Programming The Semantic Web

Chapter · January 2006

DOI: 10.1007/978-0-387-34685-4_14

1 author:

Jorge Cardoso
University of Coimbra
307 PUBLICATIONS **7,251** CITATIONS

SEE PROFILE

Chapter 14

PROGRAMMING THE SEMANTIC WEB

Jorge Cardoso
[1]*Department of Mathematics and Engineering,University of Madeira, 9000-390, Funchal, Portugal – jcardoso@uma.pt*

1. INTRODUCTION

Many researchers believe that a new Web will emerge in the next few years based on the large-scale ongoing research and developments in the semantic Web. Nevertheless, the industry and its main players are adopting a "wait-and-see" approach to see how real-world applications can benefit from semantic Web technologies (Cardoso, Miller et al. 2005). The success of the semantic Web vision (Berners-Lee, Hendler et al. 2001) is dependant on the development of practical and useful semantic Web-based applications.

While the semantic Web has reached considerable stability from the technological point of view with the development of languages to represent knowledge (such as OWL (OWL 2004)), to query knowledge bases (RQL (Karvounarakis, Alexaki et al. 2002) and RDQL (RDQL 2005)), and to describe business rules (such as SWRL (Ian Horrocks, Peter F. Patel-Schneider et al. 2003)), the industry is still skeptical about its potential. For the semantic Web to gain considerable acceptance from the industry it is indispensable to develop real-world semantic Web-based applications to validate and explore the full potential of the semantic Web (Lassila and McGuinness 2001). The success of the semantic Web depends on its capability of supporting applications in commercial settings (Cardoso, Miller et al. 2005).

In several fields, the technologies associated with the semantic Web have been implemented with considerable success. Examples include semantic Web services (OWL-S 2004), tourism information systems (Cardoso 2004), semantic digital libraries, (Shum, Motta et al. 2000), semantic Grid (Roure, Jennings et al. 2001), semantic Web search (Swoogle 2005), and bioinformatics (Kumar and Smith 2004).

To increase the development of semantic Web systems and solutions, in this chapter we will show how semantic Web applications can be developed using the Jena framework.

2. THE SEMANTIC WEB STACK

The semantic Web identifies a set of technologies, tools, and standards which form the basic building blocks of an infrastructure to support the vision of the Web associated with meaning. The semantic Web architecture is composed of a series of standards organized into a certain structure that is an expression of their interrelationships. This architecture is often represented using a diagram first proposed by Tim Berners-Lee (Berners-Lee, Hendler et al. 2001). Figure 14-1 illustrates the different parts of the semantic Web architecture. It starts with the foundation of URIs and Unicode. On top of that we can find the syntactic interoperability layer in the form of XML, which in turn underlies RDF and RDF Schema (RDFS). Web ontology languages are built on top of RDF(S). The three last layers are the logic, proof, and trust, which have not been significantly explored. Some of the layers rely on the digital signature component to ensure security.

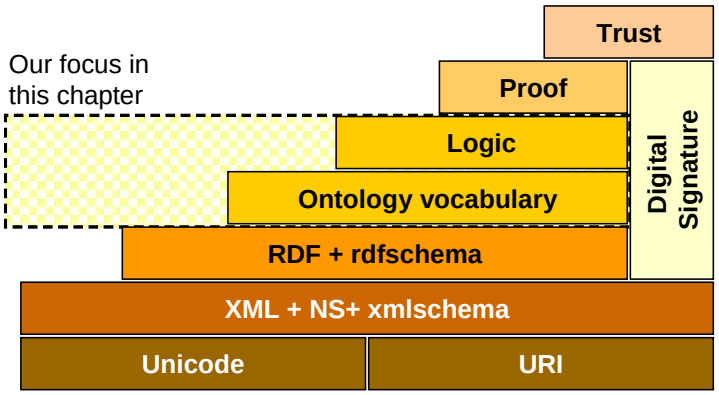

Figure 14-1. Semantic Web stack (Berners-Lee, Hendler et al. 2001)

In the following sections we will briefly describe these layers. While the notions presented have been simplified, they provide a reasonable conceptualization of the various components of the semantic Web.

URI and Unicode. A Universal Resource Identifier (URI) is a formatted string that serves as a means of identifying abstract or physical resource. A URI can be further classified as a Uniform Resource Locator (URL) or a Uniform Resource Name (URN). A URL identifies resources via a representation of their primary access mechanism. A URN remains globally unique and persistent even when the resource ceases to exist or becomes unavailable.

Unicode provides a unique number for every character, independently of the underlying platform or program. Before the creation of unicode, there were various different encoding systems making the manipulation of data complex and required computers to support many different encodings.

XML. XML is accepted as a standard for data interchange on the Web allowing the structuring of data but without communicating the meaning of the data. It is a language for semi-structured data and has been proposed as a solution for data integration problems, because it allows a flexible coding and display of data, by using metadata to describe the structure of data. While XML has gained much of the world's attention it is important to recognize that XML is simply a way of standardizing data formats. But from the point of view of semantic interoperability, XML has limitations. One significant aspect is that there is no way to recognize the semantics of a particular domain because XML aims at document structure and imposes no common interpretation of the data (Decker, Melnik et al. 2000). Even though XML is simply a data-format standard, it is part of the set of technologies that constitute the foundations of the semantic Web.

RDF. At the top of XML, the World Wide Web Consortium (W3C) has developed the Resource Description Framework (RDF) (RDF 2002) language to standardize the definition and use of metadata. RDF uses XML and it is at the base of the semantic Web, so that all the other languages corresponding to the upper layers are built on top of it. RDF is a simple general-purpose metadata language for representing information in the Web and provides a model for describing and creating relationships between resources. RDF defines

a resource as any object that is uniquely identifiable by a URI. Resources have properties associated with them. Properties are identified by property-types, and property-types have corresponding values. Property-types express the relationships of values associated with resources. The basic structure of RDF is very simple and basically uses RDF triples in the form of (subject, predicate, object). RDF has a very limited set of syntactic constructs and no other constructs except for triples is allowed.

RDF Schema. The RDF Schema (RDFS 2004) provides a type system for RDF. Briefly, the RDF Schema (RDFS) allows users to define resources (rdfs:Resource) with classes, properties, and values. The concept of RDFS class (rdfs:Class) is similar to the concept of class in object-oriented programming languages such as Java and C++. A class is a structure of similar things and inheritance is allowed. This allows resources to be defined as instances of classes. An RDFS property (rdf:Property) can be viewed as an attribute of a class. RDFS properties may inherit from other properties (rdfs:subPropertyOf), and domain (rdfs:domain) and range (rdfs:range) constraints can be applied to focus their use. For example, a domain constraint is used to limit what class or classes a specific property may have and a range constraint is used to limit its possible values. With these extensions, RDFS comes closer to existing ontology languages.

Ontologies. An ontology is an agreed vocabulary that provides a set of well-founded constructs to build meaningful higher level knowledge for specifying the semantics of terminology systems in a well defined and unambiguous manner. Ontologies can be used to assist in communication between humans, to achieve interoperability and communication among software systems, and to improve the design and the quality of software systems (Jasper and Uschold 1999).

In the previous sections, we have established that RDF and RDFS were the base models and syntax for the semantic Web. On the top of the RDF/S layer it is possible to define more powerful languages to describe semantics. The most prominent markup language for publishing and sharing data using ontologies on the Internet is the Web Ontology Language (OWL 2004). OWL adds a layer of expressive power to RDF/S, providing powerful mechanisms for defining complex conceptual structures, and formally describes the semantics of classes and properties used in Web resources using, most

commonly, a logical formalism known as Description Logic (DL 2005).

Logic, Proof, and Trust. The purpose of this layer is to provide similar features to the ones that can be found in First Order Logic (FOL). The idea is to state any logical principle and allow the computer to reason by inference using these principles. For example, a university may decide that if a student has a GPA higher than 3.8, then he will receive a merit scholarship. A logic program can use this rule to make a simple deduction: "David has a GPA of 3.9, therefore he will be a recipient of a merit scholarship."

The use of inference engines in the semantic Web allows applications to inquire why a particular conclusion has been reached (inference engines, also called reasoners, are software applications that derive new facts or associations from existing information.). Semantic applications can give proof of their conclusions. Proof traces or explains the steps involved in logical reasoning.

Trust is the top layer of the Semantic Web architecture. This layer provides authentication of identity and evidence of the trustworthiness of data and services. While the other layers of the semantic Web stack have received a fair amount of attention, no significant research has been carried out in the context of this layer.

3. SEMANTIC WEB DEVELOPMENT ENVIRONMENTS

Several frameworks supporting OWL ontologies are available. We will briefly discuss the ones that are used the most by the developer community, namely the Jena framework, Protégé-OWL API and the WonderWeb OWL API, which are all available for Java language. These three APIs are open-source and thus interested people can carry out an in-depth study of their architecture. This is very important for the current stage of semantic Web development since it is difficult to know what the application's scope of the semantic Web will be in the near future. Therefore, open frameworks will allow for an easier integration of semantic Web components into new projects.

Jena (Jena 2002; Jena 2005) is a Java framework for building semantic Web applications developed by the HP Labs Semantic Web Programme. It provides a programmatic environment for RDF, RDFS and OWL, including a rule-based inference engine and a query

language for RDF called RDQL (RDQL 2005). Since we are mostly interested in ontology support, in subsequent sections we will discuss the Jena 2 Ontology API included in the Jena toolkit. This API supports several ontology description languages such as DAML, DAML+OIL and OWL. However building ontologies in OWL W3C's language is strongly recommended because DAML and DAML+OIL support may be removed in future releases of Jena. Because Jena 2 Ontology API is language-neutral, it should be easy to update existing projects using Jena and other ontology languages to support OWL. Jena OWL API supports all three OWL sublanguages, namely OWL Lite, OWL DL and OWL Full. Specifying an URI to an OWL ontology, Jena parses the ontology and creates a model for it. With this model it is possible to manipulate the ontology, create new OWL classes, properties or individuals (instances). The parsing of OWL documents can be highly resource consuming, especially for documents describing large ontologies. To address this particularity, Jena provides a persistence mechanism to store and retrieve ontology models from databases efficiently. As stated before, Jena includes an inference engine which gives reasoning capabilities. Jena provides three different reasoners that can be attached to an ontology model, each of them providing a different degree of reasoning capability. More capable reasoners require substantially more time to answer queries. Therefore, developers should be very careful when choosing a reasoner. Of course, it is possible to create a model with no reasoner defined. An interesting aspect of Jena is that its inference engine is written in a very generic way so that it allows developers to write their own inference rules to better address their needs. This generic implementation also allows for attaching any reasoner that is compliant with the DIG interface, which is a standard providing access to reasoners, such as Racer, FaCT, and Pellet. Another important aspect is that it is very easy to find documentation and practical programming examples for Jena.

Protégé (Protégé 2005) is a free, open-source platform that provides a growing user community with a suite of tools to construct domain models and knowledge-based applications with ontologies. It was developed by the Stanford Medical Informatics Labs of the Stanford School of Medicine. The Protégé-OWL API is an open-source Java library for OWL and RDF(S). The API provides classes and methods to load and store OWL files, to query and manipulate OWL data models, and to perform reasoning (Protégé-API 2006). This API, which is part of the Protégé-OWL plug-in, extends the

Protégé Core System based on frames so that it can support OWL ontologies and allows users to develop OWL plug-ins for Protégé or even to create standalone applications. Protégé-OWL API uses Jena framework for the parsing and reasoning over OWL ontologies and provides additional support for programming graphical user interfaces based on Java Swing library. The Protégé-OWL API architecture follows the model-view pattern, enabling users to write GUIs (the "view") to manipulate the internal representation of ontologies (the "model"). This architecture, together with the event mechanism also provided, allows programmers to build interactive user interfaces in an efficient and clean way. A community even stronger than Jena's one has grown around Protégé, making it very easy to find good documentation, examples and support for this API.

WonderWeb OWL API (OWLAPI 2006) is another API providing programmatic services to manipulate OWL ontologies. It can also infer new knowledge once a reasoner is attached to the ontology model. Pellet is one of the reasoners that is currently supported. One should note that the current release of this API is still in working progress. Consequently, there are some issues that need to be corrected. Nevertheless, WonderWeb OWL API was successfully used in several projects such as Swoop (SWOOP 2006) and Smore (SMORE 2006), respectively, an ontology editor and a semantic annotation tool, from the MIND LAB at the University of Maryland Institute for Advanced Computer Studies. This demonstrates that this API is mature enough to be considered when developing semantic Web applications. One major drawback of the WonderWeb OWL API is lack of documentation. Currently, Javadoc documentation and some open-source applications that use this API, is what can be found about it. It is very difficult to find practical examples. This fact may lead developers to choose to discard this API.

4. OUR RUNNING ONTOLOGY

Our recent work has involved the development of a Semantic Course Management System (S-CMS). Course management systems (CMS) are becoming increasingly popular. Well-known CMSs include Blackboard.com and WebCT.com whose focus has centered on distance education opportunities. Typically, a CMS include a variety of functionalities, such as class project management, registration tool for students, examinations, enrolment management, test

administration, assessment tools, and online discussion boards (Meinel, Sack et al. 2002).

The S-CMS system that we have developed is part of the Strawberry project [1] and explores the use of semantic Web technologies to develop an innovative CMS. The S-CMS provides a complete information and management solution for students and faculty members. Our focus and main objective was to automate the different procedures involved when students enroll or register for class projects. Managing a large course and its class projects is a complex undertaking. Many factors may contribute to this complexity, such as a large number of students, the variety of rules that allow students to register for a particular project, students' background, and student's grades.

The development of a semantic Web application typically starts with the creation of one or more ontology schema. For simplicity reasons, in this chapter we will only present one ontology, the University ontology. This ontology will be used in all the programming examples that we will show. As with any ontology, our ontology contains the definition of the various classes, attributes, and relationships that encapsulate the business objects that model a university domain. The class hierarchy of our simple ontology is shown in Figure 14-1 using the OWL Viz Protégé plug-in (OWLViz 2006).

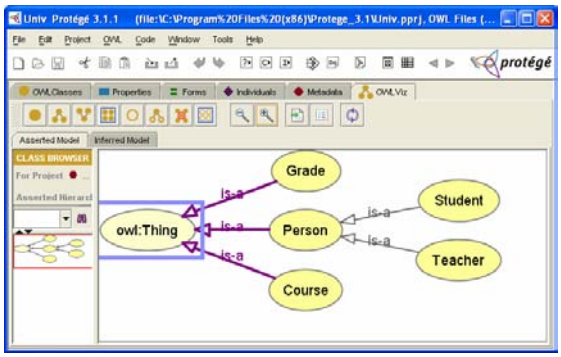

Figure 14-2. Class hierarchy

Some of the properties of our ontology are shown in Figure 14-2 using Protégé (Protégé 2005).

[1] http://dme.uma.pt/jcardoso/Research/Projects/Strawberry/

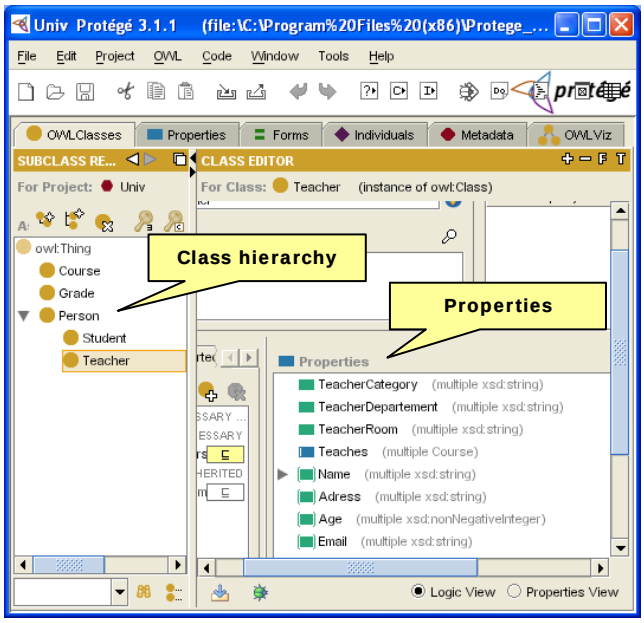

Figure 14-3. Classes and properties

5. USING JENA

Jena is a framework for building Semantic Web applications. It provides a programmatic environment for RDF, RDFS and OWL. It also includes a rule-based inference engine. Jena is open source and is a development effort of the HP Labs Semantic Web Research program. HP Labs have made considerable investments in Semantic Web research since 2000 which lead to the development of standards (such as RDF and OWL) and semantic applications (such as Jena).

The Jena toolbox includes a Java programming API that gives a framework to program semantic Web applications. The API is divided into five sets of functions that deal with the processing of ontologies, namely:

- Processing and manipulation of RDF data models
- Processing and manipulation of ontologies
- SPARQL query support

- Inference on OWL and RDFS data models
- Persistence of ontologies to databases

In this chapter we will focus primarily on the API responsible for the processing and manipulation of OWL ontologies.

5.1 Installing Jena

To install Jena the first step is to download Jena API from http://jena.sourceforge.net. The version used for the examples shown in this chapter was Jena 2.3. Once you have downloaded Jena (in our case the package was named `Jena 2.3.zip`), you need to extract the zip file.

You will find in the /lib directory all the libraries needed to use the Jena API. To develop semantic applications with Java you will need to update your CLASSPATH to include the following libraries:

- `antlr-2.7.5.jar`
- `arq.jar`
- `commons-logging.jar`
- `concurrent.jar`
- `icu4j_3_4.jar`
- `jakarta-oro-2.0.8.jar`
- `jena.jar`
- `jenatest.jar`
- `junit.jar`
- `log4j-1.2.12.jar`
- `stax-1.1.1-dev.jar`
- `stax-api-1.0.jar`
- `xercesImpl.jar`
- `xml-apis.jar`

5.2 Creating an Ontology Model

The main Java class that represents an ontology in memory is the OntModel.

```
OntModel model;
```

In Jena, ontology models are created using the `ModelFactory` class. A model can be dynamically created by calling the `createOntologyModel()` method.

```
OntModel m = ModelFactory.createOntologyModel();
```

When creating an ontology it is possible to describe its characteristics, such as the ontology language used to model the ontology, the storage scheme and the reasoner.

To describe specific characteristics of an ontology, the method `createOntologyModel(OntModelSpec o)` needs to be called and accepts a parameter of the type `OntModelSpec`. For example, `OntModelSpec.OWL_DL_MEM` determines that the ontology to be created will have an OWL DL model and will be stored in memory with no support for reasoning. Various other values are available. Table 14-1 illustrates some of the possibilities.

Table 14-1. Types of ontology models with Jena

Field	Description
`DAML_MEM`	A simple DAML model stored in memory with no support for reasoning
`DAML_MEM_RDFS_INF`	A DAML model stored in memory with support for RDFS inference
`OWL_LITE_MEM`	A simple OWL Lite model stored in memory with no support for reasoning
`OWL_MEM_RULE_INF`	A OWL Lite model stored in memory with support for OWL rules inference
`RDFS_MEM`	A simple OWL Lite model stored in memory with no support for reasoning

More than 20 different ontology models can be created. The following segment of code illustrates how to create an OWL ontology model, stored in memory, with no support for reasoning.

```
import com.hp.hpl.jena.ontology.OntModel;
import com.hp.hpl.jena.ontology.OntModelSpec;
import com.hp.hpl.jena.rdf.model.ModelFactory;

public class CreateModel
  public static void main(String[] args) {
    OntModel model = ModelFactory.createOntologyModel(
                                  OntModelSpec.OWL_MEM);
  }
```

```
}
```

5.3 Reading an Ontology Model

Once we have an ontology model, we can load an ontology. Ontologies can be loaded using the `read` method which can read an ontology from an URL or directly from an input stream.

```
read(String url)
read(InputStream reader, String base)
```

In the following example, we show a segment of code that creates an OWL ontology model in memory and loads the University ontology from the URL `http://dme.uma.pt/jcardoso/owl/University.owl`.

```
OntModel model = ModelFactory.createOntologyModel(
                                        OntModelSpec.OWL_MEM);
model.read("http://dme.uma.pt/jcardoso/owl/University.owl");
```

For performance reasons, it is possible to cache ontology models locally. To cache a model, it is necessary to use a helper class that manages documents (`OntDocumentManager`), allowing subsequent accesses to an ontology to be made locally. The following example illustrates how to add an entry for an alternative copy of an OWL file with the given OWL URI. An alternative copy can be added by calling the method `addAltEntry`.

```
import com.hp.hpl.jena.ontology.OntDocumentManager;
import com.hp.hpl.jena.ontology.OntModel;
import com.hp.hpl.jena.ontology.OntModelSpec;
import com.hp.hpl.jena.rdf.model.ModelFactory;

public class CacheOntology {
  public static void main(String[] args) {
    OntModel m = ModelFactory.createOntologyModel(
                                    OntModelSpec.OWL_MEM);
    OntDocumentManager dm = m.getDocumentManager();
    dm.addAltEntry(
              "http://dme.uma.pt/jcardoso/owl/University.owl",
              "file:///c:/University.OWL");
```

```
    m.read("http://dme.uma.pt/jcardoso/owl/University.owl");
  }
}
```

Since we specify that a local copy of our University ontology exists in `file:///c:/University.OWL`, Jena can load the ontology from the local copy instead of loading it from the URL.

5.4 Manipulating Classes

OWL ontology classes are described using the `OntClass` Java class. To retrieve a particular class from an ontology we can simply use the method `getOntClass(URI)` from the `OntModel` or, alternatively, it is possible to use the `listClasses()` method to obtain a list of all the classes of an ontology. The class `OntClass` allows us to retrieve all the subclasses of a class using the method `listSubClasses()`. For example, the following segment of code allows listing of all the subclasses of the class `#Person` of our University ontology.

```
String baseURI=
            "http://dme.uma.pt/jcardoso/owl/University.owl#";

OntModel model = ModelFactory.createOntologyModel(
                                        OntModelSpec.OWL_MEM);

model.read(“http://dme.uma.pt/jcardoso/owl/University.owl");
OntClass p = model.getOntClass(baseURI+"Person");
for(ExtendedIterator i=p.listSubClasses(); i.hasNext();)
{
    OntClass Class=(OntClass)i.next();
    System.out.println(Class.getURI());
}
```

In our scenario the output of this example is:

```
http://dme.uma.pt/jcardoso/owl/University.owl#Student
http://dme.uma.pt/jcardoso/owl/University.owl#Teacher
```

The `createClass` method can be used to create a new class. For example we can create the new class `#Researcher` and set as superclass the class `#Person` from the previous example,

```
OntClass p = model.getOntClass(baseURI+"Person");
OntClass r = model.createClass(baseURI+"Researcher");
r.addSuperClass(p)
```

The class `OntClass` has several methods available to check the characteristics of a class. All these methods return a Boolean parameter. Some of these methods are illustrated in table 14-2.

Table 14-2. Methods to check the characteristics of an OntClass object

`isIntersectionClass()`	`isComplementClass()`
`isRestriction()`	`hasSuperClass()`

5.5 Manipulating Properties

With Jena, properties are represented using the class `OntProperty`. Two types of OWL properties exist:

- Datatype Properties are attributes of a class. These types of properties link individuals to data values and can be used to restrict an individual member of a class to RDF literals and XML Schema datatypes.
- Object Properties are relationships between classes. They link individuals to individuals. They relate an instance of one class to an instance of another class.

It is possible to dynamically create new properties. The `OntModel` class includes the method `createXXX()` to create properties (and classes as we have already seen previously). As an example, the following code creates a new class named `#Project` and an `ObjectProperty` named `#ProjectOwner`. Using the `setRange` and `setDomain` methods of the class `ObjectProperty` we set the domain of the new property to `#Project` and its range to `#Person`.

```
...
OntClass p=model.createClass(BaseUri +"#Project");
ObjectProperty po=
```

```
          model.createObjectProperty(BaseUri+"#ProjectOwner");
  po.setRange(model.getResource(BaseUri+"#Person"));
  po.setDomain(p);
  ...
```

A DatatypeProperty can be created in the same way, but using the `createDatatypeProperty` method, i.e.

```
  DatatypeProperty p=
    model.createDatatypeProperty(BaseUri+"#ProjectDate");
```

The class `OntProperty` has several methods available to check the characteristics of a Property. All these methods return a Boolean parameter. For example,

Table 14-3. Methods to check the characteristics of an OntProperty object

`isTransitiveProperty()`	`isSymmetricProperty()`
`isDatatypeProperty()`	`isObjectProperty()`

The following segment of code can be used to list the properties of a class. Basically the `listDeclaredProperties()` from the class `OntClass` needs to be called.

```
  import com.hp.hpl.jena.ontology.OntClass;
  import com.hp.hpl.jena.ontology.OntModel;
  import com.hp.hpl.jena.ontology.OntModelSpec;
  import com.hp.hpl.jena.rdf.model.ModelFactory;
  import com.hp.hpl.jena.util.iterator.ExtendedIterator;

  public class ListProperties {
    public static void main(String[] args) {
      String baseURI=
               "http://dme.uma.pt/jcardoso/owl/University.owl#";

      OntModel model = ModelFactory.createOntologyModel(
                                          OntModelSpec.OWL_MEM);

      model.read(
               "http://dme.uma.pt/jcardoso/owl/University.owl");

      OntClass cls = model.getOntClass(baseURI+"Person");
```

```
      System.out.println("Class:");
      System.out.println("  "+cls.getURI());
      System.out.println("Properties:");
      for(ExtendedIterator j=cls.listDeclaredProperties();
                                                  .hasNext();)
      {
          System.out.println("  "+(OntProperty)j.next());
      }
    }
  }
```

The output of executing this example is:

```
Class:
  http://dme.uma.pt/jcardoso/owl/University.owl#Person
Properties:
  http://dme.uma.pt/jcardoso/owl/University.owl#Age
  http://dme.uma.pt/jcardoso/owl/University.owl#Address
  http://dme.uma.pt/jcardoso/owl/University.owl#Email
  http://dme.uma.pt/jcardoso/owl/University.owl#Name
```

`#Age`, `#Address`, `#Email`, and `#Name` are properties of the class `#Person`.

5.6 Manipulating Instances

Instances, also known as individuals of classes, are represented through the class `Instance`. Having a class `OntClass` it is possible to list all its instances using the method `listInstances()`. A similar method exists in the class `OntModel` but is named `listIndividuals()`. For example, the following segment of code lists all the individuals of the University ontology,

```
import com.hp.hpl.jena.ontology.Individual;
import com.hp.hpl.jena.ontology.OntModel;
import com.hp.hpl.jena.ontology.OntModelSpec;
import com.hp.hpl.jena.rdf.model.ModelFactory;
import com.hp.hpl.jena.util.iterator.ExtendedIterator;

public class ListInstances {
  public static void main(String[] args) {
```

```
    OntModel model = ModelFactory.createOntologyModel(
                                        OntModelSpec.OWL_MEM);

    model.read(
             "http://dme.uma.pt/jcardoso/owl/University.owl");
    for(ExtendedIterator i= model.listIndividuals();
                                                .hasNext();)
    {
      System.out.println(((Individual)i.next()).toString());
    }
  }
}
```

The output of executing this example is:

```
http://dme.uma.pt/jcardoso/owl/University.owl#Adelia
http://dme.uma.pt/jcardoso/owl/University.owl#Fatima
http://dme.uma.pt/jcardoso/owl/University.owl#Carolina
http://dme.uma.pt/jcardoso/owl/University.owl#ASP
http://dme.uma.pt/jcardoso/owl/University.owl#SD
http://dme.uma.pt/jcardoso/owl/University.owl#CF
http://dme.uma.pt/jcardoso/owl/University.owl#Grade_1
http://dme.uma.pt/jcardoso/owl/University.owl#Grade_3
http://dme.uma.pt/jcardoso/owl/University.owl#Grade_2
http://dme.uma.pt/jcardoso/owl/University.owl#IC
http://dme.uma.pt/jcardoso/owl/University.owl#JC
http://dme.uma.pt/jcardoso/owl/University.owl#RF
```

To list all the individuals of the class `#Student`, we can add the following lines of code to the previous example:

```
OntClass Student = model.getOntClass(
     "http://dme.uma.pt/jcardoso/owl/University.owl#Student");
for(ExtendedIterator   i= Student.listInstances();i.hasNext();)
{
    System.out.println(((Individual)i.next()).toString());
}
```

Now we can create instances dynamically. The following example creates an instance `#Jorge` of type `#Teacher` and set the name and

e-mail of the instance `#Jorge` to "Jorge Cardoso" and jcardoso@uma.pt, respectively.

```
Resource tClass=model.getResource(baseURI+"#Teacher");
Individual teacher=
             model.createIndividual(baseURI+"#Jorge",tClass);
DatatypeProperty name =
             model.getDatatypeProperty(baseURI+"#Name");
teacher.addProperty(name,"Jorge Cardoso");
DatatypeProperty email =
                 model.getDatatypeProperty(baseURI+"#Email");
teacher.addProperty(email,"jcardoso@uma.pt");
```

5.7 Queries with Jena

One task that is particularly useful once an ontology is available, is to query its data. An OWL knowledge base can be queried using API function calls or using RDQL (RDF Data Query Language). Jena's built-in query language is RDQL, a query language for RDF. While not yet a formally established standard, (it was submitted in January 2004), RDQL is commonly used by many RDF applications. RDQL has been designed to execute queries in RDF models, but it can be used to query OWL models since their underlying representation is RDF. It is a very effective way of retrieving data from an RDF model.

5.7.1 RDQL Syntax

RDQL's syntax is very similar to SQL's syntax. Some of their concepts are comparable and will be well-known to people that have previously worked with relational database queries. A simple example of a RDQL query structure is,

```
SELECT variables
WHERE conditions
```

Variables are represented with a question mark followed by the variable name (for example: ?a, ?b). Conditions are written as triples (Subject Property Value) and delimited with "<" and ">". RDQL allows us to search within a RDF graph to find subgraphs that match some patterns of RDF node triples.

Using our University ontology, we can inquire about the direct subclasses of the class `#Person`. This can be achieved with the following RDQL query:

```
SELECT ?x WHERE (?x <rdfs:subClassOf> <univ:Person>)
USING rdfs FOR <http://www.w3.org/2000/01/rdf-schema#>
      univ FOR
              http://dme.uma.pt/jcardoso/owl/University.owl#>
```

The ?x in this query is a variable representing something that we want of the query. The query engine will try to substitute a URI value for ?x when it finds a subclass of `#Person`. The “rdfs” and “univ” prefixes make the URIs in the query shorter and more understandable. Executing the above query to the University ontology illustrated in Figure 14-1 we expected to retrieve two URIs. One corresponding to the `#Student` concept and the other to the concept `#Teacher`, i.e.

```
<http://dme.uma.pt/jcardoso/owl/University.owl#Student>
<http://dme.uma.pt/jcardoso/owl/University.owl#Teacher>
```

RDQL allows complex queries to be expressed succinctly, with a query engine performing the hard work of accessing the data model. Sometimes, not every part of the ontology structure is known. For example, if we wish to inquire about the list of courses that a student has enrolled for. Since we do not know all the URIs, we have to use variables to represent the unknown items in the query. For instance, “Show me all Y where Y is a “Course”, X is a “Student”, X is named “Adelia Gouveia”, and X studies Y.” The response will list all the possible values for Y that would match the desired properties. The query for this question would be,

```
SELECT ?y
WHERE (?x <univ:Name> "Adelia Gouveia"^^xsd:string),
      (?x <univ:Studies> ?y)
USING univ FOR
             <http://dme.uma.pt/jcardoso/owl/University.owl#>
```

We can also ask for all the students that have passed courses with a grade higher than 12,

```
SELECT ?x,?c
```

```
WHERE (?x <univ:HasGrade> ?y),
      (?x <univ:Studies> ?c),
      (?y <univ:Value> ?z) AND ?z>12
USING univ FOR
             <http://dme.uma.pt/jcardoso/owl/University.owl#>
```

5.7.2 RDQL and Jena

Jena's `com.hp.hpl.jena.rdql` package contains all of the classes and interfaces needed to use RDQL in a Java application.

```
import com.hp.hpl.jena.rdql;
```

Jena's RDQL is implemented as an object called `Query`. To create a query it is sufficient to put the RDQL query in a `String` object, and pass it to the constructor of `Query`,

```
String queryString =”...”;
Query query = new Query(queryString);
```

The method `setSource` of the object `Query` must be called to explicitly set the ontology model to be used as the source for the query (the model can alternatively be specified with a FROM clause in the RDQL query.)

```
query.setSource(model);
```

Once a `Query` is prepared, a `QueryEngine` must be created and the query can be executed using the `exec()` method. The `Query` needs to be passed to the `QueryEngine` object, i.e.

```
QueryEngine qe = new QueryEngine(query);
```

The results of a query are stored in a `QueryResult` object.

```
QueryResults results = qe.exec();
```

Once we have the results of a RDQL query, a practical object that can be used to display the results in a convenient way is to use the `QueryResultsFormatter` object.

```
QueryResultsFormatter formatter =
          new QueryResultsFormatter((QueryResults) results );
formatter.printAll(new PrintWriter(System.out));
```

An alternative to using the `QueryResultsFormatter` object is to iterate through the data retrieved using an iterator. For example,

```
QueryResults result = new QueryEngine(query).exec();
for (Iterator i = result; i.hasNext();) {
    System.out.println(i.next());
}
```

With RDQL it is possible to inquire about the values that satisfy a triple with a specific subject and property. To run this query in Jena, the University ontology is loaded into memory. The query is executed using the static `exec` method of Jena's `Query` class and the results are processed. For example, the following segment of code retrieves all the RDF triples of an ontology.

```
import java.util.Iterator;

import com.hp.hpl.jena.ontology.OntModel;
import com.hp.hpl.jena.ontology.OntModelSpec;
import com.hp.hpl.jena.rdf.model.ModelFactory;
import com.hp.hpl.jena.rdql.Query;
import com.hp.hpl.jena.rdql.QueryEngine;
import com.hp.hpl.jena.rdql.QueryResults;

public class RDQL {
  public static void main(String[] args) {
    OntModel model = ModelFactory.createOntologyModel(
                                        OntModelSpec.OWL_MEM);
    model.read(
             "http://dme.uma.pt/jcardoso/owl/University.owl");

    String sql= "SELECT ?x,?y,?z WHERE (?x ?y ?z)";
    Query query=new Query(sql);
    query.setSource(model);
    QueryResults result = new QueryEngine(query).exec();
    for (Iterator i = result; i.hasNext();) {
        System.out.println(i.next());
```

```
    }
  }
};
```

5.8 Inference and Reasoning

Inference engines, also called reasoners, are software applications that derive new facts or associations from existing information. Inference and inference rules allow for deriving new data from data that is already known. Thus, new pieces of knowledge can be added based on previous ones. By creating a model of the information and relationships, we enable reasoners to draw logical conclusions based on the model. For example, with OWL it is possible to make inferences based on the associations represented in the models, which primarily means inferring transitive relationships. Nowadays, many inference engines are available.

- Jena reasoner – Jena includes a generic rule based inference engine together with configured rule sets for RDFS and for OWL.
- Jess – Using Jess (Gandon and Sadeh 2003) it is possible to build Java software that has the capacity to "reason" using knowledge supplied in the form of declarative rules. Jess has a small footprint and it is one of the fastest rule engines available. It was developed at Carnegie Melon University.
- SWI-Prolog Semantic Web Library – Prolog is a natural language for working with RDF and OWL. The developers of SWI-Prolog have created a toolkit for creating and editing RDF and OWL applications, as well as a reasoning package (Wielemaker 2005).
- FaCT++ – This system is a Description Logic reasoner, which is a re-implementation of the FaCT reasoner. It allows reasoning with the OWL language (FaCT 2005).

In the following sections we will concentrate our attention on using the Jena rule based inference engine programmatically.

5.8.1 Jena Reasoners

The Jena architecture is designed to allow several inference engines to be used with Jena. The current version of Jena includes five predefined reasoners that can be invoked, namely:

- **Transitive reasoner**: A very simple reasoner which implements only the transitive and symmetric properties of `rdfs:subPropertyOf` and `rdfs:subClassOf`.
- **DAML micro reasoner**: A DAML reasoner which provides an engine to legacy applications that use the DAML language.
- **RDFS rule reasoner**: A RDFS reasoner that supports most of the RDFS language.
- **Generic rule reasoner**: A generic reasoner that is the basis for the RDFS and OWL reasoners.
- **OWL reasoners**: OWL rule reasoners are an extension of the RDFS reasoner. They exploit a rule-based engine for reasoning. OWL reasoners supports OWL Lite plus some of the constructs of OWL Full.

In this section we will study how to develop Java applications using the OWL reasoning engines since OWL is becoming the most popular language on the semantic Web compared to DAML and RDFS.

5.8.2 Jena OWL Reasoners

Jena provides three internal reasoners of different complexity: OWL, OWL Mini, and OWL Micro reasoners. They range from the simple Micro reasoner with only domain-range and subclass inference, to a complete OWL Lite reasoner.

The current version of Jena (version 2.3) does not fully support OWL yet. It can understand all the syntax of OWL, but cannot reason in OWL Full. Jena supports OWL Lite plus some constructs of OWL DL and OWL Full, such as `owl:hasValue`. Some of the important constructs that are not supported in Jena include `owl:complementOf` and `owl:oneOf`. Table 14-4 illustrates the OWL constructs supported by the reasoning engines available.

Table 14-4. Jena reasoning support

OWL Construct	Reasoner
rdfs:subClassOf, rdfs:subPropertyOf, rdf:type	all
rdfs:domain, rdfs:range	all
owl:intersectionOf	all
owl:unionOf	all
owl:equivalentClass	all
owl:disjointWith	full, mini
owl:sameAs, owl:differentFrom, owl:distinctMembers	full, mini

owl:Thing	all
owl:equivalentProperty, owl:inverseOf	all
owl:FunctionalProperty, owl:InverseFunctionalProperty	all
owl:SymmeticProperty, owl:TransitiveProperty	all
owl:someValuesFrom	full, (mini)
owl:allValuesFrom	full, mini
owl:minCardinality, owl:maxCardinality, owl:cardinality	full, (mini)
owl:hasValue	all
owl:complementOf	none
owl:oneOf	none

For a complete OWL DL reasoning it is necessary to use an external DL reasoner. The Jena DIG interface makes it easy to connect to any reasoner that supports the DIG standard. By communicating with other ontology processing systems, such as RACER or FAcT, Jena can enhance its ability for reasoning in large and complex ontologies.

5.8.3 Programming Jena reasoners

Given an ontology model, Jena's reasoning engine can derive additional statements that the model does not express explicitly. Inference and inference rules allow for deriving new data from data that is already known. Thus, new pieces of knowledge can be added based on previous ones. By creating a model of the information and relationships, we enable reasoners to draw logical conclusions based on the model.

As we have already done previously, the first step to develop a semantic Web application with support for reasoning is to create an ontology model,

```
String baseURI=
            "http://dme.uma.pt/jcardoso/owl/University.owl#";

OntModel model = ModelFactory.createOntologyModel(
                                        OntModelSpec.OWL_MEM);

model.read("http://dme.uma.pt/jcardoso/owl/University.owl");
```

The main class to carry our reasoning is the class `Reasoner`. This class allows us to extract knowledge from an ontology. Jena provides several reasoners to work with different types of ontology. Since in

our example we want to use our OWL University ontology, we need to obtain an OWL reasoner. This reasoner can be accessed using the `ReasonerRegistery.getOWLReasoner()` method call, i.e.,

```
Reasoner reasoner = ReasonerRegistry.getOWLReasoner();
```

Other reasoners can be instantiated with a call to the methods `getOWLMicroReasoner()`, `getOWLMiniReasoner()`, `getRDFSReasoner()`, and `getTransitiveReasoner()`.

Once we have a reasoner, we need to bind it to the ontology model we have created. This is achieved with a call to the method `bindSchema`, i.e.,

```
reasoner = reasoner.bindSchema(model);
```

This invocation returns a reasoner which can infer new knowledge from the ontology's rules. The next step is to use the bound reasoner to create an `InfModel` from the University model,

```
InfModel  infmodel=ModelFactory.createInfModel(reasoner,model);
```

Since several Java packages are needed to execute and run the examples that we have given, the following segment shows all the Java code needed to instantiate a reasoner.

```
import com.hp.hpl.jena.ontology.OntModel;
import com.hp.hpl.jena.ontology.OntModelSpec;
import com.hp.hpl.jena.rdf.model.InfModel;
import com.hp.hpl.jena.rdf.model.ModelFactory;
import com.hp.hpl.jena.reasoner.Reasoner;
import com.hp.hpl.jena.reasoner.ReasonerRegistry;

public class InstanciateReasoner {
  public static void main(String[] args) {
    OntModel model = ModelFactory.createOntologyModel(
                                        OntModelSpec.OWL_MEM);

  String BaseUri=
             "http://dme.uma.pt/jcardoso/owl/University.owl";
  model.read(BaseUri);
```

```
  Reasoner reasoner = ReasonerRegistry.getOWLReasoner();
  reasoner=reasoner.bindSchema(model);
  InfModel infmodel
               =  ModelFactory.createInfModel(reasoner,model);
  }
}
```

Once a reasoner is instantiated, one of the first tasks that we can execute is to check for inconsistencies within the ontology data by using the `validate()` method, i.e.,

```
ValidityReport vr = infmodel.validate();
if (vr.isValid()){
  System.out.println("Valid OWL");
}
else {
  System.out.println("Not a valid OWL!");
  for (Iterator i =  vr.getReports(); i.hasNext();){
    System.out.println(i.next());
  }
}
```

This example prints a report if the ontology data is found to be inconsistent. The following output shows the example of a report generated when trying to validate an inconsistent ontology,

```
Not a valid OWL
 - Error ("range check"): "Incorrectly typed literal due to
range (prop, value)"
Culprit=
       http://dme.uma.pt/jcardoso/owl/University.owl#Carolina
Implicated node:
          http://dme.uma.pt/jcardoso/owl/University.owl#Email
Implicated node: 'carolina@uma.pt'
```

The report indicates that the email address (`#Email`) of the individual `#Carolina` has an incorrect type.

One other interesting operation that we can carry out is to obtain information from the ontology. For example, we can retrieve all the

pairs (property, resource) associated with the resource describing the course CS8050, which is defined with ID #CS8050.

```
String BaseUri=
              "http://dme.uma.pt/jcardoso/owl/University.owl";
. . .
Resource res = infmodel.getResource(BaseUri+"#CS");
System.out.println("CS8050 *:");

for (StmtIterator i =
   infmodel.listStatements(res,(Property)null,(Resource)null);
   i.hasNext(); )
{
  Statement stmt = i.nextStatement();
  System.out.println(PrintUtil.print(stmt));
}
```

The output of running the previous example is shown below. To make the output more readable we have replaced the URI http://dme.uma.pt/jcardoso/owl/University.owl with the symbol @ and the URI http://www.w3.org/2001/XMLSchema with the symbol §.

```
CS8050 *:
(@#CS8050 rdf:type @#Course)
(@#CS8050 @#IsStudiedBy @#Adelia)
(@#CS8050 @#CourseName 'Semantic Web'^^§#string)
(@#CS8050 @#IsStudiedBy @#Carolina)
(@#CS8050 @#IsTeachedBy @#IsabelCardoso)
(@#CS8050 rdf:type owl:Thing)
(@#CS8050 rdf:type rdfs:Resource)
. . .
(@#CS8050 owl:sameAs @#CS8050)
```

Instance recognition is another important operation in inference. Instance recognition tests if a particular individual belongs to a class. For example, in our University ontology, #Adelia is known to be an individual of the class #Student and the class #Student is a subclass of the class #Person. One question that can be asked is if #Adelia is recognized to be an instance or individual of the class

#Person, in other words is Adelia a person? This can be asked of the inference model using the contains method, i.e.,

```
Resource r1 = infmodel.getResource(BaseUri+"#Adelia");
Resource r2 = infmodel.getResource(BaseUri+"#Person");

if (infmodel.contains(r1, RDF.type, r2)) {
     System.out.println("Adelia is a Person");
} else {
     System.out.println("Adelia is not a Person");
}
```

Other interesting examples of inference include the use of the transitivity, union, functional, and intersection properties.

5.9 Persistence

As we have seen above, Jena provides a set of methods to load ontologies from files containing information models and instances. Jena can also store and load ontologies from relational databases. Depending on the database management system used, it is possible to distribute stored metadata. While Jena itself is not distributed, by using a distributed database back end, an application may be distributed. Currently, Jena only supports MySQL, Oracle and PostgreSQL. To create a persistent model in a database we can use the ModelFactory object and invoke the createModelRDBMaker method. This method accepts a DBConnection connection object to the database. An object ModelMaker will be created and can subsequently be used to create the model in the database.

For example, to store an existing ontology model in a database we can execute the following segment of code,

```
Class.forName("com.mysql.jdbc.Driver");
String BaseURI=
        "http://dme.uma.pt/jcardoso/owl/University.owl";
DBConnection conn = new DBConnection(
                      "jdbc:mysql://localhost/UnivDB",
                      "mylogin",
                      "mypassword",
                      "MySQL");
ModelMaker maker=ModelFactory.createModelRDBMaker(conn);
Model db=maker.createModel(BaseURI,false);
```

```
db.begin();
db.read(BaseURI);
db.commit();
```

And to read a model from a database we can use the following program,

```
Class.forName("com.mysql.jdbc.Driver");
String BaseURI=
             "http://dme.uma.pt/jcardoso/owl/University.owl";
DBConnection conn = new DBConnection(
                      "jdbc:mysql://localhost/UnivDB",
                      "mylogin", "mypassword", "MySQL");
ModelMaker maker=ModelFactory.createModelRDBMaker(conn);
Model base=maker.createModel(BaseURI, false);
model=ModelFactory.createOntologyModel(
                                    OntModelSpec.OWL_MEM,base);
```

6. QUESTIONS FOR DISCUSSION

Beginner:

1. Identify the main differences between XML and RDF.
2. Install Jena in your computer and create programmatically an OWL ontology describing painters and their paintings. The ontology should be able to represent the following statements: “Painter X has painted the painting Y”, “Painter X was born in W”, and “Painting Y was painted in year Z”.
3. Create several individuals for the Painters ontology. For example: Paul Cezanne, born 1839, Aix-en-Provence, France, painted “Le paysan” and “Le Vase Bleu”; Leonardo da Vinci, born 1452, Vinci, Florence, painted “Mona Lisa” and “The Last Supper”; Michelangelo Buonaroti, born 1475, Florence, painted “Sybille de Cummes” and “Delphes Sylphide”.

Intermediate:

1. Identify the main differences between RDFS and OWL.
2. Write down an RDQL query which retrieves the names of all the painters born in Florence using the ontology created in the previous exercise.

3. Use Jena to execute the previous RDQL query and write down the results of executing the query on the ontology.
4. Make your ontology persistent in a database.

Advanced:

1. Write down and execute an RDQL query which retrieves the paintings Michelangelo Buonaroti painted in 1512 (note: The "Sybille de Cummes" was painted 1512).
2. Validate your model using Jena's inference engine.
3. Why is inference a time consuming operation?

7. SUGGESTED ADDITIONAL READING

- Jena Documentation, http://jena.sourceforge.net/documentation.html. This is a fundamental source of information to start programming with the Jena Framework.
- Antoniou, G. and van Harmelen, F. *A semantic Web primer*. Cambridge, MA: MIT Press, 2004. 238 pp.: This book is a good introduction to Semantic Web languages.
- H. Peter Alesso and Craig F. Smith, *Developing Semantic Web Services*, AK Peters, Ltd, October, 2004, 445 pp.: The book presents a good overview of Semantic Tools in chapter thirteen.

8. REFERENCES

Berners-Lee, T., J. Hendler, et al. (2001). The Semantic Web. Scientific American. **May 2001**.

Berners-Lee, T., J. Hendler, et al. (2001). The Semantic Web: A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities. Scientific American.

Cardoso, J. (2004). Issues of Dynamic Travel Packaging using Web Process Technology. International Conference e-Commerce 2004, Lisbon, Portugal.

Cardoso, J., J. Miller, et al. (2005). Academic and Industrial Research: Do their Approaches Differ in Adding Semantics to Web Services. Semantic Web Process: powering next generation of processes with Semantics and Web services. J. Cardoso and S. A. Heidelberg, Germany, Springer-Verlag. **3387:** 14-21.

Decker, S., S. Melnik, et al. (2000). "The Semantic Web: The Roles of XML and RDF." Internet Computing **4**(5): 63-74.

DL (2005). Description Logics, http://www.dl.kr.org/.

FaCT (2005). FaCT++, http://owl.man.ac.uk/factplusplus/.

Gandon, F. L. and N. M. Sadeh (2003). OWL inference engine using XSLT and JESS, http://www-2.cs.cmu.edu/~sadeh/MyCampusMirror/OWLEngine.html.

Ian Horrocks, Peter F. Patel-Schneider, et al. (2003). SWRL: A Semantic Web Rule Language Combining OWL and RuleML, http://www.daml.org/2003/11/swrl/.

Jasper, R. and M. Uschold (1999). A framework for understanding and classifying ontology applications. IJCAI99 Workshop on Ontologies and Problem-Solving Methods.

Jena (2002). The jena semantic web toolkit, http://www.hpl.hp.com/semweb/jena-top.html, Hewlett-Packard Company.

Jena (2005). Jena - A Semantic Web Framework for Java, http://jena.sourceforge.net/,.

Karvounarakis, G., S. Alexaki, et al. (2002). RQL: a declarative query language for RDF. Eleventh International World Wide Web Conference, Honolulu, Hawaii, USA.

Kumar, A. and B. Smith (2004). On Controlled Vocabularies in Bioinformatics: A Case Study in Gene Ontology. Drug Discovery Today: BIOSILICO. **2:** 246-252.

Lassila, O. and D. McGuinness (2001). "The Role of Frame-Based Representation on the Semantic Web." Linköping Electronic Articles in Computer and Information Science **6**(5).

Meinel, C., H. Sack, et al. (2002). Course management in the twinkle of an eye - LCMS: a professional course management system. Proceedings of the 30th annual ACM SIGUCCS conference on User services, Providence, Rhode Island, USA, ACM Press.

OWL (2004). OWL Web Ontology Language Reference, W3C Recommendation, World Wide Web Consortium, http://www.w3.org/TR/owl-ref/. **2004**.

OWLAPI (2006). "The WonderWeb OLW API, http://sourceforge.net/projects/owlapi."

OWL-S (2004). OWL-based Web Service Ontology. **2004**.

OWLViz (2006). OWL Viz. [Online] Available at http://www.co-ode.org/downloads/owlviz/.

Protégé (2005). Protégé, Stanford Medical Informatics. **2005**.

Protégé-API (2006). The Protégé-OWL API - Programmer's Guide, http://protege.stanford.edu/plugins/owl/api/guide.html.

RDF (2002). Resource Description Framework (RDF), http://www.w3.org/RDF/.

RDFS (2004). RDF Vocabulary Description Language 1.0: RDF Schema, W3C, http://www.w3.org/TR/rdf-schema/.

RDQL (2005). Jena RDQL, http://jena.sourceforge.net/RDQL/.

Roure, D., N. Jennings, et al. (2001). Research Agenda for the Future Semantic Grid: A Future e-Science Infrastructure http://www.semanticgrid.org/v1.9/semgrid.pdf.

Shum, S. B., E. Motta, et al. (2000). "ScholOnto: an ontology-based digital library server for research documents and discourse." International Journal on Digital Libraries **3**(3): 237-248.

SMORE (2006). "SMORE - Create OWL Markup for HTML Web Pages, http://www.mindswap.org/2005/SMORE/."

Swoogle (2005). Search and Metadata for the Semantic Web - http://swoogle.umbc.edu/.

SWOOP (2006). "SWOOP - A Hypermedia-based Featherweight OWL Ontology Editor, www.mindswap.org/2004/SWOOP/."

Wielemaker, J. (2005). SWI-Prolog Semantic Web Library, http://www.swi-prolog.org/packages/semweb.html.