



Semantic Repository for RDF(S) and OWL

**SwiftOWLIM/BigOWLIM**

versions 2.9÷3.X, 25 July 2009

# ***OWLIM Primer***

# Table of contents

- 1 Foreword ..... 1**
- 2 About This Document ..... 2**
  - 2.1 OWLIM User Documentation – Overview ..... 2
  - 2.2 Purpose, Intended Readership and Overview, of This Document..... 3
  - 2.3 How to Use This Document? ..... 4
  - 2.4 Credits and Licensing ..... 4
- 3 Background Knowledge ..... 6**
  - 3.1 Introduction to Semantic Web Knowledge Management Concepts..... 6
    - 3.1.1 Resource Description Framework (RDF) ..... 6
    - 3.1.2 RDF Schema (RDFS) .....11
    - 3.1.3 Ontologies and Knowledge Bases .....14
    - 3.1.4 Logic, Inference, and Ontology Languages.....17
    - 3.1.5 Web Ontology Language (OWL) and Its Dialects.....19
    - 3.1.6 Querying Languages .....21
    - 3.1.7 Reasoning Strategies.....23
    - 3.1.8 Semantic Repositories .....24
  - 3.2 Introduction to Sesame..... 24
    - 3.2.1 Sesame Architecture .....25
    - 3.2.2 The SAIL API.....26
- 4 Introduction to OWLIM..... 28**
  - 4.1 Advantages of OWLIM ..... 28
  - 4.2 Limitations of OWLIM..... 28
  - 4.3 OWLIM Interoperability and Architecture ..... 29
    - 4.3.1 Integration with Sesame.....30
    - 4.3.2 The TRREE Engine .....30
  - 4.4 OWLIM Editions..... 32
    - 4.4.1 SwiftOWLIM and BigOWLIM.....32
    - 4.4.2 Versioning of OWLIM .....33
  - 4.5 Supported Semantics ..... 33
    - 4.5.1 Pre-Defined Rule Sets .....34
    - 4.5.2 OWL Compliance .....34
    - 4.5.3 PROTON Primitives .....35
    - 4.5.4 Custom Rule-Sets .....35
- 5 Installation and Configuration Overview..... 37**
  - 5.1 Contents of the Distribution Package ..... 37
  - 5.2 How to Get Started Quickly ..... 38

5.3 Creating Custom Configurations .....	38
<b>6 Glossary of Terms.....</b>	<b>39</b>
<b>7 References.....</b>	<b>40</b>

## 1 Foreword

This book is the introductory document of the OWLIM's set of user documents. OWLIM is the Ontotext's database system for storing and making requests against structured data. It is packaged as a Storage and Inference Layer (SAIL) for the [Sesame](#) RDF framework. OWLIM is based on [TRIPLE](#) – a native RDF database and rule-entailment engine. If you need a quick overview of OWLIM or a download link to its latest releases, please visit OWLIM's product page at <http://www.ontotext.com/owlim/index.html>.

At present, OWLIM has become the undisputed leader among structured data repositories for its unsurpassed processing capacity (currently it supports non-trivial OWL inference against 3 Billion triples and successful loading of 12 billion triples!) and speed (like loading and materializing LUBM(50) within 42 sec. on a desktop machine, delivering unmatched throughput of 161 KSt./sec).

*{няколко думи за историята на OWLIM тук}*

The purpose of this book is to provide you with a conceptual overview of OWLIM – that is, with some general information about its structure and functioning – as well as with the instructions needed to install the system. In addition to that, a good portion of this document is geared towards introducing the basics of the Semantic Web<sup>1</sup>, the knowledge of which is an indispensable prerequisite for the successful adoption of OWLIM (or any other structured data repository, for that matter).

---

<sup>1</sup> For a detailed explanation of the Semantic Web, refer to <http://www.w3.org/2001/sw/>

## 2 About This Document

This chapter briefly explains the purpose and the scope of this document, its place in the OWLIM user documentation set, and provides some suggestions for efficient use.

### 2.1 OWLIM User Documentation – Overview

OWLIM comes in two editions – SwiftOWLIM and BigOWLIM, and the user documentation set for each of them comprises several documents. Except for the current document, which is common for both editions, all other documents are edition-specific. (See Table 1 - The OWLIM User Documentation Set.)









<i>OWLIM Editions</i> ⇒	<u>Swift-OWLIM</u> ↓	<u>Big-OWLIM</u> ↓	<u>Document Type</u>	<u>Document Description</u>
<b>Document Titles</b> ↓ 	 OWLIM Primer		tutorial	This document. See “ <b>Purpose, Intended Readership and Overview, of This Document</b> ”.
	 SwiftOWLIM Fact Sheet	 BigOWLIM Fact Sheet	reference	An excerpt from the User Guide for the respective OWLIM edition with the following factual information: succinct product description and installation instructions, licensing, release notes, requirements, supported standards, and download and support links.
	 SwiftOWLIM User Guide	 BigOWLIM User Guide	manual	Provides in-depth explanation of the respective OWLIM edition from the user perspective. Features detailed installation, configuration, customization and usage instructions.
	 SwiftOWLIM Tests and Benchmarks	 BigOWLIM Tests and Benchmarks	reference	Describes the tests with which the respective OWLIM edition is being distributed.

Table 1 - The OWLIM User Documentation Set

In this document, the SwiftOWLIM User Guide and the BigOWLIM User Guide are referred to collectively as "OWLIM User Guides" in cases where it is not necessary to distinguish between the editions of OWLIM.

## 2.2 Purpose, Intended Readership and Overview, of This Document

This document is designed for database specialists who need to implement the OWLIM semantic database in an information retrieval system or to use OWLIM as a stand-alone application for storing and making requests against structured data. The document is also useful for system administrators who need to support and maintain an OWLIM-based system but have no prior knowledge of OWLIM.

This document presumes that the reader is a database specialist but not an expert in semantic database systems, semantic information retrieval, or Semantic Web. He is expected to be able to draw from his knowledge of relational databases; the required minimum of Semantic Web concepts and related information is provided in the course of this book.

Understanding of XML would be beneficial as well, as it is the most widely used metadata description language of the Web today and, for this reason, standardly used as a tool for illustrating the Semantic Web concepts in the referencing literature, this book included.

The purpose of this document is three-fold:

1. To deliver a general overview of the OWLIM system and to make you acquainted with its architecture and operational logic.
2. To introduce the Semantic Web and its concepts and to provide enough usable information so that you could start using your OWLIM system practically unimpeded in case your Semantic Web knowledge is currently limited.
3. To provide you with instructions on how to install OWLIM.

The first part of this document, "Background Knowledge," provides, in succinct form, the general conceptual information about the Semantic Web. Because there is a great deal of information about the Semantic Web in existence both over the Internet and in print, the topics in this section basically serve to check the required knowledge rather than go into details about the concepts being explained. This is by necessity so because an in-depth discussion on these topics goes beyond the scope of this product-related user document. However, on every subject, ample references are provided, so you could explore a subject further if you feel the need to do so.

The second and the third parts deal with OWLIM itself. The second part, "Introduction to OWLIM" gives you the basic information about OWLIM. It creates the necessary foundation for your OWLIM knowledge so you could later get down to the specifics in the User Guide of your chosen OWLIM edition and do your actual work with it.

Once you know what OWLIM is about and how it operates, part three, "Installation and Configuration Overview" explains the installation package and procedure and shows you how to customize your OWLIM configuration. Though this part already deals with OWLIM's specifics, it is included here and not in the User Guides because the installation procedure and configuration options are common for all editions of OWLIM.

## 2.3 How to Use This Document?

Reading the first part, “Background Knowledge” is optional depending on how familiar the Semantic Web concepts are to you. If you are new to the Semantic Web, it is recommended to read this entire section first and preferably check as many of the references as your time permits. If you have a certain level of knowledge however, you might want to read the part selectively, just perusing what you already know and paying more attention to the content that is new to you. And, of course, if you consider yourself knowledgeable about the Semantic Web, you can skip the entire first part.

The second part, “Introduction to OWLIM” is a real “must-read”: it should be read wall-to-wall (all of it) and sequentially (from the beginning to the end) without skipping any topics. You are expected to cover this part before attempting to actually do anything with OWLIM.

Reading the third part, “Installation and Configuration Overview”, becomes necessary when you decide to actually install OWLIM on your system, but reading it anyway can be a useful addition to your OWLIM education.

The following formatting conventions are used in this book:

- Code examples are listed in **typewriter-like font**.
- The first occurrence of a term that is explained in the Glossary at the end of this book is hyperlinked to its Glossary entry and is formatted [like this](#).
- Other important terms, when first introduced, may be written in *italics*.
- Formulas, too, are always in *italics*.
- References to the list of the referred literature are given in square brackets, e.g. [3] means “Refer to document #3 in the References section”.

## 2.4 Credits and Licensing

OWLIM benefits from the scalable architecture and numerous “basic” components of Sesame.

The development of OWLIM is partly supported by [SEKT](#), [TAO](#), [TripCom](#), [LarkC](#), and other [FP6](#) and [FP7](#) European research [projects](#).

The this document products discussed in it are copyrighted and subject to licensing as follows:

- © **Copyright 2005-2009, Ontotext Lab, Sirma Group Corp.**  
135 Tsarigradsko Shosse, Sofia 1784, Bulgaria, <http://www.ontotext.com>.
- **SwiftOWLIM** is a free software, available under the [GNU Lesser General Public License](#) (LGPL) version 2.1. One can redistribute and/or modify it freely under the terms of this license.
- Licensing of the third-party libraries:
  - **Sesame**, © Copyright Aduna b.v. Sesame, is an open-source library, available under the [LGPL](#).
  - **TRREE**, © Copyright Ontotext Lab, Sirma Group Corp., is proprietary software owned by Ontotext Lab. SwiftTRREE v2.9.1 is licensed for use free of charge as an integral part of SwiftOWLIM v2.9.1. Re-distribution of TRREE in any form, except as part of the original SwiftOWLIM distribution package, is strictly prohibited. Any form of modification or reverse-engineering of TRREE is prohibited. SwiftTRREE is distributed

together with SwiftOWLIM without warranty of any kind including, but not limited to suitability for any particular purpose.

- All other trademarks mentioned in this document, if any, belong to their respective owners.

Full licensing information is available at <http://www.ontotext.com/owlim/licence.html>, as well as in the licence.txt file in the main folder in the distribution package.



## 3 Background Knowledge

The background knowledge you need to be able to use OWLIM comprises some basic [Semantic Web](#) concepts and general understanding of the Sesame framework. This chapter provides introductions to both.

### 3.1 Introduction to Semantic Web Knowledge Management Concepts

The Semantic Web is about presenting web-based data in machine-readable form so it could be further processed and eventually presented to the user as some relevant and useful information with as little human intervention in the process as possible. Retrieved in that way, this information content may even include some knowledge that has not been explicitly available prior to that processing.

The ambition of the Semantic Web is to solve the most problematic issues that come with the spread of the “standard” non-semantic (HTML-based or similar) Web and which result in high level of human effort involved when it comes to finding and retrieving precise information in usable form. For example, automated searches nowadays while being quite fast still have a lot to want in terms of relevance and accuracy: of the thousands of matches typically returned, only a few point to truly relevant content and some of this content may be buried deeply in the subsequent pages of the results. All these issues highly reduce the value of the information received and of automation as a means to obtain it. There are also problems related to synonymy and homonymy, the lack of unification of the concepts, and many others.

The Semantic Web resolves this issues by adopting unique identifiers for the pieces of information that are to be retrieved, as well as for the relationships between those pieces where such relationships exist. These identifiers, called “[Universal Resource Identifiers](#)” (URIs) (a “resource” is any retrievable piece of information, including the relationships these pieces have to each other) are similar to Web pages URLs but do not necessarily have a web page or any kind of content: their sole purpose is to uniquely identify an object and its relationships.

Having taken down, through the use of URIs, the ambiguity of the retrievable data, the Semantic Web takes a step further and relates the individual pieces of data to the conceptual categories they belong thus making it possible to turn the existing disarray of information into truly usable knowledge. This is achieved by [ontologies](#) – hierarchical structures of concepts– and by bonding the individual pieces of data to these concepts.

#### 3.1.1 Resource Description Framework (RDF)

Notwithstanding its rapid growth and development, the World-Wide Web today is still largely burdened by its original sin – its content cannot be interpreted by machines as far as meaning and semantics are concerned. Machines cannot understand meaning, therefore they cannot understand web content. For this reason, most attempts to retrieve some useful pieces of information from the Web require a high degree of user involvement – manually retrieving information from multiple sources (different Web pages), “digging” through multiple search engine results (where the useful pieces of data are often buried many pages deep), comparing differently structured results sets of data (most of them incomplete), and so on.

For the machine interpretation of semantic content to become possible, there are two prerequisites:

1. Every piece of useful information needs to be uniquely identified. (For example, if one and the same person, named, say John Doe, owns a web site, authors articles on other sites, gives an interview on another site, and, in addition to that, has profiles in a couple of social media like Facebook and LinkedIn, the occurrences of his name in all those places should be related to one and the same unique identifier.)
2. There must be a unified system of conveying and interpreting meaning that all automated search agents and data storage applications could use.

One reasonable approach to this which, in the recent years, has become the backbone of the Semantic Web as it currently is, is to embed the necessary machine-processable information into the web content itself through the use of special meta-descriptors (meta-tagging) in addition to the existing meta-tags that concern mainly the layout.

Within these meta tags, the [resources](#) (the pieces of useful information) can be uniquely identified in the same manner in which Web pages are uniquely identified (by extending the existing URL system into something more universal – a URI (Uniform Resource Identifier) system, just like in algebra complex numbers extend the real numbers system). In addition to that, some simple programming-logic-like conventions can be devised and agreed upon, so that resources can be described in terms of properties and values (resources can have properties and properties can have values). The concrete implementations of these conventions can be then embedded into the web pages (through meta-descriptors again) thus effectively “telling” the processing machines things like “resource ‘John Doe’ has a [property] ‘web site’” and “this particular property ‘web site’ has the value of ‘www.johndoesite.com’.”

More precisely, there needs to be a system of machine-processable identifiers for identifying these concrete implementations (called “[statements](#)”) and the individual elements of which the statements are built (subject, predicate, or object) without any possibility of confusion with a similar-looking identifier that might be used by someone else on the Web.

The [Resource Description Framework](#) (RDF) developed by the World Wide Web Consortium (W3C) makes possible the automated semantic processing of information as outlined above. Although frequently referred to as a “language”, RDF is mainly a data model. It is based on the idea that the things being described have properties which have values, and that resources can be described by making statements. RDF prescribes how to make statements about resources, in particular, Web resources, in the form of subject-predicate-object expressions. The examples above (“resource ‘John Doe’ has a [property] ‘web site’” and “this particular property ‘web site’ has the value of ‘www.johndoesite.com’”) are precisely this kind of statements. Because these expressions always follow the triple subject-predicate-object structure, they are known as “[triples](#)” in RDF terminology. (“Resource ‘John Doe’ [= subject] has [= predicate] a ‘web site’ [=object].”) [32]

The basic RDF concepts include Uniform Resource Identifiers, statements, and properties. They are discussed in the respective topics that follow.

### 3.1.1.1 Uniform Resource Identifiers (URIs)

If we can think of a resource as an object, a “thing” we want to talk about, then the Uniform Resource Identifier (URI) is its “ID card” which uniquely identifies it. Resources can be authors, books, publishers, places, people, hotels, goods, articles search queries, and so on. In the Semantic Web, every resource has a URI. A URI can be a URL or some other kind of unique identifier. Unlike URLs, URIs do not necessarily enable access to the resource they describe, that is, in most cases they do not represent actual web pages. For example, the string “http://www.johndoesite.com/aboutme.htm” if used as a URL (Web link) is expected to take us to a Web page of the site providing information about

the site owner, the person John Doe; the same string however can serve as the URI uniquely identifying that person on the Web irrespective of whether such a page exists or not.

Thus URI schemes can be defined not only for Web locations but also for such diverse objects as telephone numbers, ISBN numbers, and geographic locations. There has been a long discussion about the nature of URIs (see [40]), but we will not go into detail here. In general, we assume that a URI is the identifier of a Web resource and, as such, can be used as either the subject or the object of a statement. Once the subject is assigned a URI, it can be treated as a resource and further statements can be made about it.

This idea of using URIs to identify “things” and the relations between them is quite important. This choice gives us in one stroke global, worldwide, unique naming schemes. The use of such a scheme greatly reduces the homonym problem that has plagued distributed data representation until now.

### 3.1.1.2 Statements – Object-Attribute-Value Triples

If we want to make the information in following sentence

“The web site `www.johndoesite.com` is created by John Doe.”

machine-accessible on the Semantic Web, the first thing we need to do is to express it the form of an RDF statement, that is, an object-attribute-value triple:

“The web site `www.johndoesite.com` has a creator whose name is John Doe.”

Revised in this manner, the statement emphasizes some of its parts to illustrate that, in order to describe something, there need to be ways to name, or identify, a number of things:

- the thing the statement describes (Web site “`www.johndoesite.com`”, in this case)
- a specific property (“creator”, in this case) of the thing the statement describes
- the thing the statement says is the value of this property (who the owner is), for the thing the statement describes

The respective RDF terms for the various parts of the statement are:

- the subject is the URL “`www.johndoesite.com`”
- the predicate is the expression “has creator”
- the object is name of the owner
- the value of the object is the phrase “John Doe”

Next, each member of the subject-predicate-object triple should be presented through its URIs, for example:

- the subject is “`http://www.johndoesite.com`”
- the predicate is “`http://purl.org/dc/elements/1.1/creator`” (this is according to a particular [RDF Schema](#) (see “RDF Schema (RDFS)” on page 11), namely, the Dublin Core Metadata Element Set; refer to “Sharing Vocabularies” (page 13) and “Dublin Core Metadata Initiative” (page 13) for details)
- the object is “`http://www.johndoesite.com/aboutme`” (may not exist as an actual web page)

Note that in this version of the statement, instead of identifying the creator of the web site by the character string “John Doe”, we used a URI, namely “`http://www.johndoesite.com/aboutme`”. An advantage of using a URI in this case is that the identification of the statement’s subject can be more

precise. That is, the creator of the page is neither the character string "John Doe", nor any one of the thousands of people with that name, but the particular John Doe associated with that URI (whoever created the URI defines the association). Moreover, since there is a URI to refer to John Doe, he is now a full-fledged resource and additional information can be recorded about him, simply by adding additional RDF statements with John's URI as the subject.

What we basically have now is a the logical formula  $P(x, y)$ , where the binary predicate  $P$  relates the object  $x$  to the object  $y$  and we may think of this formula as written in the form  $(x, P, y)$ <sup>1</sup>. Therefore, we can describe the statement as

```
<http://www.johndoesite.com> <http://purl.org/dc/elements/1.1/creator>
    <http://www.johndoesite.com/aboutme>
```

Now this is quite a clumsy notation, so there is a convention in RDF for an abbreviated way of describing statements. It uses a shorthand way of writing triples (the same shorthand is also used in the RDF specifications). This shorthand employs an XML *qualified name* (or *QName*) without angle brackets as an abbreviation for a full URI reference. A QName contains a prefix that has been assigned to a namespace URI, followed by a colon, and then a local name. The full URI reference is formed from the QName by appending the local name to the namespace URI assigned to the prefix. So, for example, if the QName prefix "foo" is assigned to the namespace URI "http://example.com/somewhere/", then the QName "foo:bar" is shorthand for the URI "http://example.com/somewhere/bar".

In our example, we can define the namespace "jds" for "http://www.johndoesite.com" and use the existing in the Dublin Core Metadata namespace "dc" for "http://purl.org/dc/elements/1.1/." So the shorthand form of the statement is going to be:

```
jds: dc:creator jds:aboutme
```

The triple nature of the RDF statements naturally leads to representing them as graphs. (The [RDF's graph model](#) is defined in [34].) In this notation, a statement is represented by:

- a node for the subject
- a node for the object
- an arc for the predicate, directed from the subject node to the object node.

So the RDF statement above would be represented by the graph shown in Figure 1:

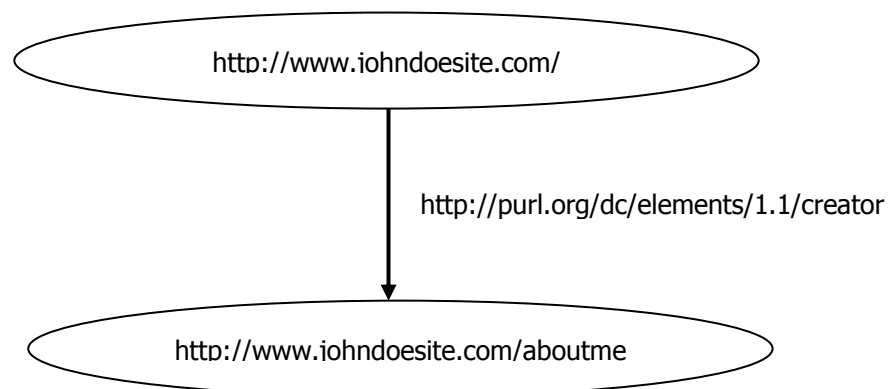


Figure 1 - Graphic representation of a triple

---

<sup>1</sup> In fact, RDF offers only binary predicates (properties). If more complex relationships are to be defined, this is done through sets of multiple RDF triples.

This kind of graph is known in the artificial intelligence community as a “semantic net”. [32]

In order to represent RDF statements in a machine-processable way, RDF uses mark-up languages, namely (and almost exclusively) the Extensible Mark-up Language (XML)<sup>1</sup>. XML was designed to allow anyone to design their own document format and then write a document in that format. RDF defines a specific XML mark-up language, referred to as RDF/XML, for use in representing RDF information and for exchanging it between machines. Written in RDF/XML, our example will look as follows:

```
<?xml version="1.0" encoding="UTF-16"?>

<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:jds="http://www.johndoesite.com/">

  <rdf:Description rdf:about="http://www.johndoesite.com/">
    <dc:creator rdf:resource="jds:aboutme">
  </rdf:Description>

</rdf:RDF>
```

Note that RDF/XML uses the namespace mechanism of XML, but in an expanded way. In XML, namespaces are only used for disambiguation purposes. In RDF/XML, external namespaces are expected to be RDF documents defining resources, which are then used in the importing RDF document. This mechanism allows the reuse of resources by other people who may decide to insert additional features into these resources. The result is the emergence of large, distributed collections of knowledge.

Also observe that the `rdf:about` attribute of the element `rdf:Description` is, strictly speaking, equivalent in meaning to that of an ID attribute, but it is often used to suggest that the object about which a statement is made has already been “defined” elsewhere.<sup>2</sup>

Of course, there is much more to RDF/XML logic and syntax than we could possibly cover here. For a discussion of the principles behind the modelling of RDF statements in XML (known as “stripping”), together with a presentation of the available RDF/XML abbreviations and other details and examples about writing RDF in XML, see the (normative) RDF/XML Syntax Specification by the W3C, [33].

### 3.1.1.3 Properties

Properties are a special kind of resources: they describe relations between resources, for example “written by”, “age”, “title”, and so on. Properties in RDF are also identified by URIs (in most cases, these are actual URLs). Therefore, properties themselves can be used as the object in an object-attribute-value triple (statement). This possibility is rather unusual for modelling languages and therefore potentially confusing for modellers, but it offers great flexibility.

---

<sup>1</sup> Because an abstract data model needs a concrete syntax in order to be represented and transmitted, RDF has been given a syntax in XML. As a result, it inherits the benefits associated with XML. However, it is important to understand that other syntactic representations of RDF, not based on XML, are also possible; XML-based syntax is not a necessary component of the RDF model.

<sup>2</sup> Strictly speaking, a set of RDF statements together simply forms a large graph, relating things to other things through properties, and there is no such concept as “defining” an object in one place and referring to it elsewhere. Nevertheless, in the serialized XML syntax, it is sometimes useful (if only for human readability) to suggest that one location in the XML serialization is the “defining” location, while other locations state “additional” properties about an object that has been “defined” elsewhere.

### 3.1.1.4 Named Graphs

Basically, a [named graph](#) (NG) is a set of triples named by an URI. This URI can then be used outside or within the graph to refer to it, [26]. Named Graphs reflect the idea that having multiple RDF graphs in a single document or repository and naming them with URIs provides some further useful functionality in addition to that of the standard RDF triples.

Named graphs represent an extension of the RDF data model, where quadruples  $\langle s, p, o, ng \rangle$  are used to define RDF multi-graph. This mechanism allows for handling provenance when multiple RDF graphs are integrated in a single repository. For information on the semantics and the abstract syntax of named graphs, refer to [6].

From the perspective of OWLIM, named graphs are important because the comprehensive support for [SPARQL](#) (the most popular RDF query language and current W3C recommendation – see “SPARQL” on page 22) requires NG support.

### 3.1.2 RDF Schema (RDFS)

While being a universal model that lets users describe resources using their own vocabularies, RDF does not make assumptions about any particular application domain, nor does it define the semantics of any domain. Is it up to the user to do so in RDF Schema (RDFS).

*RDF Schema* is a vocabulary description language for describing properties and classes of RDF resources, with a semantics for generalization hierarchies of such properties and classes.<sup>1</sup> Thus RDFS makes semantic information machine-accessible, in accordance with the Semantic Web vision. RDF Schema is a primitive ontology language. It offers certain modelling primitives with fixed meaning.

RDF Schema does not provide a vocabulary of application-specific classes. Instead, it provides the facilities needed to describe such classes and properties, and to indicate which classes and properties are expected to be used together (for example, to say that the property “JobTitle” will be used in describing a class “Person”). In other words, RDF Schema provides a type system for RDF.

The RDF Schema type system is similar in some respects to the type systems of object-oriented programming languages such as Java. For example, RDFS allows resources to be defined as instances of one or more classes. In addition, it allows classes to be organized in a hierarchical fashion; for example a class “Dog” might be defined as a subclass of “Mammal” which itself is a subclass of “Animal”, meaning that any resource which is in class “Dog” is also implicitly in class “Animal” as well.

RDF classes and properties, however, are in some respects very different from programming language types. RDF class and property descriptions do not create a straightjacket into which information must be forced, but instead provide additional information about the RDF resources they describe.

The RDFS facilities are themselves provided in the form of an RDF vocabulary; that is, as a specialized set of predefined RDF resources with their own special meanings. The resources in the RDFS vocabulary have URIs with the prefix <http://www.w3.org/2000/01/rdf-schema#> (conventionally associated with the QName prefix `rdfs`). Vocabulary descriptions (schemas) written in the RDFS language are legal RDF graphs. Hence, RDF software that is not written to also process the additional RDFS vocabulary can still interpret a schema as a legal RDF graph consisting of various resources and properties, but will not “understand” the additional built-in meanings of the RDFS terms. To

---

<sup>1</sup> Please be aware of the fact that that the RDF Schema is conceptually different from the XML Schema even though the common term “schema” suggests similarity. XML Schema constrains the structure of XML documents, whereas RDF Schema defines the vocabulary used in RDF data models.

understand these additional meanings, RDF software must be written to process an extended language that includes not only `rdf:vocabulary`, but also `rdfs:vocabulary`, together with their built-in meanings.

### 3.1.2.1 Describing Classes

A class can be thought of as a set of elements. Individual objects that belong to a class are referred to as instances of that class. A class in RDFS corresponds to the generic concept of a type or category, somewhat like the notion of a class in object-oriented programming languages such as Java. RDF classes can be used to represent almost any category of thing, such as web pages, people, document types, databases, or abstract concepts. Classes are described using the RDF Schema resources `rdfs:Class` and `rdfs:Resource`, and the properties `rdf:type` and `rdfs:subClassOf`. The relationship between instances and classes in RDF is defined using `rdf:type`.

An important use of classes is to impose restrictions on what can be stated in an RDF document using the schema. In programming languages, typing is used to prevent nonsense from being written (such as  $A+1$ , where  $A$  is an array, so we explicitly state that the arguments of  $+$  must be numbers). The same is needed in RDF: imposing a restriction on the objects to which the property can be applied. In mathematical terms, this is a restriction of the domain of the property.

### 3.1.2.2 Describing Properties

In addition to describing the specific classes of things they want to describe, user communities also need to be able to describe specific properties that characterize those classes of things (such as `NumberOfBedrooms` to describe an apartment). In RDFS, properties are described using the RDF class `rdf:Property`, and the RDFS properties `rdfs:domain`, `rdfs:range`, and `rdfs:subPropertyOf`.

All properties in RDF are described as instances of class `rdf:Property`. So a new property, such as `externs:weightInKg`, is described by assigning the property a `URIref`, and describing that resource with an `rdf:type` property whose value is the resource `rdf:Property`, for example, by writing the RDF statement:

```
externs:weightInKg    rdf:type    rdf:Property .
```

RDFS also provides vocabulary for describing how properties and classes are intended to be used together in RDF data. The most important information of this kind is supplied by using the RDFS properties `rdfs:range` and `rdfs:domain` to further describe application-specific properties.

The `rdfs:range` property is used to indicate that the values of a particular property are instances of a designated class. For example, if a site located on domain "example.com" and, for this reason using the prefix "ex" as a shortcut for "http://www.example.com", wanted to indicate that the property `ex:author` had values that are instances of class `ex:Person`, it would write the RDF statements:

```
ex:Person    rdf:type    rdfs:Class .
ex:author    rdf:type    rdf:Property .
ex:author    rdfs:range  ex:Person .
```

These statements indicate that `ex:Person` is a class, `ex:author` is a property, and that RDF statements using the `ex:author` property have instances of `ex:Person` as objects.

The `rdfs:domain` property is used to indicate that a particular property applies to a designated class. For instance, if example.com wanted to indicate that the property `ex:author` applies to instances of class `ex:Book`, it would write the RDF statements:

```

ex:Book      rdf:type      rdfs:Class .
ex:author    rdf:type      rdf:Property .
ex:author    rdfs:domain   ex:Book .
  
```

These statements indicate that `ex:Book` is a class, `ex:author` is a property, and that RDF statements using the `ex:author` property have instances of `ex:Book` as subjects. [32]

### 3.1.2.3 Sharing Vocabularies

Just because RDFS provides you with the means of creating your own vocabularies, this does not mean that you necessarily have to do so. In most cases, it is much better and easier to use an existing vocabulary created by someone else who has already been describing a conceptual domain same or similar as your own. Such publicly available vocabularies, called “shared vocabularies” are not only cost-efficient to use, but they also reflect and promote the shared understanding of the described concepts.

In the example used earlier, in the triple

```

jds:      dc:creator      jds:aboutme .
  
```

the predicate `dc:creator`, when fully expanded as a URI, is an unambiguous reference to the *creator* attribute in the Dublin Core metadata attribute set (discussed further in the Dublin Core Metadata Initiative topic), a widely-used set of attributes (properties) for describing information of this kind. So this triple is effectively saying that the relationship between the web site (identified by `http://www.johndoesite.com/`) and the creator of the site (a distinct person, identified by `http://www.johndoesite.com/aboutme.htm`) is exactly the concept identified by `http://purl.org/dc/elements/1.1/creator`. This way, anyone familiar with the Dublin Core vocabulary or those who find out what `dc:creator` means (say, by looking up its definition on the Web) will know what is meant by this relationship. In addition, based on this understanding, people can write programs to behave in accordance with that meaning when processing triples containing the predicate `dc:creator`.

Of course, this depends on increasing the general use of URIs to refer to things instead of using literals (i.e., using URIs like `jds:aboutme` and `dc:creator` instead of character string literals like “John Doe” and “creator”). Even then, RDF’s use of URIs does not solve all identification problems because, for example, people can still use different URIrefs to refer to the same thing. For this reason, it is a good idea to have a preference towards using terms from existing vocabularies (such as the Dublin Core) where possible, rather than making up new terms that might overlap with those of some other vocabulary. Appropriate vocabularies for use in specific application areas are being developed all the time, however, even when synonyms are created this way, the fact that these different URIs are used in the commonly-accessible “Web space” provides the opportunity both to identify equivalences among these different references and to migrate toward the use of common references.

### 3.1.2.4 Dublin Core Metadata Initiative

An example of a shared vocabulary that is readily available for reuse is The Dublin Core. The Dublin Core is a set of “elements” (properties) for describing documents (and hence, for recording metadata). The element set was originally developed at the March 1995 Metadata Workshop in Dublin, Ohio, USA. The Dublin Core has subsequently been modified on the basis of later Dublin Core Metadata workshops and is currently maintained by the [Dublin Core Metadata Initiative](#).

The goal of the Dublin Core is to provide a minimal set of descriptive elements that facilitate the description and the automated indexing of document-like networked objects, in a manner similar to a



library card catalogue. The Dublin Core metadata set is suitable for use by resource discovery tools on the Internet, such as the web crawlers employed by search engines. In addition, the Dublin Core is meant to be sufficiently simple to be understood and used by the wide range of authors and casual publishers who contribute information to the Internet.

Dublin Core elements have become widely used in documenting Internet resources (the Dublin Core creator element has already been used in the earlier examples). The current elements of the Dublin Core are defined in [10] and contain definitions for the following properties like "Title" ("A name given to the resource."), "Creator" ("An entity primarily responsible for making the content of the resource."), "Date" ("A date associated with an event in the life cycle of the resource."), and "Type" ("The nature or genre of the content of the resource.").

Information using the Dublin Core elements may be represented in any suitable language (e.g., in HTML meta elements). However, RDF is an ideal representation for Dublin Core information. The following example taken from [19] uses Dublin Core by itself to describe an audio recording of a guide to growing rose bushes:

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:dc="http://purl.org/dc/elements/1.1/">

  <rdf:Description rdf:about="http://media.example.com/audio/guide.ra">

    <dc:creator>Rose Bush</dc:creator>
    <dc:title>A Guide to Growing Roses</dc:title>
    <dc:description>Describes process for planting and nurturing
different kinds of rose bushes.</dc:description>
    <dc:date>2001-01-20</dc:date>

  </rdf:Description>
</rdf:RDF>
```

### 3.1.3 Ontologies and Knowledge Bases

In general, an ontology describes formally a domain of related concepts. It consists of (usually finite) list of terms and the relationships between these terms. The terms denote important concepts (classes of objects) of the domain. For example, in a company setting, staff members, managers, company products, offices, and departments are some important concepts. The relationships typically include hierarchies of classes. A hierarchy specifies a class  $C$  to be a subclass of another class  $C'$  if every object in  $C$  is also included in  $C'$ . For example, all managers are staff members.

Apart from subclass relationships, ontologies may include information

- such as properties ( $X$  is subordinated  $Y$ )
- value restrictions (only managers may head departments)
- disjointness statements (managers and general employees are disjoint)
- specifications of logical relationships between objects (every department must have at least three staff members)

Ontologies are important because [semantic repositories](#) use ontologies as semantic schemata. This makes automated reasoning about the data possible (and easy to implement) since the most essential relationships between the concepts are built-in into the ontology. (For details on semantic repositories refer to the Semantic Repositories topic on page 24.)

Formal [knowledge representation](#) (KR) is about building models<sup>1</sup> of the world (of a particular state of affairs, situation, domain or problem), which allow for automatic reasoning and interpretation. Such formal models are achieved through the use of ontologies, whenever they (are intended to) represent a shared conceptualization (e.g. a basic theory, a schema, or a classification). Ontologies can be used to provide formal semantics (i.e. machine-interpretable meaning) to any sort of information: databases, catalogues, documents, web pages, etc. Ontologies can be used as semantic frameworks: the association of information with ontologies makes such information much more amenable to machine processing and interpretation. This is because formal ontologies are represented in logical formalisms, such as [OWL](#), [8], which allow automatic inferencing over them and over datasets aligned to them. An important role of ontologies is to serve as schemata or “intelligent” views over information resources<sup>2</sup>. Thus they can be used for indexing, querying, and reference purposes over non-ontological datasets and systems, such as databases, document and catalogue management systems. Because ontological languages have a formal semantics, ontologies allow a wider interpretation of data, i.e. inference of facts which are not explicitly stated. In this way, they can improve the interoperability and the efficiency of the usage of arbitrary datasets.

An ontology can be characterized as comprising a 4-tuple<sup>3</sup> is

$$O = \langle C, R, I, A \rangle$$

where

- $C$  is a set of classes representing *concepts* we wish to reason about in the given domain (invoices, payments, products, prices,...).
- $R$  is a set of relations (also referred to as properties or predicates) holding between (instances of) those classes (Product *hasPrice* Price).
- $I$  is a set of instances, where each instance can be an instance of one or more classes and can be linked to other instances or to literal values (strings, numbers, ...) by relations (product23 *compatibleWith* product348; product23 *hasPrice* €170).
- $A$  is a set of axioms (if a product has a price greater than €200, then shipping is free).

### 3.1.3.1 Classification of Ontologies

The ontologies can be classified as light-weight or heavy-weight according to the complexity of the KR language used. Light-weight ontologies allow for more efficient and scalable reasoning, but do not possess the high predictive (or restrictive) power of the full-bodied concept definitions of heavy-weight ontologies. The ontologies can be further differentiated according to the sort of conceptualization that they formalize: upper-level ontologies model general knowledge, while domain- and application ontologies represent knowledge about a specific domain (e.g. medicine or sport) or a type of applications (e.g. knowledge management systems). Basic definitions regarding ontologies can be found in [13], [14], [15], and [16].

---

<sup>1</sup> The typical modelling paradigm is mathematical logic, but there are also other approaches, rooted in information and library science. KR is a very broad term; here we only refer to one of its main streams.

<sup>2</sup> Comments in the same spirit are provided in [14] also. This is also the role of ontologies on the Semantic Web.

<sup>3</sup> A more formal and extensive mathematical definition of an ontology is given in [11]. The characterization offered here is suitable for the purposes of our discussion, however.

Finally, ontologies can be distinguished according to the sort of semantics being modelled and their intended usage. The major categories from this perspective are:

- Schema-ontologies: ontologies which are close in purpose and nature to database and object-oriented schemata. They define classes of objects, their appropriate attributes and relationships to objects of other classes. A typical usage of such an ontology is defining and managing of large sets of instances of the classes. Intuitively, a class in a schema ontology corresponds to a table in an RDBMS; a relation – to a column; an instance – to a row in the table for the corresponding class.
- Topic-ontologies: taxonomies which define hierarchies of topics, subjects, categories, or designators. These have a wide range of applications related to classification of different things (entities, information resources, files, web-pages, etc.) The most popular examples are library classification systems and taxonomies, which are widely used in the KM field. Yahoo and DMOZ<sup>1</sup> are popular large scale incarnations of this approach in the context of the Web. A number of the most popular taxonomies are listed as encoding schemata in Dublin Core [9].
- Lexical ontologies: lexicons with formal semantics, which define lexical concepts<sup>2</sup>, word-senses and terms. These can be considered as semantic thesaurus or dictionaries. The concepts defined in such ontologies are not instantiated, rather they are directly used as reference, e.g. for annotation of the corresponding terms in text. WordNet is the most popular general purpose (i.e. upper-level) lexical ontology.

### 3.1.3.2 Knowledge Bases

[Knowledge base](#) (KB) is a broader term than ontology. Similarly to an ontology, a KB is represented in a KR formalism, which allows automatic inference. It could include multiple axioms, definitions, rules, facts, statements, and any other primitives. In contrast to ontologies, however, KBs are not intended to represent a shared or consensual conceptualization. Thus, ontologies are a specific sort of KB. Many KBs can be split into ontology and instance data parts, in a way analogous to the splitting of schemata and concrete data in databases. A broader discussion on the different terms related to ontology and semantics can be found in [23].

#### 3.1.3.2.1 PROTON

PROTON (see [38]) is a light-weight upper-level schema-ontology developed in the scope of the SEKT project (see [35]). It is used in the KIM system, [30] for semantic annotation, indexing and retrieval. We will also use it for ontology-related examples within this section. PROTON is encoded in [OWL Lite](#) and defines about 300 classes and 100 properties, providing good coverage of named entity types and concrete domains (i.e. modelling of concepts such as people, organizations, locations, numbers, dates, addresses, etc.), [31]. A snapshot of the PROTON class hierarchy is given in Figure 2.

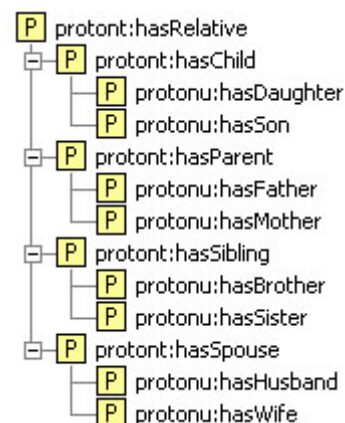


Figure 2 - A view of the top part of the PROTON class hierarchy

<sup>1</sup> <http://www.yahoo.com> and <http://www.dmoz.org> respectively.

<sup>2</sup> We use 'lexical concept' here as some kind of a formal representation of the meaning of a word or a phrase. In Wordnet, for example, lexical concepts are modelled as synsets (synonym sets), while word-sense is the relation between a word and a synset.

### 3.1.4 Logic, Inference, and Ontology Languages

The topics that follow take a closer look at the logic that underlies the retrieval and manipulation of semantic data and the kind of programming that supports it.

#### 3.1.4.1 Logical Programming

All that semantic structuring of data discussed in the previous topics would be meaningless if there weren't in existence a special kind of programs that are capable of reading and "understanding" such data and also of making deductions and inferences based on it. This kind of programs operate on the premises of logic and are a product of a specialized kind of programming called "[logical programming](#)" (LP).

In LP, a program consists of a collection of statements expressed as formulas in symbolic logic. (This is in sharp contrast to declarative programming where a program is a set of commands that need to be executed in a strict order.) There are rules of inference from logic that allow a new statement to be derived from old ones, with the guarantee that if the old statements are true, so is the new one.

Because these rules of inference can be expressed in purely symbolic terms, applying them is the kind of symbol manipulation that can be carried out by a computer. This is what happens when a computer executes a logical program: it uses the rules of inference to derive new statements from the ones given in the program, until it finds one that expresses the solution to the problem that has been formulated. If the statements in the program are true, then so are the statements that the machine derives from them, and the answers it gives will be correct.

The program can give correct answers only if the following two conditions are met:

1. The program must contain only true statements.
2. The program should contain enough statements to allow solutions to be derived for all the problems that are of interest.

There must also be a reasonable time frame for the entire inference process. To this end, the derivations the machine carries out should be fairly short, so that the machine could find answers quickly, and this may affect the form in which definitions are made and properties stated in the program. Nevertheless, each formula can be understood in isolation as a true statement about the problem to be solved.

#### 3.1.4.2 Predicate Logic

From a more abstract viewpoint, the subject of the previous topic is related to the foundation upon which logical programming resides, which is logic, particularly in the form of [predicate logic](#) (also known as "first order logic"). Some of the specific features of predicate logic render it very suitable for making inferences over the Semantic Web, namely:

- It provides a high-level language in which knowledge can be expressed in a transparent way and with a high expressive power.
- It has a well-understood formal semantics, which assigns an unambiguous meaning to logical statements.
- There exist proof systems that can automatically derive statements syntactically from a set of premises. Its proof systems are both sound (meaning that all derived statements follow semantically from the premises) and complete (all logical consequences of the premises can be derived in the proof system).

- It is possible to trace the proof that leads to a logical consequence. (This is because the proof system is sound and complete.) In this sense, the logic can provide explanations for answers.

The languages of RDF and OWL (Lite and DL) can be viewed as specializations of predicate logic. One reason for such specialized languages to exist is that they provide a syntax that fits well with the intended use (in our case, Web languages based on tags). The other major reason is that they define reasonable subsets of logic. This is important because there is a trade-off between the expressive power and the computational complexity of certain logics: the more expressive the language, the less efficient (in the worst case) the corresponding proof systems. As we stated, OWL Lite and [OWL DL](#) correspond roughly to a [description logic](#), a subset of predicate logic for which efficient proof systems exist.

Another subset of predicate logic with efficient proof systems comprises the so-called rule systems (also known as [Horn logic](#) or *definite logic programs*). A rule has the form

$$A_1, \dots, A_n \rightarrow B$$

where  $A_i$  and  $B$  are atomic formulas. In fact, there are two intuitive ways of reading such a rule:

- If  $A_1, \dots, A_n$  are known to be true, then  $B$  is also true. Rules with this interpretation are referred to as “deductive rules”.
- If the conditions  $A_1, \dots, A_n$  are true, then carry out the action  $B$ . Rules with this interpretation are referred to as “reactive rules”.

Both approaches have important applications. The deductive approach, however, is more relevant to the purpose of retrieving and managing structured data. This is because it relates better to the possible queries that one can ask, as well as to the appropriate answers and their proofs.

### 3.1.4.3 Description Logic

Description Logic (DL) historically evolved from a combination of frame-based systems and predicate logic. Its main purpose is to overcome some of the problems with frame-based systems and to provide a clean and efficient formalism to represent knowledge. The main idea of DL is to describe the world in terms of “properties” or “constraints” that specific “individuals” have to satisfy. DL is based on the following basic entities, [1]:

- **Objects** – Correspond to single “objects” of the real world such as a specific person, a table or a telephone. The main properties of an object are that it can be distinguished from other objects and that it can be referred to by a name. DL objects correspond to the individual constants in predicate logic.
- **Concepts** – Can be seen as “classes of objects”. Concepts have two functions: on one hand, they describe *a set of objects*, on the other hand they determine *properties* of objects. For example the class “table” is supposed to describe the set of all table (-objects) in this world. On the other hand it determines some properties of a table such as having four legs and a flat horizontal surface that you can lay something on. DL concepts correspond to one-place predicates in predicate logic and to classes in frame-based systems.
- **Roles** – Represent relationships between objects. For example the role “lays on” may determine the relationship between a book and a table, where the book lays on the table. Roles also can be applied to concepts. However they do not describe the relationship between the classes (concepts) but describe the properties of those objects that are members of that classes.

- **Rules** – In DL, rules take the form of “if condition  $x$  (left side), then property  $y$  (right side)” and form statements that read as “if an object satisfies the condition on the left side, then it has the properties of the right side”. So, for instance, a rule can state something like “if an object is male and has atleast one child, then it is a father”.

The family of description logic system consists of many members that in particular differ with respect to the constructs they provide. Not all of the constructs can be found in a single DL system and only some of them are applicable to the area of natural language processing. For a listing of some concrete constructs with a brief explanation of their semantics, refer to [1].

### 3.1.5 Web Ontology Language (OWL) and Its Dialects

In order to match the expectations for a useful heap of ontologies and structured metadata, the Semantic Web requires scalable high-performance storage and reasoning infrastructure. The major challenge towards building such an infrastructure is the expressivity of the underlying standards: RDF (see [24]), RDFS (see [3]) and OWL (see [8]). Even though RDFS can be considered a simple KR language, it is already a challenging task to implement a repository for it, which provides performance and scalability comparable to those of relational database management systems (RDBMS). Going up the stairs of the Semantic Web specifications stack, the challenges for the repository engineers are getting more and more serious. Even the simplest dialect of OWL (OWL Lite) is a description logic (DL) formalism with no algorithms enabling efficient inference and query answering over reasonably scaled KBs. Furthermore, the semantics of OWL Lite and DL are incompatible with that of RDF(S), see [12]. This causes lack of “backward compatibility”. Imagine the situation when an application, which uses RDFS schemata and RDFS-compliant repository, should be “upgraded” to OWL. The evolution should start with the replacement of the RDFS schemata with the OWL ontologies and adoption of a repository supporting (the corresponding part of) OWL. Even the most direct translation (re-labelling the `rdfs:Class`-es to `owl:Class`) of the schema, without adding further complexity, can lead to different inference and to inconsistencies.

Figure 3 presents a simplified map of the expressivity or complexity<sup>1</sup> of a number of OWL-related languages, as well as their bias towards description logic (DL) and Logical Programming (LP, see “Logical Programming” on page 17) based semantics. It presents the positioning of OWLIM’s reasoning capabilities (the owl-max rule-set), as it is discussed in Ref. An extended discussion on the topic can be found at: [http://www.ontotext.com/inference/rdfs\\_rules\\_owl.html](http://www.ontotext.com/inference/rdfs_rules_owl.html).

---

<sup>1</sup> The diagram provides a very rough idea about the expressivity of the languages, based on the complexity of entailment for them. Direct comparison between the different languages is impossible in many of the cases. For instance, Datalog is not simpler than OWL DL, it just allows for a different type of complexity.

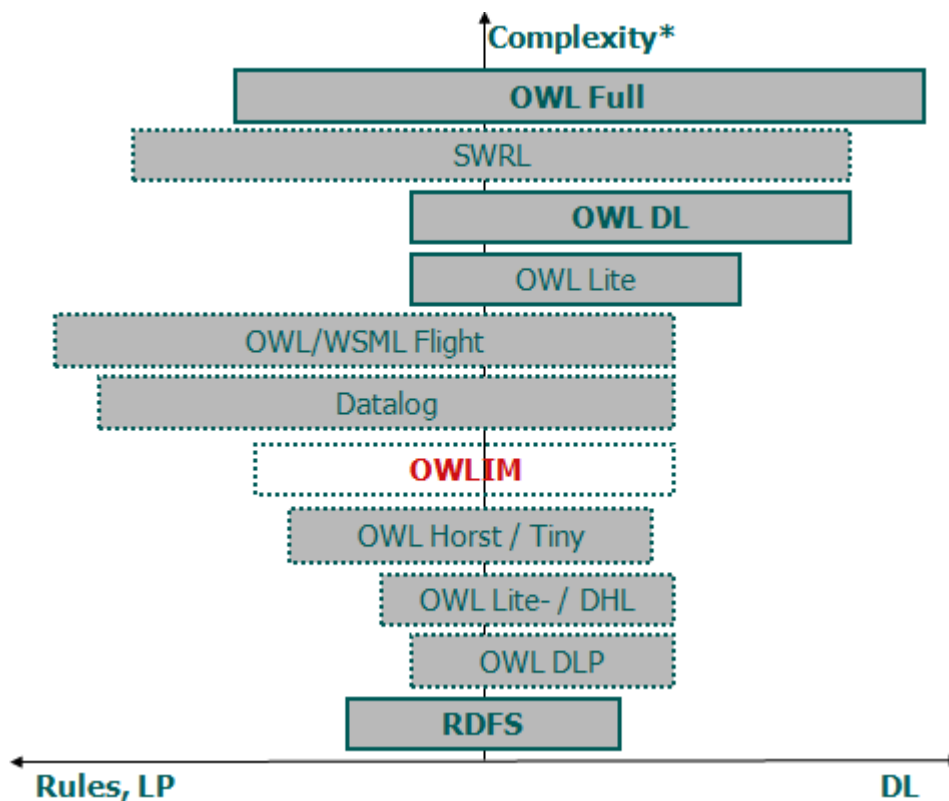


Figure 3 - OWL Layering Map

### 3.1.5.1 OWL DLP

[OWL DLP](#) is a non-standard dialect, offering a promising compromise between expressive power, efficient reasoning, and compatibility. It is defined in [12] as the intersection of the expressivity of OWL DL and LP. In fact, OWL DLP is defined as the most expressive sub-language of OWL DL, which can be mapped to [Datalog](#). OWL DLP is simpler than OWL Lite. The alignment of its semantics to the one of RDFS is easier, as compared to the Lite and DL dialects. Still, this can only be achieved through the enforcement of some additional modelling constraints and transformations. A broad collection of information related to OWL DLP can be found on [27].

Horn logic and description logics are orthogonal (in the sense that neither of them is a subset of the other). DLP is the “intersection” of Horn logic and OWL; it is the Horn-definable part of OWL, or stated another way, the OWL-definable part of Horn logic.

DLP has certain advantages:

- From modeller’s perspective, there is freedom to use either OWL or rules (and associated tools and methodologies) for modelling purposes, depending on the modeller’s experience and preferences.
- From implementation perspective, either description logic reasoners or deductive rule systems can be used. Thus it is possible to model using one framework, for instance, OWL, and to use a reasoning engine from the other framework, for instance, rules. This feature provides extra flexibility and ensures interoperability with a variety of tools.

Experience with using OWL has shown that existing ontologies frequently use very few constructs outside the DLP language.

### 3.1.5.2 OWL Horst

In [37] ter Horst defines RDFS extensions towards rule support and describes a fragment of OWL, more expressive than DLP. He introduces the notion of [R-entailment](#) of one (target) RDF graph from another (source) RDF graph on the basis of a set of entailment rules  $R$ . R-entailment is more general than the [D-entailment](#) used by Hayes, [18], in defining the standard RDFS semantics. Each rule has a set of premises, which conjunctively define the body of the rule. The premises are “extended” RDF statements, where variables can take any of the three positions.

The head of the rule comprises of one or more consequences, each of which is, again, an extended RDF statement. The consequences may not contain free variables, i.e. which are not used in the body of the rule. The consequences may contain blank nodes.

The extension of the R-entailment (as compared to the D-entailment) is that it “operates” on top of the so-called generalized RDF graphs, where blank nodes can appear as predicates. R-entailment rules without premises are used to declare axiomatic statements. Rules without consequences are used to imply inconsistency.

In this document we refer to this extension of RDFS as “[OWL Horst](#)”. As outlined in [37], this language has a number of important characteristics:

- It is a proper (backward-compatible) extension of RDFS. In contrast to OWL DLP, it puts no constraints on the RDFS semantics. The widely discussed meta-classes (classes as instances of other classes) are not disallowed in OWL Horst. It also does not enforce unique name assumption;
- Unlike the DL-based rule languages, like [SWRL](#), [20] and [25], R-entailment provides a formalism for rule extensions without DL-related constraints;
- Its complexity is lower than the one of SWRL and other approaches combining DL ontologies with rules; see section 5 of [37].

The “OWLIM” box covers the position on the map of the most complex OWL dialect supported by OWLIM – the `owl-max` rule-set with its `partialRDFS` parameter set to `false`; see the OWLIM User Guides for parameter description. The pre-defined rule sets in OWLIM do not support entailment of typed literals (D-entailment); more details on the semantics supported by OWLIM can be found in the Supported Semantics topic on page 33.

OWL Horst is close to what SWAD-Europe has intuitively described as [OWL Tiny](#), [36]. The major difference is that OWL Tiny (like the fragment supported by OWLIM) does not support entailment over data types.

### 3.1.6 Querying Languages

In this section, we introduce some query languages for RDF. This may beg the question as to why we need RDF-specific query languages at all instead of using an XML query language. The answer is that XML is located at a lower level of abstraction than RDF. This fact would lead to complications if we were querying RDF documents with an XML-based language. The RDF query languages explicitly capture the RDF semantics in the language design itself.

All the querying languages discussed below have an SQL-like syntax, but there are also a few non-SQL-like languages like Versa and Adenine.

The query languages supported by Sesame (which is the Java framework within which OWLIM operates) and, therefore, by OWLIM, are SPARQL and SeRQL (see “SPARQL” and “SeRQL” below).



### 3.1.6.1.1 RQL, RDQL

RQL (RDF Query Language) has been initially developed by the Institute of Computer Science at in Heraklion, Greece, in the context of the European IST project MESMUSES.3. RQL adopts the syntax of OQL (a query language standard for object-oriented databases), and, like OQL, is defined by means of a set of core queries, a set of basic filters, and a way to build new queries through functional composition and iterators.

The core queries are the basic building blocks of RQL, which give access to the RDFS-specific contents of an RDF triple store. RQL allows queries such as `Class` (retrieving all classes), `Property` (retrieving all properties) or `Employee` (returning all instances of the class with name `Employee`). This last query, of course, also returns all instances of subclasses of `Employee`, since these are also instances of the class `Employee`, by virtue of the semantics of RDFS.

RDQL (RDF Data Query Language) is a query language for RDF first developed for Jena models. RDQL is an implementation of the SquishQL RDF query language, which itself is derived from `rdfDB`. This class of query languages regards RDF as triple data, without schema or ontology information unless explicitly included in the RDF source.

Apart from Sesame, the following systems, too, currently provide RDQL<sup>1</sup>: Jena, RDFStore, PHP XML Classes, 3Store, and RAP (RDF API for PHP).

### 3.1.6.1.2 SPARQL

SPARQL (pronounced “sparkle”) is currently the most popular RDF query language; its name is a recursive acronym that stands for “SPARQL Protocol and RDF Query Language”. It was standardized by the RDF Data Access Working Group (DAWG) of the World Wide Web Consortium, and is now considered a key Semantic Web technology. On 15 January 2008, SPARQL became an official W3C Recommendation.

SPARQL allows for a query to consist of triple patterns, conjunctions, disjunctions, and optional patterns. Several SPARQL implementations for multiple programming languages exist at present.

### 3.1.6.1.3 SeRQL

SeRQL (Sesame RDF Query Language, pronounced “circle”) is an RDF/RDFS query language developed by Sesame’s developer – Aduna – as part of Sesame. It selectively combines the features considered by its creators to be the best of some other (query) languages (RQL, RDQL, N-Triples, N3) and adds some features of its own. As of this writing, SeRQL provides advanced features not yet available in SPARQL. Some of SeRQL’s most important features are:

- Graph transformation
- RDF Schema support
- XML Schema datatype support
- Expressive path expression syntax
- Optional path matching

---

<sup>1</sup> All these implementations are known to derive from the original grammar.

### 3.1.7 Reasoning Strategies

The two principle strategies for rule-based inference are forward-chaining and backward-chaining.

- [Forward-chaining](#): to start from the known facts (the explicit statements) and to perform inference in an inductive fashion. The goals of such reasoning can vary: to compute the [inferred closure](#); to answer a particular query; to infer a particular sort of knowledge (e.g. the class taxonomy).
- [Backward-chaining](#): to start from a particular fact or a query and to verify it or get all possible results, using deductive reasoning. In a nutshell, the reasoner decomposes (or transforms) the query (or the fact) into simpler (or alternative) facts, which are available in the KB or can be proven through further recursive transformations.

Both of these strategies have different strong and weak points, which are studied well in the history of KR and expert systems. Hybrid strategies (involving partial forward- and backward-chaining) are also possible and proven to be efficient in many contexts.

Let us imagine a repository which performs total forward-chaining, i.e. it tries to make sure that, after each update to the KB, the inferred closure is computed and made available for query evaluation or retrieval. This strategy is generally known as [materialization](#). In order to avoid ambiguity with various partial materialization approaches, let us call such an inference strategy, taken together with the monotonic entailment<sup>1</sup> assumption, [total materialization](#).

The principle advantages and disadvantages of the total materialization are discussed at length in [3]; here we provide just a short summary of them:

- Upload/store/addition of new facts is relatively slow, because the repository is extending the inferred closure after each transaction for modification. In fact, all the reasoning is performed during the upload;
- Deletion of facts is also slow, because the repository should remove from the inferred closure all the facts which are not true any longer.
- The maintenance of the inferred closure usually requires considerable additional space (RAM, disk, or both, depending on the implementation);
- Query and retrieval are fast, because no deduction, satisfiability checking, or other sorts of reasoning are required. The evaluation of the queries becomes computationally comparable to the same task for relation database management systems (RDBMS).

Probably the most important advantage of the inductive systems, based on total materialization, is that they can easily benefit from RDBMS-like query optimization techniques, as long as all the data is available at query time. The latter makes it possible for the query evaluation engine to use statistics and other means in order to make "educated" guesses about the "cost" and the "selectivity" of a particular constraint. These optimizations are much more complex in the case of deductive query evaluation.

Total materialization is adopted as reasoning strategy in a number of popular Semantic Web repositories, including some of the standard configurations of Sesame and Jena. Based on publicly available evaluation data, it is also the only strategy which allows scalable reasoning in the range of a

---

<sup>1</sup> Under a monotonic logic, when new explicit facts (statements) are added to the KB (repository), this can cause that new implicit facts can extend its inferred closure, but in no case facts which were part of the inferred closure before, should be removed. In other words, addition of new facts can only monotonically extend the inferred closure.

billion of triples; such results are published by BBN (for DAML DB) and ORACLE (the RDF support in ORACLE 11g).

### 3.1.8 Semantic Repositories

Over the last decade the Semantic Web emerged as an area where semantic repositories become as important as the HTTP servers are today. This perspective boosted the development, under W3C driven community processes, of a number of robust metadata and ontology standards. Those standards play the role, which SQL had for the development and spread of the relational DBMS. Although designed for Semantic Web, these standards face increasing acceptance in areas like Enterprise Application Integration and life sciences.

In the OWLIM-speak (and, therefore, throughout this document), the term “semantic repository” is used to refer to a system for storage, querying, and management of structured data with respect to ontologies. At present, there is no single well-established term for such engines. Weak synonyms are: reasoner, ontology server, metastore, semantic/triple/RDF store, database, repository, knowledge base. The different wording usually reflects a somewhat different approach to implementation, performance, intended application, etc.. Introducing the term “semantic repository”, we are trying to cover the core functionality offered by most of these tools.

Semantic repositories can be used as a replacement for the database management systems (DBMS), offering easier integration of diverse data and more analytical power. In a nutshell, a semantic repository can dynamically interpret metadata schemata and ontologies, which define the structure and the semantics related to the data and the queries. Compared to the approach taken in the relational DBMS, this allows for easier changing and combining of data schemata and automated interpretation of the data. The latter means that, for example, given a simple ontology, a semantic repository can return a mobile operator in the UK, when queried for telecom companies in Europe.

## 3.2 Introduction to Sesame

Sesame is a system/framework for storing, querying and reasoning with RDF data. It is implemented in Java by Aduna as an open source project and includes various storage back-ends (memory, file, database), query languages, inferencers, and client-server protocols.

There are two uses of Sesame:

- as an RDF/RDFS database
- as a Java library for an application designed to work with RDF internally

Sesame enables you to parse, interpret, query and store this kind of structured information. Depending on what your preference is, you can embed that information in your own application or in a separate database (which can even reside on a remote server). In other words, Sesame provides application developers with the functionality they need to process RDF.

Sesame supports the SPARQL RDF query language, which, as mentioned earlier, is the recommended standard of the W3C, and SeRQL (Aduna’s own query language). It also supports most popular RDF file formats and query result formats.

Sesame offers a JDBC-like user API, streamlined system APIs and a RESTful HTTP interface. Various extensions are available or being under development by third parties.

Currently in version 2.X, Sesame is completely targeted at Java 5. All APIs use Java 5 features such as typed collections and iterators. Sesame version 2.1 added support for storing RDF data in relational databases. The supported relational databases are MySQL, PostgreSQL, MS SQL Server, and Oracle. As of version 2.2, Sesame also includes support for Mulgara (a native RDF database).

### 3.2.1 Sesame Architecture

A schematic representation of Sesame's architecture is shown in Figure 4. Following is a brief overview of the main components.

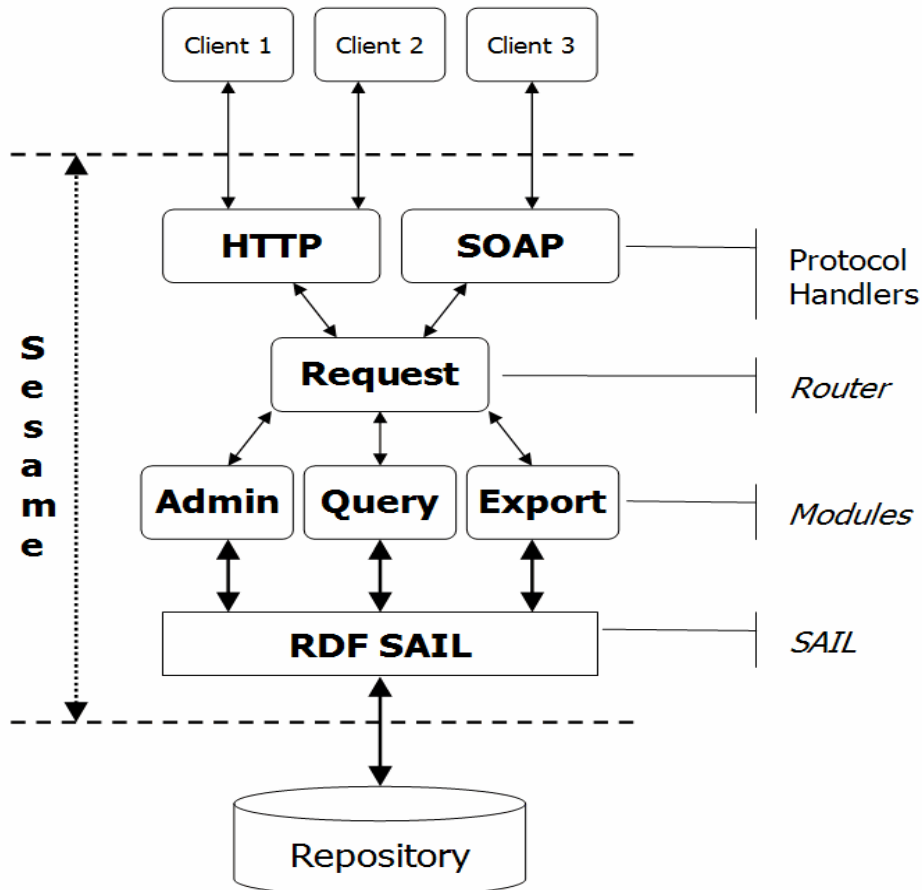


Figure 4 - Sesame's architecture - a schematic representation

Sesame is independent of the actual DataBase Management System (DBMS) that stores the RDF data. It has no DBMS of its own but interfaces to an outside DBMS through its Storage And Inference Layer (SAIL). The SAIL is a single layer that carries all the necessary DBMS-specific code for Sesame. It is an application programming interface (API) that offers RDF-specific methods to its clients and translates these methods to calls to its specific DBMS.

Considering the large variety of DBMSs in existence, this is a very flexible solution allowing the RDF data to be stored in numerous ways depending on the particular DBMS chosen and this choice can be based on what the best approach for the particular application is.

Sesame's functional modules are clients of the SAIL API. Currently, there are three such modules: the Query module (which runs the RQL query engine), the RDF Admin module and the RDF Export module.

Depending on the environment in which it is deployed, different ways to communicate with the Sesame modules may be desirable. For example, communication over HTTP may be preferable in a Web context, but in other contexts, protocols such as Remote Method Invocation (RMI) or the Simple Object Access Protocol (SOAP) (see [2]) may be more suited. In order to allow maximal flexibility, the actual handling of these protocols has been placed outside the scope of the functional modules. Instead, protocol handlers are provided as intermediaries between the modules and their clients, each handling a specific protocol.

The use of the SAIL and the protocol handlers makes Sesame a generic architecture for RDFS storage and querying, rather than just a particular implementation of such a system. Adding additional protocol handlers makes it easy to connect Sesame to different operating environments. As a result, Sesame's architecture is extensible and adaptable – it is possible to use various kinds of repositories and to add additional modules or protocol handlers as well. [5]

### 3.2.2 The SAIL API

The SAIL API is a set of Java interfaces that has been specifically designed for storage and retrieval of RDFS-based information. The main characteristic features of the SAIL are as follows:

- It is the basic interface for storing/retrieving/deleting RDF and RDFS to/from (persistent) repositories.
- It abstracts from the actual storage mechanism: it could be applicable to RDBMSs, file systems, or in-memory storage, for example.
- It can be used on low-end hardware like PDAs, but also offers enough freedom for optimizations to handle huge amounts of data efficiently on e.g. enterprise-level database clusters.
- It is extendable to other RDF-based languages like DAML+OIL.
- It is possible to put multiple SAILS on top of each other. The SAIL at the top can perform some action when the modules make calls to it, and then forward these calls to the SAIL beneath it. This process continues until one of the SAILS finally handles the actual retrieval request, propagating the result back up again.
- It caches all schema data in a dedicated data structure in main memory. This is a good solution because the schema data is often very limited in size and is requested very frequently, while, at the same time, it is the most difficult to query from a DBMS because of the transitivity of the `subClassOf` and `subPropertyOf` properties. (This schema-caching SAIL can be placed on top of some other SAIL(s), handling all calls concerning schema data. The rest of the calls are forwarded to the underlying SAIL.)
- It handles concurrency. Since any given RQL query is broken down into several operations on the SAIL level, it is important to preserve repository consistency over multiple operations. The SAIL selectively blocks and releases read and write access to repositories, on a first come first serve basis. This setup allows for supporting concurrency control for any type of repository.

Other proposals for RDF APIs are currently under development. The most prominent of these are the Jena toolkit and the Redland Application Framework. The SAIL shares many characteristics with both approaches.

An important difference between these two proposals and SAIL, is that the SAIL API specifically deals with RDFS on the retrieval side: it offers methods for querying class and property subsumption, and

domain and range restrictions. In contrast, both Jena and Redland focus exclusively on the RDF triple set, leaving interpretation of these triples to the user application. In SAIL, these RDFS inferencing tasks are handled internally. The main reason for this is that there is a strong relationship between the efficiency of the inferencing and the actual storage model being used. Since any particular SAIL implementation has a complete understanding of the storage model (e.g. the database schema in the case of an RDBMS), this knowledge can be exploited to infer, for example, class subsumption more efficiently.

Another difference between SAIL and other RDF APIs is that SAIL is considerably more lightweight: only four basic interfaces are pre-defined, offering basic storage and retrieval functionality and transaction support, but not much beyond that. The reason for this is that in some applications such minimality may be preferable to an API that has more features, but is also more complex to understand and implement.

The current Sesame system offers several implementations of the SAIL API. The most important of these is the SQL92SAIL, which is a generic implementation for SQL92 [21]. This allows for connecting to any RDBMS while having to re-implement as little as possible. In the SQL92SAIL, only the definitions of the datatypes (which are not part of the SQL92 standard) have to be changed when switching to a different database platform. The SQL92SAIL features an inferencing module for RDFS, based on the RDFS entailment rules as specified in the RDF Model Theory [17]. This inferencing module computes the schema closure of the RDFS being uploaded, and asserts these implicates of the schema as derived statements. For example, whenever a statement of the form (`foo`, `rdfs:domain`, `bar`) is encountered, the inferencing module asserts that (`foo`, `rdf:type`, `property`) is an implied statement. The SQL92SAIL has been tested in use with several DBMSs, including PostgreSQL8 and MySQL9. [5]

## 4 Introduction to OWLIM

OWLIM is a high-performance semantic repository, implemented in Java and packaged as a Storage and Inference Layer (SAIL) for the Sesame RDF database. OWLIM is based on Ontotext's TRREE – a native RDF rule-entailment engine. The supported semantics can be configured through rule-set definition and selection. The most expressive pre-defined rule-set combines unconstrained RDFS and OWL Lite. Custom rule-sets allow tuning for optimal performance and expressivity. OWLIM supports RDFS (page 11), OWL DLP (page 20), OWL Horst (page 21), and most of OWL Lite (page 20).

The two major varieties of OWLIM are SwiftOWLIM and BigOWLIM. In SwiftOWLIM, reasoning and query evaluation are performed in-memory, while, at the same time, a reliable persistence strategy assures data preservation, consistency, and integrity. BigOWLIM is the “enterprise” variety: it deals with data and indices directly from on-disc or other file storage, which allows for highly improved scaling. In addition to that, BigOWLIM's indices are specially designed to allow efficient query evaluation against huge volumes of data. SwiftOWLIM can manage millions of explicit statements on desktop hardware. On an entry-level server, BigOWLIM can handle billions of statements and serve multiple simultaneous user sessions.

A principal limitation of the reasoning strategy adopted (forward-chaining) is the relatively slow delete operation. Uploading, reasoning, and query evaluation proceed extremely fast even against huge ontologies and knowledge bases. According to the limited evaluation data available, SwiftOWLIM is the fastest OWL Lite repository available, while BigOWLIM is the most efficient repository with reasoning support in the enterprise class.

The key differences between the editions of OWLIM are discussed in “OWLIM Editions” on page 32 and in the OWLIM presentation, [28]. The results form a number of benchmarks, as well as plenty of other performance evaluation and analysis information, are provided in [29].

### 4.1 Advantages of OWLIM

One of the main advantages of OWLIM the in-memory reasoning implementation: the full content of the repository is loaded and maintained in the main memory, which allows for efficient retrieval and query answering. Although the reasoning is handled in-memory, the OWLIM SAIL offers a relatively comprehensive persistency and backup strategy.

The persistency of OWLIM is implemented via [N-Triples files](#). The repository can be split into several files. All these files except one are read-only; the writable file is considered as both the source from which the triples are loaded and the target where the new statements are stored. This backup strategy ensures that no loss of newly asserted triples can occur in cases of power failure or abnormal termination – the detailed description is presented in OWLIM User Guides. Although relatively simple, this strategy had proven to be very efficient and reliable over the years during which the RdfSchemaRepositoryV2 and OWLIM have been used as a semantic repository for different applications of the KIM platform (for details, see [22]).

### 4.2 Limitations of OWLIM

The limitations of OWLIM are related to its reasoning strategy. In general, the expressivity of the language supported cannot be extended in the direction of DL. The rule-engine behind OWLIM is

limited in expressivity by the Horn logic. The total materialization strategy has its obvious drawbacks, as discussed briefly in “Reasoning Strategies” on page 23 and in detail in [4]. For specific ontologies and KBs, the count of the implicit statements can appear to grow rapidly. What is even more important, the delete operation is really slow, which means that OWLIM is not suitable for applications where removal of data is a typical transaction. The most obvious disadvantage of the in-memory reasoning is that the size of the KB which can be handled is limited by the size of the available RAM.<sup>1</sup>

### 4.3 OWLIM Interoperability and Architecture

This topic provides information on several technical aspects related to the interoperability, the design and the behaviour of OWLIM.

The inference is performed by the TRREE engine (see [39]), which applies total materialization (see “Reasoning Strategies” on page 23), i.e. it generates and caches the inferred closure – all implicit statements which can be entailed from the current state of the repository, through the currently active rule-set. The contents of the repository and the inferred closure are available in highly-optimized data structures in-memory for query evaluation and further inference. The inferred closure is updated (through inference) at the end of each transaction that modifies the repository.

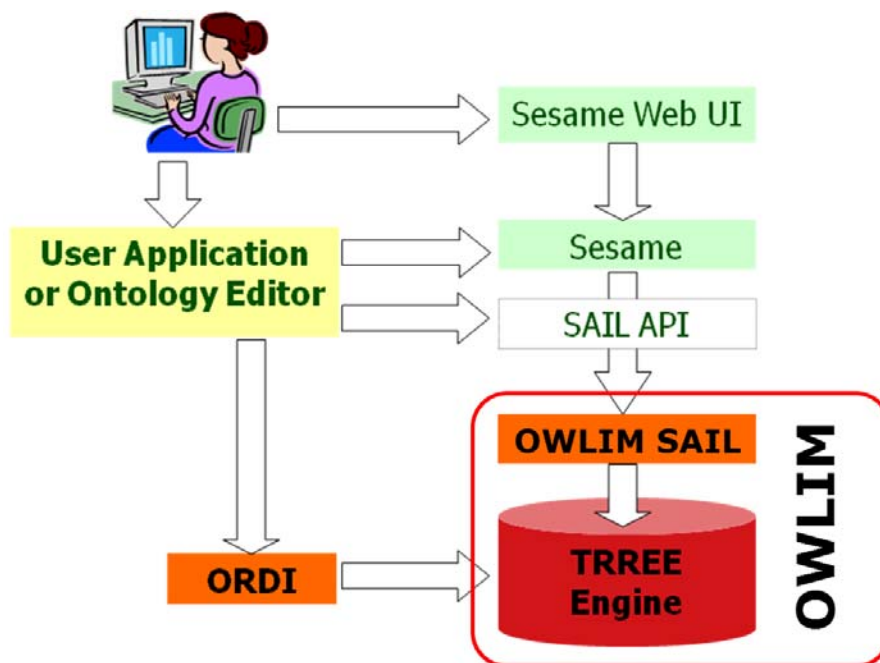


Figure 5 - OWLIM Usage and Relations to Sesame and TRREE

OWLIM implements the SAIL interface of Sesame (refer to The SAIL API topic on page 26), so, it is integrated with all the facilities of Sesame, e.g. the query engines and the web UI. A user application can choose whether to use OWLIM directly, through the low-level SAIL interface, or through the higher-level functional interfaces of Sesame (see “Introduction to Sesame” on page 24). Users can

<sup>1</sup> Considering the currently available commodity hardware, OWLIM can handle millions of statements on desktop machines and above ten millions on an almost-entry-level server.



access the data stored in OWLIM through Sesame's web user interface or through other tools integrated with Sesame (for instance, ontology editors like Protégé and TopBraid Composer).

The easiest way to use OWLIM is in the so-called embedded mode, i.e. as a Java library. The distribution of OWLIM contains a RMI factory (`CustomFactory`), that allows the applications to access the SAIL layer of the repository over RMI. (Further details are provided in OWLIM User Guides.) The installation and configuration of OWLIM are discussed in "Installation and Configuration Overview" on page 37. More information on the various aspects of the Sesame specifications, its architecture and implementations can be found in "Introduction to Sesame" on page 24.

### 4.3.1 Integration with Sesame

OWLIM is a specific configuration for the Sesame RDF database (see "Introduction to Sesame" on page 24) and counts on it for various sorts of features and infrastructure, including, but not limited to, an extensive set of RDF and query language parsers. The fact that Sesame is one of the most mature and popular semantic repositories allows for easy adoption of OWLIM. SwiftOWLIM ver. 3 is packaged as a Storage and Inference Layer (SAIL) for Sesame v.2.x named `OWLIMSchemaRepository`; it implements the `RdfSchemaRepository` interface. More information related to various aspects of Sesame's specification, architecture, and implementations can be found in [41].

### 4.3.2 The TRREE Engine

The TRREE is OWLIM's database engine. TRREE (see [39]) stands for "Triple Reasoning and Rule Entailment Engine". The TRREE performs reasoning based on forward-chaining of entailment rules over RDF triple patterns with variables. TRREE's reasoning strategy is total materialization, as introduced in "Reasoning Strategies" on page 23. A departure from this strategy can be implemented in case of the so called "[transitivity optimization](#)" (see "Transitivity Optimization" on page 31).

#### 4.3.2.1 Rule Format and Semantics

The rule format and the semantics enforced is analogous to R-entailment (see [37] and the Reasoning Strategies topic on page 23) with the following differences:

- Free variables in the head (without binding in the body) are treated as blank nodes. This feature can be considered "syntactic sugar".
- Variable inequality constraints can be specified in the body of the rules, in addition to the triple patterns. This leads to lower complexity as compared to R-entailment.
- the `[cut]` operator can be associated with rule premises, the TRREE compiler interprets it like the `!` operator in Prolog.
- Inconsistency rules are not supported, i.e. there is no specific mechanism to allow inconsistency checks. One can easily model these via regular rules which entail `<x, rdf:type, owl:Nothing>` statements, without affecting the complexity class;
- Axioms can be provided as a set of statements, although those are not modelled as rules with empty bodies.

The TRREE can be configured via "rule-sets" – sets of axiomatic triples and entailment rules, which determine the supported semantics. The implementation of TRREE relies on a compile stage, during which the rules are compiled into chunks of Java code that are post-processed and merged together to generate the main entry point for the inferencer.

The edition of TRREE used in SwiftOWLIM is referred to as “SwiftTRREE” and performs reasoning and query evaluation in-memory; the latter means that the full content of the repository is loaded and maintained in a proprietary representation in the main memory, which allows for rapid reasoning and retrieval. The TRREE edition used in BigOWLIM operates with data and index structures from a file storage. Its data structures are organized to allow query optimizations which improve the query evaluation performance with respect to big datasets dramatically, for instance on one standard tests BigOWLIM evaluates queries against 7 million statements three times faster than SwiftOWLIM, although it takes it two-three times more time to initially load the data.

#### 4.3.2.2 The Rule Language

This language is almost identical with the R-Entailment defined by Horst; the major difference is that at present the TRREE provides no support of the R-Entailment's axiomatic triples and inconsistency rules.

A rule-set file can have up to three sections named **Prefices**, **Axioms**, and **Rules**. The **Rules** section is mandatory; when the other sections exists, the sections must appear sequentially in the order in which they are listed here.

- The **Prefices** section defines the common namespaces.
- The **Axioms** section defines a set of axiomatic triples to be asserted by default into repository. These triples are usually used to describe the meta-level primitives used to define the schema, such as `rdf:type`, `rdfs:Class`, etc..
- The **Rules** section provides all rule definitions. Each definition consists of premises and corollaries that are RDF statements defined via a subject, predicate and object components. Any component can be a variable, a full URI or the short name for the URI. Constraints, too, can be used to state that the values of the variables in a statement must not be equal to some specific full URI (or its short name) or to the value of another variable within the same rule.

#### 4.3.2.3 Transitivity Optimization

The language of the TRREE allows for entering of either single-line or multiple-line comments wherever necessary. The TRREE<sup>1</sup> can operate in a special mode, in which it does not materialize implicit statements, inferred as “closure” of transitive, symmetric, or inverse properties<sup>2</sup>. In this mode, two of the rules related to the support of `owl:TransitiveProperty`, `owl:inverseOf` are skipped; “local” backward-chaining is used in their place to provide equivalent entailments without materialization. Following are the two rules which are searched for in the rule-set and omitted in this mode:

```

Id: owl_invOf
  x p y
  p <owl:inverseOf> q
-----
  y q x

Id: proton_TransitiveOver
  p <protons:transitiveOver> q
    
```

<sup>1</sup> This transitivity optimization appears first in SwiftTRREE version 2.9.

<sup>2</sup> This mode is activated with the `transitive` parameter of OWLIM.

```

x  p  y
y  q  z
-----
x  p  z

```

Inference and query answering is slower in this mode (as compared with the 'normal' one, performing total materialization) since it effectively implements partial backward-chaining, which requires additional calculations and as well as maintenance of intermediate data structures. This strategy pays off in cases when long chains of resources are related through transitive properties. The inferred closure of such chain of n resources (connect through n-1 explicit statements) contains n(n-1)/2 statements. In other words, the inferred closure grows in polynomial (quadratic) dependence on the length of the chain. The situation gets even worse when the transitive property is also symmetric or has an inverse one, as, the inferred closure of such chain doubles in size.

### 4.4 OWLIM Editions

OWLIM comes in two major OWLIM editions – SwiftOWLIM and BigOWLIM – and, additionally, in two versions depending on the supported version of Sesame.

#### 4.4.1 SwiftOWLIM and BigOWLIM

The two major OWLIM editions – SwiftOWLIM and BigOWLIM – are identical in terms of usage and integration except for some minor differences in a few configuration parameters. The editions differ in the respective version of the TRREE engine they are based upon, but share the same inference and semantics (rule-compiler, etc.).

SwiftOWLIM is designed for medium data volumes and prototyping. Its key features are:

- reasoning and query evaluation performed in the main memory, so it keeps the major part of its indices there
- persistence strategy that assures data preservation and consistency
- extremely fast loading of data (including inference and storage)

BigOWLIM is suitable for massive volumes of data and heavy query loads. It is designed as an enterprise-grade database management system. It features:

- file-based indices (Enables it to scale to billions of statements even on desktop machines.)
- query optimizations (Ensures fast query evaluations.)
- optimized handling of `owl:sameAs` (identifier equality) (Boosts its efficiency for data integration tasks.)

Parameter ↓	SwiftOWLIM	BigOWLIM
<b>Scale</b> (Mill. of explicit St.)	10 MSt, using 1.6 GB RAM <b>100 MSt</b> , using 16 GB RAM	130 MSt, using 1.6GB <b>1068 MSt</b> , using 12GB
<b>Processing speed</b> (load+infer+store)	30 KSt/s on notebook <b>200 KSt/s</b> on server	5 KSt/s on notebook <b>60 KSt/s</b> on server
<b>Query optimization</b>	No	Yes

Parameter ↓	SwiftOWLIM	BigOWLIM
<b>Persistence</b>	Back-up in N-Triples	Binary data files and indices
<b>Efficient owl:sameAs</b>	No	Yes
<b>Licence and Availability</b>	Open-source under LGPL; Uses SwiftTRREE that is free, but not open-source	Commercial. Research and evaluation copies provided for free

Table 2 - Comparison between SwiftOWLIM and BigOWLIM

#### 4.4.2 Versioning of OWLIM

OWLIM's versions are designed and tailored to meet as efficiently as possible the wide range of requirements towards RDF databases and reasoning engines. The following table summarizes the major differences between the versions that are currently supported.

Feature ⇒ OWLIM Edition and version number ↓	Sesame version	SPARQL support	Instant initializ.	owl:sameAs optimization	Comment
<b>SwiftOWLIM 2.9.x</b>	1.2.x	–	–	–	The fastest OWL database. Multi-threaded inference, with transitive inference optimization.
<b>BigOWLIM 2.x</b>	1.2.x	–	yes	yes	Optimal performance and scalability. The fastest query evaluation. Successor of 0.9.x.
<b>SwiftOWLIM 3.x</b>	2.x.x	yes	yes	–	The fastest RDF machine with NG and SPARQL support.
<b>BigOWLIM 3.x</b>	2.x.x	yes	yes	yes	Ultimate scalability and fast SPARQL evaluation.

Table 3 - Comparison table of all OWLIM versions

#### 4.5 Supported Semantics

OWLIM offers several predefined levels of entailment but allows also for customization of the supported semantics. The particular semantics to be used is specified (through the `ruleset` parameter, see the Configuration topic in the User Guide) for each specific repository instance. Applications, which do not need the complexity of the most expressive supported semantics, can

choose one of the lower levels, which will result in faster inference. The division of the rules into the different complexity levels is clearly indicated (through comments) in the `rules.pie` file.

The semantics supported by OWLIM is highly configurable – the TRREE engine can be configured to work with not only with any of the several pre-defined rule-sets but also with a rule-set defined by the user. The complexity, and thus the speed, of the inference can vary considerably across different rule-sets.

#### 4.5.1 Pre-Defined Rule Sets

The pre-defined rule-sets are “nested” into each other so that each one is extending the preceding one ; following is a list ordered by increasing complexity:

- **empty**: no reasoning, i.e. OWLIM operates as a plain RDF store;
- **rdfs**: supports the standard theoretic RDFS semantics model;
- **owl-horst**: OWL dialect close to OWL Horst; the differences are discussed in “OWL Compliance” below.
- **owl-max**: a combination of most of the semantics of OWL Lite in combination with full compatibility with (support for) RDFS.

Additional variety of the supported level of semantics can be brought in through the so-called **partialRDFS** modification of the predefined rule-sets. It is discussed in OWLIM User Guides. Define custom rule-sets is always an option, as described in “Creating Custom Configurations” on page 38. The rule-set to be used for a specific repository is defined through the `ruleset` parameter (For details, refer to OWLIM User Guides).

The richest rule-set (`owl-max`) encompasses the model theoretic semantics of RDFS, as defined in [18], extended with support for most of the OWL primitives as explained in the OWL Compliance topic below. There are certain limitations to the support of these primitives. They are discussed in OWLIM User Guides.

#### 4.5.2 OWL Compliance

Regarding OWL compliance, OWLIM supports a dialect (rule-set `owl-horst`) similar to OWL Horst, as defined in [37] and introduced in “OWL Horst” on page 21. The OWLIM support of OWL with the `owl-max` rule-set and without **partialRDFS** optimizations is as follows:

- OWLIM supports the full RDFS semantics without constraints or limitations, apart from the entailments related to typed literals (known as D-entailment). For instance, meta-classes (and any arbitrary mixture of class, property, and individual) can be combined with the supported OWL semantics.
- The supported OWL semantics does not cover the full OWL Lite. The OWL semantics supported by OWLIM, combined with unconstrained RDFS semantics, constitutes a language which is not comparable to OWL Lite – its full compatibility with RDFS makes it more expressive than OWL Lite in some aspects.
- OWLIM supports a language richer than OWL DLP;

The differences between OWL Horst, [37], and the OWL dialects supported by OWLIM through `owl-horst` and `owl-max` rule-sets can be summarized as follows:

- OWLIM does not provide the extended support for typed literals, introduced with the D\*-entailment extension of the RDFS semantics. Although such support is conceptually clear and easy to implement, it is our understanding that the performance “penalty” is too high for most applications. One can easily implement adapt the rules defined for this purpose by ter Horst and put them in a customized rule-set (see “Custom Rule-Sets” on page 35 and OWLIM User Guides).
- There are no inconsistency rules.
- Few more OWL primitives are supported by OWLIM (rule-set `owl-max`). These are listed in OWLIM User Guides.
- There is extended support for schema-level (T-Box) reasoning in OWLIM.

Even though the concrete rules pre-defined in OWLIM differ from those defined in OWL Horst in [37], the complexity and decidability results reported for R-entailment are relevant for TRREE and OWLIM. Put it more precisely, the rules in the `owl-horst` rule-set, do not introduce new B-Nodes, which means that R-entailment with respect to them takes polynomial time. In KR terms, this means that the `owl-horst` inference within OWLIM is tractable.

OWLIM implements reasoning with lesser complexity, as compared to other formalisms, which combine DL ontologies with rules. In addition, it puts no constraints with respect to meta-modelling.

The correctness of the support of the OWL semantics (for those primitives which are supported) is checked against the normative Positive and Negative-entailment OWL test cases, [7]. These checks are available as JUnit tests together with the OWLIM distribution; they are documented in [29] and can also be used as sample applications.

### 4.5.3 PROTON Primitives

Besides the RDF- and OWL-specific primitives, the rules, predefined in OWLIM, provide semantics for primitives from the PROTON ontology, [31]. The most important example is the rule that supports `transitiveOver` property, which is defined as follows:

```

Id: proton_TransitiveOver
  p <protons:transitiveOver> q
  x p y
  y q z
-----
  x p z
    
```

This property is used later on for alternative definition of the semantics of some RDFS primitives. These definitions support semantics, equivalent to the one given, for instance, with the normative axioms and rules in [18], but allow for better inference speed. They have no “side-effects” on the entailment, unless the corresponding PROTON properties are being used by the application. `transitiveOver` is also used for definition of the semantics of some of the OWL primitives.

The rules supporting the semantics of the PROTON primitives are clearly marked in the `rules.pie` file.

### 4.5.4 Custom Rule-Sets

OWLIM has an internal rule compiler that could be used to configure the TRREE with a custom set of inference rules and axioms. The user may define a proprietary rule-set in a `*.pie` file (e.g.

MySemantics.pie). The easiest way to come up with your own rule-set is to start modifying one of the .pie files that were used to build the precompiled rule-sets. (All pre-defined \*.pie files are included in the distribution.) The syntax of the .pie files is easy to follow; the rule-sets utilize Java syntax.

## 5 Installation and Configuration Overview

OWLIM is a specific plug-in (namely, a storage and inference layer, SAIL) for Sesame. To configure, run, and use OWLIM means to do so for a specific configuration of Sesame. Please, refer to the online documentation of Sesame, [41]. Instructions and samples of how an application uses Sesame in embedded mode can be found there, too.

As different versions of OWLIM are compliant with different versions of Sesame, code and configuration files samples are not provided here but in OWLIM User Guides.

### 5.1 Contents of the Distribution Package

There are minor differences between the distributions of the different version of OWLIM – those are commented accordingly in the OWLIM User Guides.

Licensing information about the corresponding version of OWLIM and Sesame can be found in files `licence.txt` and `licence_sesame.txt`, respectively, in the main folder of the distribution. The distributions of OWLIM include also:

<b>lib</b> folder	The binary executable version of OWLIM as a JAR (Java library) files.
<b>ext</b> folder	All required third party libraries are placed there for user's convenience. The libraries of the corresponding version of Sesame could be downloaded separately from <a href="http://www.openrdf.org/">http://www.openrdf.org/</a> . The folder also contains a copy of Leigh University Benchmark library ( <code>lubm.jar</code> ) and also JUnit v3.8.1. They are needed for the execution of the respective tests.
<b>doc</b> folder	OWLIM user documentation including this document and tests documentation.
<b>src</b> folder	The Java sources of OWLIM and the RMI factory.
<b>test</b> folder	Contains sources, data, and documentation of the unit tests and benchmarks, documented in [29].
<b>standalone</b>	A folder with the scripts for running OWLIM as a standalone server enabled for RMI access.
<b>*.pie</b> files	Contain description of the built-in rule-sets.
<b>getting-started</b> folder	A folder with a an example setup for an application using OWLIM, with all required auxiliary files and folders. Can be used as a template for creating custom OWLIM configurations.
<b>wordnet</b>	A folder with copy of the getting-started template customized for loading the RDF/OWL representation of WordNet 2.0 and performing a set of queries against it.
<b>setvars</b>	script ( <code>.cmd</code> or <code>.sh</code> , respectively for Windows or Linux): a batch file defining several environment variables used by the scripts that run the test cases and the getting-started application. It should be customized for each individual installation as it determines the Java machine to be started and the path to all the JAR files used including those of OWLIM and Sesame.



## 5.2 How to Get Started Quickly

OWLIM comes equipped with an example application setup (the `getting-started` folder, see the Contents of the Distribution Package topic above), which can be used as a template for bootstrapping applications that use OWLIM. The sample code of this application performs a sequence of typical operations: initialization of the repository, loading a KB into it, performing queries and obtaining results, and making modifications. This application template comes packed with:

- source code and compiled class files
- sample ontology and data files
- Sesame configuration file
- scripts which invoke the application

One easy way for setting up an application using OWLIM is to copy the `getting-started` folder and modify the contents as necessary. The easiest way to learn how to use it is to read the source code of the `GettingStarted` class, located in the `src` folder; the code is commented extensively. The configuration parameters are in the `owlim.properties` file in `getting-started` folder; these parameters can be changed easily. For details on the configuration parameters and how the sample application operates, refer to OWLIM User Guides.

## 5.3 Creating Custom Configurations

There are three main ways to create a custom configuration:

- You can change the semantic repository you operate on.
- You can change the query language.
- You can modify the rule-set, either partially (by adding, removing, or modifying statements) or completely (including changing the ontology language and the inference logic).

Further details on this subject are available in OWLIM User Guides.

## 6 Glossary of Terms

## 7 References

- [1] BERGMANN, F. **Introduction to Description Logics**. Web page, <http://www.fraber.de/sitec/dl.html>
- [2] BOX, D; EHNEBUSKE, D; KAKIVAYA, G; LAYMAN, A; MENDELSON, N; NIELSEN, H F; THATTE, S; WINER, D. **Simple Object Access Protocol (SOAP) 1.1**, *W3c note*, World Wide Web Consortium, May 2000  
<http://www.w3.org/TR/SOAP/>
- [3] BRICKLEY, D., GUHA, R.V; (eds.). **Resource Description Framework (RDF) Schemas**, W3C  
<http://www.w3.org/TR/2000/CR-rdf-schema-20000327/>
- [4] BROEKSTRA, J. **Storage, Querying and Inferencing for Semantic Web Languages**. Ph.D. thesis, *SIKS Dissertation Series* No. 2005-09, ISBN 90 9019 2360, Vrije Universiteit Amsterdam. 2005.  
<http://www.cs.vu.nl/~jbroeks/#pub>
- [5] BROEKSTRA, J; Kampman, A; van Harmelen, F. **Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema**. *International Semantic Web Conference*, Sardinia, Italy, 2002.
- [6] CARROLL, J J; BIZER, C; HAYES, P; STICKLER, P. **Named Graphs, Provenance and Trust**. *International Semantic Web Conference*, Hiroshima, Japan, 2004.
- [7] CARROLL, J. J; DE ROO, J. **OWL Web Ontology Language: Test Cases**. *W3C Recommendation* 10 Feb. 2004.  
<http://www.w3.org/TR/owl-test/>
- [8] DEAN, M; SCHREIBER, G; (eds.); BECHHOFFER, S; VAN HARMELEN, F; HENDLER, J; HORROCKS, I; MCGUINNESS, D L; PATEL-SCHNEIDER, P F; STEIN, L A. **OWL Web Ontology Language Reference**. *W3C Recommendation*. 10 Feb. 2004.  
<http://www.w3.org/TR/owl-ref/>
- [9] **Dublin Core Metadata Element Set**, Version 1.1.  
<http://dublincore.org/documents/dces/>
- [10] **Dublin Core Metadata Element Set**, Version 1.1: Reference Description.  
<http://dublincore.org/documents/2003/06/02/dces/>
- [11] EHRIG, M; HAASE, P; HEFKE, M; STOJANOVIC, N. **Similarity for ontologies — a Comprehensive Framework**. *Proc. 13th European Conference on Information Systems*, May 2005.
- [12] GROSOFF, B; HORROCKS, I; VOLZ, R; DECKER, S. **Description Logic Programs: Combining Logic Programs with Description Logic**. In *Proc. of WWW2003*, Budapest, May 2003.
- [13] GRUBER, T R. **A translation approach to portable ontologies**. *Knowledge Acquisition*, 5(2):199-220, 1993.  
[http://ksl-web.stanford.edu/KSL\\_Abstracts/KSL-92-71.html](http://ksl-web.stanford.edu/KSL_Abstracts/KSL-92-71.html)
- [14] GRUBER, T R. **Toward Principles for the Design of Ontologies Used for Knowledge Sharing**. Presented at the Padua workshop on Formal Ontology. March 1993, later published in *International Journal of Human-Computer Studies*, Vol. 43, Issues 4-5, Nov. 1995, pp. 907-928.  
<http://tomgruber.org/writing/onto-design.htm>
- [15] GUARINO, N; GIARETTA, P. **Ontologies and Knowledge Bases: Towards a Terminological Clarification**. In N. Mars (ed.) *Towards Very Large Knowledge Bases: Knowledge Building and Knowledge Sharing*. IOS Press, Amsterdam: pp. 25-32. 1995

- [16] GUARINO, N. **Formal Ontology in Information Systems**. *Proceedings of FOIS'98*, Trento, Italy, June 6-8, 1998. Amsterdam, IOS Press.  
<http://www.loa-cnr.it/Papers/FOIS98.pdf>
- [17] HAYES, P. **RDF Model Theory**. *Working draft*, World Wide Web Consortium. September 2001.  
<http://www.w3.org/TR/rdf-mt/>
- [18] HAYES, P. **RDF Semantics**. *W3C Recommendation*. Feb. 10, 2004.  
<http://www.w3.org/TR/2004/REC-rdf-mt-20040210/>
- [19] HILLMANN, D.; **Using Dublin Core**; *DCMI Recommended Resource*. Nov. 7, 2005
- [20] HORROCKS, I. PATEL-SCHNEIDER, P F, BECHHOFFER, S, TSARKOV, D. **OWL Rules: A Proposal and Prototype Implementation**. *Journal of Web Semantics*, 3 (2005), pp. 23-40.
- [21] ISO. **Information Technology-Database Language SQL**. *Standard No. ISO/IEC 9075:1999*, International Organization for Standardization (ISO), 1999. (Available from American National Standards Institute, New York, NY 10036, (212) 642-4900.).
- [22] **KIM**. Home page,  
<http://www.ontotext.com/kim>
- [23] KIRYAKOV, A. **Ontologies for Knowledge Management**, Chapter 7 in: Davies, J; Studer, R; Warren, P. (eds.). *Semantic Web Technologies: Trends and Research in Ontology-based Systems*. Wiley, UK, 2006.
- [24] KLYNE, G; CARROL, J. J; (eds). (2004). **Resource Description Framework (RDF): Concepts and Abstract Syntax**. *W3C Recommendation* 10 Feb. 2004.  
<http://www.w3.org/TR/rdf-concepts/>
- [25] MOTIK, B; SATTLER, U; STUDER R. **Query Answering for OWL-DL with Rules**. *Journal of Web Semantics*, issue 3 (2005), pp. 41-60.
- [26] **Named Graphs**. W3C Overview.  
<http://www.w3.org/2004/03/trix/>
- [27] **Ontology Logic and Reasoning at Semantic Karlsruhe**. Home page,  
<http://logic.aifb.uni-karlsruhe.de/>
- [28] **OWLIM – Pragmatic OWL Semantic Repository**. Presentation slides, Ontotext AD, 2008  
<http://www.ontotext.com/owlim/OWLIMPRES.pdf>
- [29] **OWLIM Tests and Benchmarks**. Ontotext Lab. 2007.  
<http://www.ontotext.com/owlim/v2.9.0/doc/OWLIMTest.pdf>
- [30] Popov, B; Kiryakov, A; Kirilov, A; Manov, D; Ognyanoff, D; Goranov, M. **KIM – Semantic Annotation Platform**. In *The Semantic Web - ISWC 2003*, Sanibel Island, USA, 2003.
- [31] **PROTON Ontology (PROTo Ontology)**. Home page.  
<http://proton.semanticweb.org/>
- [32] **RDF Primer**. In *W3C Recommendation* 10 February 2004.  
<http://www.w3.org/TR/rdf-primer/>
- [33] **RDF/XML Syntax Specification (Revised)**. In *W3C Recommendation*, 10 February 2004.  
<http://www.w3.org/TR/rdf-syntax-grammar/>
- [34] **Resource Description Framework (RDF): Concepts and Abstract Syntax**, section Graph Data Model. In *W3C Recommendation* 10 February 2004.  
<http://www.w3.org/TR/rdf-concepts/#section-data-model>
- [35] **Semantic Knowledge Technologies (SEKT)**. Home page.  
<http://www.sekt-project.com/>

- [36] *SWAD-Europe Workshop on Semantic Web Storage and Retrieval*. Amsterdam, Holland, November 2003.  
[http://www.w3.org/2001/sw/Europe/reports/dev\\_workshop\\_report\\_4/#owl-tiny](http://www.w3.org/2001/sw/Europe/reports/dev_workshop_report_4/#owl-tiny).
- [37] TER HORST, H J. **Combining RDF and Part of OWL with Rules: Semantics, Decidability, Complexity**. In *Proc. of ISWC 2005*, Galway, Ireland, Nov. 6-10, 2005. LNCS 3729, pp. 668-684.
- [38] TERZIEV, I; KIRYAKOV, A. **PROTo ONtology: A Base Upper-Level Ontology for the Semantic Web**, *SEKT Q4 Meeting*. Innsbruck, Austria., Jan. 17-19, 2005.  
<http://proton.semanticweb.org/PROTON.ppt>
- [39] **TRREE – Triple Reasoning and Rule Entailment Engine**. Home page.  
<http://ontotext.com/trree/>
- [40] **Uniform Resource Identifier**, Wikipedia.  
<http://en.wikipedia.org/wiki/URI>
- [41] **User Guide of Sesame**. Aduna b. v..  
<http://www.openrdf.org/doc/sesame/users/index.html>