URI;
13 import java.util.Date;
14 import java.util.Set;
15
16 public class IKenTestNewClusteringOWLAPI {
17
18         private OWLOntology ontology;
19
20     public IKenTestNewClusteringOWLAPI() {
21     }
22
23         public static void main(String[] args) {
24         try {
25                         Date date1 = new Date();
26                         System.out.println(date1.toString());
27
28                         // We first need to obtain a copy of an OWLOntologyManager, which
                               manages a set of
29                         // ontologies.
30                         OWLOntologyManager manager = OWLManager.createOWLOntologyManager();
31
32                         // We load an ontology from a physical URI - in this case the inferred
                                model of the KitchenMentor ontology.
33                         URI physicalURI = URI.create("file:/Users/dominic/ontologies/iken3/
                               iken_infered3.owl");
34
35                         // Now ask the manager to load the ontology
36                         OWLOntology ontology = manager.loadOntologyFromPhysicalURI(physicalURI
                               );
37
38                         IKenTestNewClusteringOWLAPI iken = new IKenTestNewClusteringOWLAPI();
39
40                         // Print out all of the classes which are referenced in the ontology
                               by hierarchy level
41                         Date timeA = new Date();
42                         System.out.println("List classes by level for the IKen ontology start:
                                "+timeA.toString());
43                         Set<OWLClass> klasses = ontology.getReferencedClasses();
44                         for (OWLClass klass : klasses) {
45                                 try {
46                                         iken.printHierarchy(ontology, klass);
47                                 } catch (OWLException e) {
48                                         System.out.println("The class hierarchy could not be
                                               printed: " + e.getMessage());
49                                 }
50                         }
51                         Date timeB= new Date();
```

```
52                          System.out.println("List classes by level for the IKen ontology end: "
                                +timeB.toString());
53                          System.out.println("Required time: "+(timeB.getTime()-timeA.getTime())
                                );
54
55                          Date date2 = new Date();
56                          System.out.println(date2.toString());
57                          System.out.println((date2.getTime()-date1.getTime()));
58                  } catch (OWLOntologyCreationException e) {
59                          // TODO Auto-generated catch block
60                          e.printStackTrace();
61                  }
62          }
63
64      public void printHierarchy(OWLOntology ontology, OWLClass clazz) throws OWLException {
65          this.ontology = ontology;
66          printHierarchy( clazz, 0 );
67      }
68
69      public void printHierarchy(OWLClass clazz, int level) throws OWLException {
70
71           System.out.println("Level: "+level);
72           System.out.println("  "+clazz);
73
74           /* Find this classes instances*/
75           System.out.println("  This classes direct instances:");
76           Set<OWLIndividual> instances = clazz.getIndividuals(this.ontology);
77           for (OWLIndividual instance : instances) {
78                   Boolean is_direct_instance = true;
79
80                   Set<OWLDescription> subclasses = clazz.getSubClasses(this.ontology);
81                   for (OWLDescription subclass : subclasses) {
82                           if (!subclass.isAnonymous() && !subclass.equals(clazz) && !subclass.
                                isOWLNothing()) {
83                                 if (instance.getTypes(this.ontology).contains(subclass)) {
84                                         is_direct_instance = false;
85                                 }
86                           }
87                   }
88
89                   if (is_direct_instance) {
90                          System.out.println("    "+instance);
91                   }
92           }
93           System.out.println("--------------------------");
94
95           /* Find the children and recurse */
96           Set<OWLDescription> children = clazz.getSubClasses(this.ontology);
97
98           for (OWLDescription child : children) {
99              if (!child.equals(clazz) && !child.isAnonymous() && !child.isOWLNothing() ) {
100                printHierarchy(child.asOWLClass(), level + 1);
101             }
102          }
103     }
104 }
```

**Listing 6.2:** Test 1 implementation using JENA2.

```
1 package dissertation.xperimentr;
2
3 import java.util.Date;
4 import org.mindswap.pellet.jena.PelletReasonerFactory;
5 import com.hp.hpl.jena.ontology.Individual;
6 import com.hp.hpl.jena.ontology.OntClass;
7 import com.hp.hpl.jena.ontology.OntModel;
```

```
8  import com.hp.hpl.jena.rdf.model.ModelFactory;
9  import com.hp.hpl.jena.util.iterator.ExtendedIterator;
10
11 public class IKenTestNewClustering {
12
13     public IKenTestNewClustering() {
14     }
15
16         public static void main(String[] args) {
17         Date date1 = new Date();
18         System.out.println(date1.toString());
19
20         String ontology = "file:/Users/dominic/ontologies/iken3/iken3_01112008.owl";
21
22         // create an empty ontology model using Pellet spec
23         OntModel model = ModelFactory.createOntologyModel( PelletReasonerFactory.THE_SPEC );
24
25         // read the file
26         model.read( ontology );
27         IKenTestNewClustering iken = new IKenTestNewClustering();
28
29         // Print out all of the classes which are referenced in the ontology by hierarchy
               level
30         Date timeA = new Date();
31         System.out.println("List classes by level for the IKen ontology start: "+timeA.
               toString());
32         for (ExtendedIterator i = model.listNamedClasses(); i.hasNext(); ) {
33                 OntClass klass = (OntClass) i.next();
34                 if ( klass.isHierarchyRoot() && (klass.listSuperClasses(true).toSet().size() >
                        0) ) {
35                         //System.out.println( klass.getURI() );
36                         iken.printHierarchy(klass, 0);
37                 }
38         }
39         Date timeB= new Date();
40         System.out.println("List classes by level for the IKen ontology end: "+timeB.toString
               ());
41         System.out.println("Required time: "+(timeB.getTime()-timeA.getTime()));
42
43         Date date2 = new Date();
44         System.out.println(date2.toString());
45         System.out.println((date2.getTime()-date1.getTime()));
46         }
47
48     public void printHierarchy(OntClass klass, int level) {
49
50           System.out.println("Level: "+level);
51
52           System.out.println("  "+klass.getLocalName());
53
54           /* Find this classes instances*/
55           System.out.println("  This classes direct instances:");
56           for (ExtendedIterator i = klass.listInstances(true); i.hasNext(); ) {
57                 Individual instance = (Individual) i.next();
58                 System.out.println("    "+instance.getLocalName());
59           }
60           System.out.println("--------------------------");
61
62           /* Find the children and recurse */
63           for (ExtendedIterator i = klass.listSubClasses(true); i.hasNext(); ) {
64                   OntClass subClass = (OntClass) i.next();
65                   if ( subClass.isURIResource() && (subClass.listSuperClasses(true).toSet().
                         size() > 0) && (subClass.listSubClasses(true).toSet().size() > 0)  ) {
66                         printHierarchy(subClass, level + 1);
67                   }
68           }
```

```
69      }
70  }
```

**Listing 6.3:** Test 1 implementation using DEEP SEMANTICS.

```
1  require File.join(File.dirname(__FILE__), 'active_semantics.rb')
2  require File.join(File.dirname(__FILE__), 'Helper/namespaces.rb')
3
4  time1 = Time.now
5
6  puts time1
7
8  Namespaces.add_namespace('http://www.ontoverse.org/ontologies/2008/3/iken2.owl#', 'iken:')
9
10 $active_semantics = ActiveSemantics.instance
11 iken = $active_semantics.set_director({'adapter' => 'FileAdapter', 'ontology_source' => '/
       Users/dominic/ontologies/iken3/inferred_iken3_01112008.nt', 'builder' => '
       DeepIntegrationBuilderOWLLite'})
12
13 timeA = Time.now
14 puts "List classes by level for the IKen ontology start: #{timeA}"
15 iken.listClassesByLevel.each_pair do |level, classes|
16   puts "Level: #{level}"
17   classes.each do |klass|
18     puts "  "+klass.local_name
19     puts "  This classes direct instances:"
20     klass.direct_instances.each do |instance|
21       puts "    "+instance.local_name
22     end
23     puts "--------------------------"
24   end
25 end
26 timeB = Time.now
27 puts "List classes by level for the IKen ontology end: #{timeB}"
28 puts "Required time: #{(timeB - timeA)}"
29
30 time2 = Time.now
31 puts time2
32
33 puts time2 - time1
```

## 6.3.2 Test 2: find all instances matching a certain search term

**Listing 6.4:** Test 2 implementation using OWL API.

```
1  package dissertation.xperimentr;
2
3  import org.semanticweb.owl.apibinding.OWLManager;
4  import org.semanticweb.owl.model.OWLAnnotation;
5  import org.semanticweb.owl.model.OWLClass;
6  import org.semanticweb.owl.model.OWLConstant;
7  import org.semanticweb.owl.model.OWLIndividual;
8  import org.semanticweb.owl.model.OWLOntology;
9  import org.semanticweb.owl.model.OWLOntologyCreationException;
10 import org.semanticweb.owl.model.OWLOntologyManager;
11 import org.semanticweb.owl.vocab.OWLRDFVocabulary;
12
13 import java.net.URI;
14 import java.util.Date;
15 import java.util.HashSet;
16 import java.util.Set;
17
18 public class IKenTestNewClustering2OWLAPI {
19
```

```
20      public IKenTestNewClustering2OWLAPI() {
21      }
22
23          public static void main(String[] args) {
24          try {
25                          Date date1 = new Date();
26                          System.out.println(date1.toString());
27
28                          // We first need to obtain a copy of an OWLOntologyManager, which
                                manages a set of
29                          // ontologies.
30                          OWLOntologyManager manager = OWLManager.createOWLOntologyManager();
31
32                          // We load an ontology from a physical URI - in this case the inferred
                                 model of the KitchenMentor ontology.
33                          URI physicalURI = URI.create("file:/Users/dominic/ontologies/iken3/
                                iken_infered3.owl");
34
35                          // Now ask the manager to load the ontology
36                          OWLOntology ontology = manager.loadOntologyFromPhysicalURI(physicalURI
                                );
37
38                          IKenTestNewClustering2OWLAPI iken = new IKenTestNewClustering2OWLAPI()
                                ;
39
40                          // Print out all of the classes which are referenced in the ontology
                                by hierarchy level
41                          Date timeA = new Date();
42                  System.out.println("Find all instances by their RDFS labels in the IKen
                        ontology - start: "+timeA.toString());
43                  Set<String> test_search_labels = new HashSet<String>();
44
45                  test_search_labels.add("bedeckt");
46                  test_search_labels.add("wolkenlos");
47                  test_search_labels.add("Locken");
48                  test_search_labels.add("Pony");
49                  test_search_labels.add("Grinsen");
50                  test_search_labels.add("Lachen");
51                  test_search_labels.add("Juli");
52                  test_search_labels.add("Oktober");
53                  test_search_labels.add("Natur");
54                  test_search_labels.add("Sport");
55
56                  for (String search_label : test_search_labels) {
57                          Set<OWLIndividual> instanceHits = new HashSet<OWLIndividual>();
58
59                                  Set<OWLClass> klasses = ontology.getReferencedClasses();
60                                  for (OWLClass klass : klasses) {
61                          instanceHits = iken.findInstancesByLabel(ontology, klass, search_label
                                );
62                          if (instanceHits.size() > 0) {
63                                  for (OWLIndividual instanceHit : instanceHits) {
64                                          System.out.println(instanceHit);
65                                  }
66                          }
67                                  }
68                  }
69                  Date timeB= new Date();
70                  System.out.println("Find all instances by their RDFS labels in the IKen
                        ontology - end: "+timeB.toString());
71                          System.out.println("Required time: "+(timeB.getTime()-timeA.getTime())
                                );
72
73                          Date date2 = new Date();
74                          System.out.println(date2.toString());
75                          System.out.println((date2.getTime()-date1.getTime()));
```

```
76                  } catch (OWLOntologyCreationException e) {
77                          // TODO Auto-generated catch block
78                          e.printStackTrace();
79                  }
80          }
81
82      public static Set<OWLIndividual> findInstancesByLabel(OWLOntology ontology, OWLClass klass
            , String searchLabel) {
83          Set<OWLIndividual> instance_hits = new HashSet<OWLIndividual>();
84          Set<OWLIndividual> instances = klass.getIndividuals(ontology);
85
86          for (OWLIndividual instance : instances) {
87                  for (OWLAnnotation annotation : instance.getAnnotations(ontology,
                        OWLRDFVocabulary.RDFS_LABEL.getURI())) {
88                          if (annotation.isAnnotationByConstant()) {
89                                  OWLConstant value = annotation.getAnnotationValueAsConstant();
90                                          if ( value.getLiteral().equals(searchLabel) ) {
91                                                  instance_hits.add( instance );
92                                          }
93                          }
94                  }
95          }
96                  return instance_hits;
97          }
98  }
```

**Listing 6.5:** Test 2 implementation using JENA2.

```
1   package dissertation.xperimentr;
2
3   import java.util.Date;
4   import java.util.HashSet;
5   import java.util.Iterator;
6   import java.util.Set;
7
8   import org.mindswap.pellet.jena.PelletReasonerFactory;
9   import com.hp.hpl.jena.ontology.Individual;
10  import com.hp.hpl.jena.ontology.OntClass;
11  import com.hp.hpl.jena.ontology.OntModel;
12  import com.hp.hpl.jena.rdf.model.Literal;
13  import com.hp.hpl.jena.rdf.model.ModelFactory;
14  import com.hp.hpl.jena.rdf.model.Resource;
15  import com.hp.hpl.jena.util.iterator.ExtendedIterator;
16
17  public class IKenTestNewClustering2 {
18
19      public IKenTestNewClustering2() {
20      }
21
22          public static void main(String[] args) {
23          Date date1 = new Date();
24          System.out.println(date1.toString());
25
26          String ontology = "file:/Users/dominic/ontologies/iken3/iken3_01112008.owl";
27
28          // create an empty ontology model using Pellet spec
29          OntModel model = ModelFactory.createOntologyModel( PelletReasonerFactory.THE_SPEC );
30
31          // read the file
32          model.read( ontology );
33
34          IKenTestNewClustering2 iken = new IKenTestNewClustering2();
35
36          // Print out all of the classes which are referenced in the ontology by hierarchy
                level
37          Date timeA = new Date();
```

```
38          System.out.println("Find all instances by their RDFS labels in the IKen ontology -
                start: "+timeA.toString());
39          Set<String> test_search_labels = new HashSet<String>();
40
41          test_search_labels.add("bedeckt");
42          test_search_labels.add("wolkenlos");
43          test_search_labels.add("Locken");
44          test_search_labels.add("Pony");
45          test_search_labels.add("Grinsen");
46          test_search_labels.add("Lachen");
47          test_search_labels.add("Juli");
48          test_search_labels.add("Oktober");
49          test_search_labels.add("Natur");
50          test_search_labels.add("Sport");
51
52          for (String search_label : test_search_labels) {
53                  Set<Individual> instanceHits = new HashSet<Individual>();
54
55              for (ExtendedIterator c = model.listNamedClasses(); c.hasNext(); ) {
56                  OntClass klass = (OntClass) c.next();
57                  instanceHits = iken.findInstancesByLabel(klass, search_label);
58                  if (instanceHits.size() > 0) {
59                          for (Individual instanceHit : instanceHits) {
60                                  System.out.println(instanceHit);
61                          }
62                  }
63              }
64          }
65
66          Date timeB= new Date();
67          System.out.println("Find all instances by their RDFS labels in the IKen ontology - end
                : "+timeB.toString());
68          System.out.println("Required time: "+(timeB.getTime()-timeA.getTime()));
69
70          Date date2 = new Date();
71          System.out.println(date2.toString());
72          System.out.println((date2.getTime()-date1.getTime()));
73          }
74
75      public static Set<Individual> findInstancesByLabel(OntClass klass, String searchLabel) {
76          //System.out.println("findInstancesByLabel: "+klass.toString());
77          Set<Individual> instance_hits = new HashSet<Individual>();
78          ExtendedIterator instances = klass.listInstances(true);
79
80          for (ExtendedIterator i = instances; i.hasNext(); ) {
81                  Individual instance = (Individual) i.next();
82                  for (ExtendedIterator j = instance.listLabels(null); j.hasNext(); ) {
83                          Literal label = (Literal) j.next();
84                                  if ( label.getString().equals(searchLabel) ) {
85                                          instance_hits.add( instance );
86                                  }
87                  }
88          }
89
90                  return instance_hits;
91          }
92  }
```

**Listing 6.6:** Test 2 implementation using ACTIVERDF.

```
1  require 'rubygems'
2
3  require 'active_rdf'
4  require "#{File.dirname(__FILE__)}/sparql"
5  require "#{File.dirname(__FILE__)}/rdflite"
6
```

```
7  time1 = Time.now
8  puts time1
9
10 adapter = ConnectionPool.add_data_source :type => :rdflite
11 adapter.load "iken_inferred3.nt"
12
13 # register the test namespace to the specified URI
14 Namespace.register :iken, 'http://www.ontoverse.org/ontologies/2008/3/iken2.owl#'
15
16 # and now construct the necessary Ruby Moduls and Classes
17 ObjectManager.construct_classes
18
19 timeA = Time.now
20 puts "Find all instances by their RDFS labels in IKen ontology - start: #{timeA}"
21 test_search_labels = ["bedeckt", "wolkenlos", "Locken", "Pony", "Grinsen", "Lachen", "Juli", "
     Oktober", "Natur", "Sport"]
22
23 test_search_labels.each do |search_label|
24   IKEN.constants.each do |const|
25     IKEN.const_get(const).find_all.each do |instance|
26       if instance.label.class != Array
27         if instance.label.eql?(search_label)
28           puts instance
29         end
30       else
31         instance.label.each do |label|
32           if label.eql?(search_label)
33             puts instance
34             break
35           end
36         end
37       end
38     end
39   end
40 end
41 timeB = Time.now
42 puts "Find all instances by their RDFS labels in IKen ontology - end: #{timeB}"
43 puts "Required time: #{(timeB - timeA)}"
44
45 time2 = Time.now
46 puts time2
47
48 puts time2 - time1
```

**Listing 6.7:** Test 2 implementation using DEEP SEMANTICS.

```
1  require File.join(File.dirname(__FILE__), 'active_semantics.rb')
2  require File.join(File.dirname(__FILE__), 'Helper/namespaces.rb')
3
4  time1 = Time.now
5  puts time1
6
7  Namespaces.add_namespace('http://www.ontoverse.org/ontologies/2008/3/iken2.owl#', 'iken:')
8
9  $active_semantics = ActiveSemantics.instance
10 iken = $active_semantics.set_director({'adapter' => 'FileAdapter', 'ontology_source' => '
     iken_infered3.nt', 'builder' => 'DeepIntegrationBuilderOWLLite'})
11
12 timeA = Time.now
13 puts "Find all instances by their RDFS labels in IKen ontology - start: #{timeA}"
14 test_search_labels = ["bedeckt", "wolkenlos", "Locken", "Pony", "Grinsen", "Lachen", "Juli", "
     Oktober", "Natur", "Sport"]
15
16 test_search_labels.each do |search_label|
17   instance_hits = Array.new
18   instance_hits = iken.find_instances_by_label(search_label)
```

```
19
20   if instance_hits.size > 0
21     instance_hits.each do |instance_hit|
22       puts instance_hit.local_name
23     end
24   end
25 end
26 timeB = Time.now
27 puts "Find all instances by their RDFS labels in IKen ontology - end: #{timeB}"
28 puts "Required time: #{(timeB - timeA)}"
29
30 time2 = Time.now
31 puts time2
32
33 puts time2 - time1
```