12-20-2009

An ontology-based approach to Automatic Generation of GUI for Data Entry

Fangfang Liu
University of New Orleans

An ontology-based approach to Automatic Generation of GUI for Data Entry

A Thesis

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Master of Science
in
Computer Science

by

Fangfang Liu

B.A. Dalian University of Technology, China, 2006

December 2009

ACKNOWLEDGEMENT

First, I would like to give thanks to my advisor Dr.Tu. Without his help, I can not finish the thesis project and the paper. I appreciate all the ideas and help he offered me.

I also thank my project partner Sirisha Tankasala. With her help and cooperation, we can get some good user interface display for expressing the approach.

I also would like to thank Dr. Roussev , Dr. Richard III to be part of my defense committee members.

Lastly, I would like to thank all my friends and my parents who give me so much love and encouragement.

Table of Contents

LIST OF FIGURES

ABSTRACT

This thesis reports an ontology-based approach to automatic generation of highly tailored GUI components that can make customized data requests for the end users. Using this GUI generator, without knowing any programming skill a domain expert can browse the data schema through the ontology file of his/her own field, choose attribute fields according to business's needs, and make a highly customized GUI for end users' data requests input. The interface for the domain expert is a tree view structure that shows not only the domain taxonomy categories but also the relationships between classes. By clicking the checkbox associated with each class, the expert indicates his/her choice of the needed information. These choices are stored in a metadata document in XML. From the viewpoint of programmers, the metadata contains no ambiguity; every class in an ontology is unique. The utilizations of the metadata can be various; I have carried out the process of GUI generation. Since every class and every attribute in the class has been formally specified in the ontology, generating GUI is automatic. This approach has been applied to a use case scenario in meteorological and oceanographic (METOC) area. The resulting features of this prototype have been reported in this thesis.

Keywords: Ontology, Graphic user interface, OWL/RDF, Jena

CHAPTER 1 INTRODUCTION

When the graphical user interfaces (GUIs) are created for businesses, the communication between the domain experts and the programmers (possibly through analysts) is often an inefficient process if the data items are embedded in complex structures and involved in comprehensive contexts. For example, in the Meteorology and Oceanography (METOC) community [1], weather and ocean data are utilized by vastly diversified user applications. There are a lot of specifications of information exchange between METOC data providers and user applications. As a result, a standard interface is needed to support machine-to-machine requests and responses for a wide range of METOC data including gridded forecast model data, climatology, weather effects data, raw satellite data, space environment and solar, remote-sensed observations, as well as imagery [2].

Ideally, a set of highly customized GUIs would be able to shield the complexity regarding computing and the domain knowledge from the end user who consumes the data. However, in different fields the requirements of data vary dramatically from one type of users such as a ship navigator to another type such as a weather station staff. Furthermore, the data requirements of the same type of users will change when different tasks are carried out. Thus preparing a set of generic, static GUIs will not meet users' needs. On the other hand, developing highly customized GUIs is very costly because it is typically difficult for the GUI programmers to understand the requirements of the domain experts for many different tasks. This is a case of the inefficient business-IT gap between domain experts' demands and IT workers' implementations, which reveals the problem of inefficient

communications between domain experts and programmers.

In my thesis, I report an ontology-based approach to automatic generation of highly tailored GUI components that can make customized data requests for the end users. This approach can extend the domain experts' capability in creating services for end users, and reduce the efforts for the software engineers to get the changing requirements. Essentially, the domain experts have the best understanding of the data and applications of their specialized fields. It is the data formalism (such as XML or JMBL) syntax and the technical data structures that complicate the problem. By definition, the domain experts are most productive in navigating within their own knowledge structure – the domain ontology. Our approach has equipped the domain experts with the capability to construct their customized GUIs for end users.

This thesis is structured as follows: In Chapter 2, it talks about the background of technology I use to build up this application. The concepts and definitions of Semantic web service, Ontology and OWL are presented. Also, the introductions of Jena and Protégé 4 are given to introduce some basic information. In Chapter 3, I present the issue we face in this project, and explain the purpose of my thesis project. Then I describe the design and structure scheme of this implementation. In chapter 4, I use a use case scenario in meteorological and oceanographic (METOC) web application to implement this approach. I primarily use Jena (a Java framework for building Semantic Web application). We tried several ways to reach the final purpose. We have three versions of this application. The first one is in memory version which can generate an entire tree node in the memory and build up a tree view to users. Later, we have the database version to reach persistent storage level.

Then we have the column table version. Only one particular node is displayed according to users' selection. It just shows the relationship of this node. In Chapter 5, I give some comparisons between our approach and two leading ontology products on the market, Protégé and JSpace. Chapter 6 is the conclusion of this paper and some future work is going to be done.

CHAPTER 2 BACKGROUND

In this chapter, I review the concepts of Semantic Web Service and ontology, Jena API (a Java ontology retrieval and management program library) and Protégé (the most popular ontology tool).

2.1 Overview of Semantic Web Service

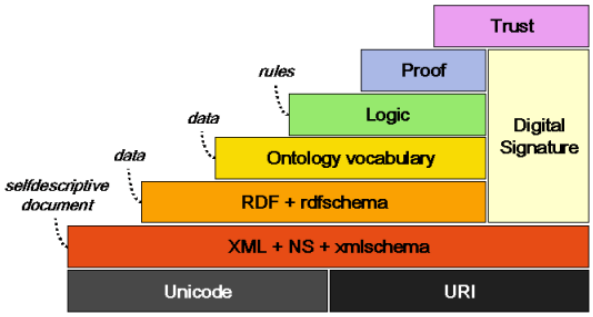
2.1.1 What is Semantic Web Service

The Semantic Web is an evolving extension of the World Wide Web in which the semantics of information and services on the web is defined, making it possible for the web to understand and satisfy the requests of people and machines to use the web content [3]. It can be considered as an intelligent web which can understand human language, making the interaction between human and computers as easy as the conversation between two persons. In the semantic web, the information are given the explicit meaning in order to be searched and collected by machines to present the relationship among several independent objects. For example, you may think there is nothing to do between your bank statement and your calendar. However, in semantic web service, according to the calendar, you can find out the activities you do each day and know why you spend money on that day. Then you won't make any confuse again about a transaction shown in your bank statement but you totally forget. So the biggest advantage of semantic web service is the comprehension and processing of network information. The Semantic Web is about two things. [4] One thing is about regular data formats generation which derives from various sources. However on

original Web, it primarily focuses on the interchange of documents. The second thing is about language for describing how the data relates to real world objects. It can let a person, or a machine not only deal with one dataset but also go through an unending set of data which are connected by semantic terms.

2.1.2 The Hierarchy of the semantic web

Because of the dispersivity in knowledge level and the universality in application level, the semantic web is defined to a unified framework. Berner-Lee mentioned in XML2000 conference about the Architecture of semantic web service in Figure 2-1.



http://www.w3.org/2001/12/semweb-fin/w3csw

Figure 2- 1

First of all, the lowest layer is UNICODE and URI which allows semantic web to find resource in the Internet. URI (Uniform Resource Identifier) can be the unique identifier for each resource on the Internet as a prefix. The second layer is XML to storage the resource in standard format for the representation of arbitrary data structure.The third layer is RDF(Resource Description Framework)，unifying the description method of information in the web. The forth layer is Ontology，which establishes the semantic relationship between the concepts of resources in order to allow each agent or computer program communicate with each other in the semantic level. Especially, the core layers of this entire architecture

are these three levels: XML, RDF and Ontology.

2.1.2.1 RDF

RDF is a W3C standard for representing distributed knowledge based on a decentralized real-world assumption. Any knowledge can be divided into triples (3-tuples) consisting of subject, predicate, and object.

2.1.2.2 OWL(Web Ontology Language)

OWL (Web Ontology Language) is an extension application of RDF, which uses the standard of XML/RDF. It facilitates much better machine interpretability of Web content than that supported by XML and RDF by providing additional abundant vocabulary along with a formal semantics. [5] There are three sublanguages in OWL: OWL Lite, OWL DL, and OWL Full.

OWL Lite supports the basic functionality for the users and gives simple constraints to Ontology. Take cardinality constraints as an example, OWL Lite only allows 0 or 1 as cardinality values.

OWL DL is used to support those need to keep computational completeness and decidability in order to have the stronger expression. OWL DL has more abundant expression capability and it can give more complicated cardinality values other than either 0 or 1 in OWL Lite. Owl DL has a lower formal complexity than OWL Full, but has more restricted syntax than OWL Lite.

OWL Full is the most powerful one of these three sublanguages. It can be used in the situation which needs strongest ability of expression but doesn't really care about the decidability. OWL Full supports those users who don't guarantee calculability, but has the strongest expression and RDF grammar.

2.1.2.3 Ontology

The concept of Ontology is derived from philosophy at the beginning and has become a branch area of philosophy for a long time. Recently, Ontology, this philosophy vocabulary，is defined as a new definition in artificial intelligence of Information Technology. Ontology is a formal explicit specification of a shared conceptualization of a domain of interest. [6] The definition of Ontology includes four levels：Conceptualization、Explicit、Formal and Share. Conceptualization is to extract the model from some phenomenon in the real world, which is independent with any particular environment. Explicit is defined that all of the concepts and restrictions have its own exact meaning. Formal refers that Ontology can be read by computers and also can be processed by computers in its standard format. Share means what the ontology defines is a common object instead of any specific individual. The purpose of Ontology is to capture the common knowledge in the related area, providing the common comprehension of specialized knowledge, confirming common vocabulary and defining explicit relationship between term and vocabulary within multiple levels. In my thesis, the METOC owl file we use is generated by this ontology technique.

2.2 Overview of Jena2

2.2.1 The framework of Jena

In the implementation phrase, we use Jena as the main functional method to achieve the implementation goal. Jena is a java framework for semantic web application which is developed HP Labs（http://www.hpl.hp.com）The framework of Jena contains:

1. Reading and writing RDF in RDF/XML, N3 and N-Triples

In Javadoc, Jena has many details about RDF and Jena RDF APIs, containing introduction about Jena RDF package, discussion about RDF model creation, reading, writing and inquiry.

2. RDFS，OWL，DAML+OIL ontology operation

Jena Framework contains an Ontology Subsystem, the API offered by this system allows different kinds of ontology data which are based on RDF, such as OWL, DAML+OIL and RDFS. There is a good combine between the Ontology API and logic subsystem to retrieve information from particular ontology. Also, Jena provides a function called OntDocumentManager which supports documentation management for ontology import.

3. Data management in Database

Jena 2 allows store data into hard disk or OWL file or related Database to achieve the persistent purpose.

4. Query Model

Jena provides ARQ query engine which implements SPARQL query language and RDQL to support model inquiry. Moreover, putting query engine and database together is to reach the highest efficiency of operating ontology in Database.

5. Rule-based inference

Jena 2 supports some simple rule-based reasoning. The mechanism is implemented by importing inference reasoners into Jena.

2.2.2 Jena Ontology API

Jena 2.4 Ontology API is contained in “com.hp.hpl.jena.ontology” package. Jena provides java interfaces which all can be compiled to .class file. Obviously, class file is not able to distinguish which language Ontology use to create itself, so each Ontology language has it own profile which lists the classes, attributes and URI. For example, in DAML framework, object attribute URI format is daml: ObjectProperty, but in OWL framework it is owl: ObjectProperty. In RDFS, there is no object property, so the URI is null. In this thesis, we only use OWL framework to be the ontology language.

In Jena, the framework is combined with Ontology Model through variable configuration. Ontology Model can inherent from Jena Model Class. Model class allows us to access RDF statement, OntModel extension, in order to support any kind of ontology object: classes, properties and individuals.

There are some java classes or interfaces described as following

1. Ontology Model : OntModel

OntModel is an extension of Jena RDF to provide the capability to process ontology data. First of all, using Jena to operate ontology is to create an ontology model. There are many operations offered by this model to do the job. In Jena, Ontology model defines more than twenty ways of model data implementation and creation. Here we use ModelFactory

to create Ontology Model. There is an example to create an Ontology Model:

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

This statement does not contain any parameter. It creates OntModel by default configuration. In another word, it uses owl language, memory-based and RDFS inference support. Otherwise, OntModel can be created into different models by OntoModelSpec. For example, this is DAML based memory ontology. It just runs in the memory instead of writing back to hard disk or database table.

```
OntModel ontModel =
ModelFactory.createOntologyModel( OntModelSpec.DAML_MEM );
```

The Ontology we use is fetched from OWL file in the disk. We can use Read method to get it.

```
ontModel.read("file:D:/temp/Creatrue/Creature.owl");
```

That is the file path where OntModel can be read. There are many overload read methods. The declaration is as below:

```
read( String url );

read( Reader reader, String base );

read( InputStream reader, String base );
read( String url, String lang );
read( Reader reader, String base, String Lang );
```

```
read( InputStream reader, String base, String Lang );
```

2. Document manager

OntDocumentManager is a class which is used to help Ontology manage documentations including importing Ontology documentation Model, caching and downloading ontology function from Internet. Each Ontology Model is related to a document manager. When an ontology model is created, an independent document manager is passed to ModelFactory as a parameter. It has many options. Every document manager's parameters can be set by java.

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

OntDocumentManager dm = m.getDocumentManager();
```

3. OntClass

This interface defines operations of Ontology. It can return an Iterator through listClasses() method of OntModel. OntClass has each corresponding method for each relevant concept. The Typical method contains adding subclass, adding constraint, returning an Iterator of a particular type and related logical determination.

4. OntResource

In ontology APIs, ontology classes are all inherited from OntResource. So it is possible to let OntResource place public class functions and set general return value. OntResource extends Jena RDF resource interface so that any class which can accept resource or

RDFNode method also can accept OntResource.

2.3 Overview of Protégé

Protégé is a free, open source ontology editor and knowledge-base framework [7] which is developed by people from Stanford University. It uses Java and Open Source as basic platform to create and edit knowledge-based model with ontology. As customer personal needs, Protégé can adapt this new language well with friendly customized user interface. For example, it can let users set up input mode by themselves or it can allow users to convert Protégé internal files into different kinds of file formats such as XML、RDF(S)、OIL、DAML、DAML+OIL、OWL etc. Protégé doesn't have imbedded logic engine, but it has very good expandability to enable some special function expansion through third party plug-ins.

There are some good features of Protégé. First of all, it is open-source also has polymorphism and inheritance to provide basic operations for ontology construction. Secondly, it adopts graphic user interface which has clear modules in each part. Furthermore, Protégé uses tree-extension as its basic ontology data structure to display hierarchy classes. Users can add or edit class, subclass or instance by clicking corresponding items. Therefore, it is easy for users to create and use ontology without learning so much about Ontology Language. That is why Protégé has become one of the most popular Ontology editors. In this thesis, we use Protégé as our Ontology editor to create a METOC OWL file and let Protégé to be compared with our implementation as well.

Protégé has some features as following areas:

- Class modeling: A graphic user interface is provided to present the relationship between class (concepts of particular area) and its attributes.
- Instance editing: protégé can generate the interactive module automatically for users or area experts to enter desired instances.
- Model processing: there is a plug-in library which can help users to do inquiry, define logic behavior and ontology.
- Model exchange: final model containing classes and instances can be loaded or saved as different format such as XML、UML or RDF.

CHAPTER 3 DESIGN

3.1 Problem Statement

In development of application software, the domain knowledge is often the critical key to success. However, the communication between the domain experts and the programmers is often problematic due to the drastic differences in backgrounds, trainings and cultures. The notorious business-IT gap has repeatedly caused difficulties. Ideally, a domain expert can effectively specify his/her needs in his/her domain language such as the ontology in his/her domain. And programmers can deal with the specialized knowledge as little as they can with the help of ontology. In my thesis project, I attempted to pursue the research along this line: to facilitate the domain experts' requirement specification with ontology-based tools. As this ambition is too broad, I narrowed down my research to a simpler task for domain experts: to specify a requirement of a GUI for end users. Furthermore, I have chosen a non-trivial domain: METOC (meteorology and oceanography). The purpose of my project is to demonstrate that this ontology-based highly tailored GUI generator can be an approach for domain experts and programmers to fill up the gap between them. As a result, I have offered an application which can help experts to generate customized GUI for end users.

3.2 Design Overview

My thesis project is an application of the dual ontology approach proposed by Michael Spahn in “Ontology-Based End-User Information Self-Service” [8]. In this paper, Ontology-based End-User Information Self-Service is built on a semantic middleware providing a standard, comprehensible data model in the form of a business level ontology (BO). The BO provides a business-relevant vocabulary that is familiar to business users from their work context and therefore intuitive and easy to understand. Based on the information they provide in BO, domain expert will describe the public or private business process in a semantically rich fashion. [9]

Secondly, the BOs are converted into an IT view which we called Technology level Ontology (TO). TOs represent the taxonomy of the terminologies and reflect a one-to-one mapping from the underlying data schema to the ontology representation formalism. In Figure 3-1, it is a fragment of Ontology file written by OWL.

```
……
<owl:Class rdf:about="#Gridded_Climatology_Request">
<rdfs:subClassOf rdf:resource="#Gridded_Data_Request"/>
</owl:Class>
−<!--
  http://www.semanticweb.org/ontologies/2009/0/Ontology1233418213187.owl
#Gridded_Data_Request
-->
<owl:Class rdf:about="#Gridded_Data_Request">
<rdfs:subClassOf rdf:resource="#Data_Request"/>
<owl:disjointWith rdf:resource="#Observation_Request"/>
<owl:disjointWith rdf:resource="#Remote_Sensed_Data_Request"/>
<owl:disjointWith rdf:resource="#Space_Weather_Request"/>
</owl:Class>
……..
```

Figure 3- 1

The BOs connect various TO terms together, and specify how technical TO terms are related to business-level notions, which is mostly consistent with the typical workflows.

In our thesis METOC data request design, to make the data request for a task, a domain expert first interacts with a BO which is presented in an OWL file, and selects the needed data items from the tree-like presentation. Then based on BO level, they can make their customized GUI.

3.3 API design based on Jena

When we get the OWL file as data model, we use Jena API library to generate Ontology Model, through the interface of which we can refer to the data structure for our later use. The following are the signatures of the methods,

- List all the subclasses of each object :

  Public String[] getSubClasses (OntClass C)

- Get Object Property of each object :

  Public OntProperty[] getObjectProperties(OntClass C)

- Get range classes of a property:

  public OntClass[] getRangeClasses(OntProperty p)

- Return Attributes of a class: public ArrayList getAttributes(OntClass C)
- Get all of the superclasses: public OntClass[] getSuperClasses(OntClass C)
- Get individuals of a class : public ArrayList getIndividual(OntClass C)

3.4 Persistent Layer Design

Due to the difference of ontology storage and relational storage, there are three types of RDF/OWL database storage schema patterns:

1) *Vertical schema*: this schema contains triples for any RDF/OWL. Each row has a record of a RDF/OWL triples. As Figure 3-2 shown, <subject, predicate, object> represents a table which is used in relational database where subject represent a resource with URI , predicate represents a name of this property and object represent a property value. In this pattern, RDF/OWL resource can be easily stored into a relational database. The design is easy and stable. Different resource's properties can be combined together by the same subject of resource URI. However, this schema is quite inefficient when some query should be taken.
2) *Specific Schema*: unlike the previous schema, this one contains multiple tables to store the resources. There are two ways. One is based on ontology class that each class has each own table; the other one is based on property. This schema makes class and property separately. The structure can be seen in Figure 3-3.
3) *Hybrid Schema:* This schema is mixed two previous types of schemas together (Figure 3-4). It will create triples for resource property and also create a table for each class's instance.

For our implementation, we use Jena to implement persistent purpose of OWL file. It permits Ontology file to be saved into an OWL file or a related Database as a persistent storage. Jena supports MySQL、HSQLDB、PostgreSQL、Oracle and Microsoft SQL Server. Unlike previous version of Jena 1, Jena 2 schema trades-off space for time. For relational

databases, it contains a statement table, a literals table and a resources table. The statement table contained all asserted and reified statements and referenced the resources and literals tables for subjects, predicates and objects [10]. To separate relational database references from literals and URIs, column values are encoded with a prefix that indicates which kind of value. A separate literals table is only used to store literal values whose length exceeds a threshold, such as blobs. Similarly, a separate resources table is used to store long URIs. It is possible to perform query operations without a join by storing values directly in the statement table. However, a denormalized schema uses more database space because the same value (literal or URI) is stored repeatedly. There are four steps to create a database model or open an existed model: loading database JDBC driver, creating database connection, creating a ModelMaker for database and creating a model for Ontology.

Triples Form Table

subject	predicate	object
(resource URI)	(property name)	(property value)

Figure 3- 2

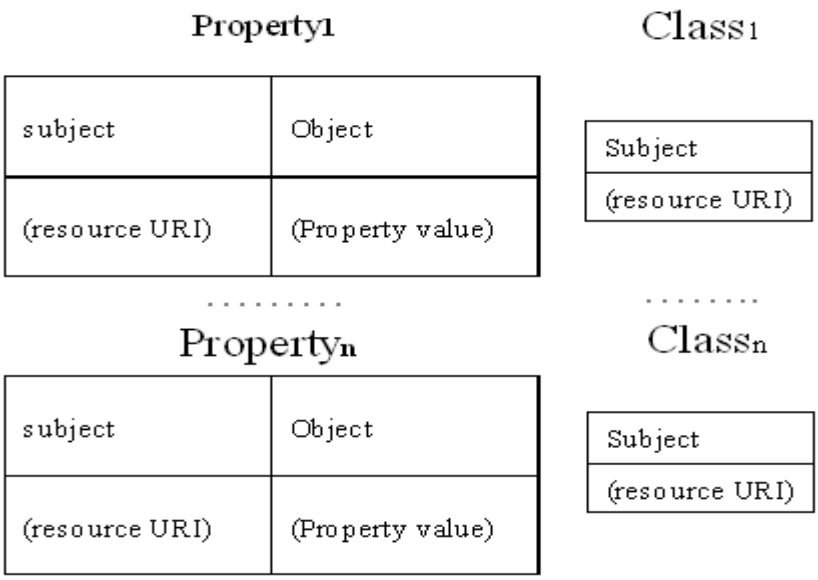


Figure 3- 3

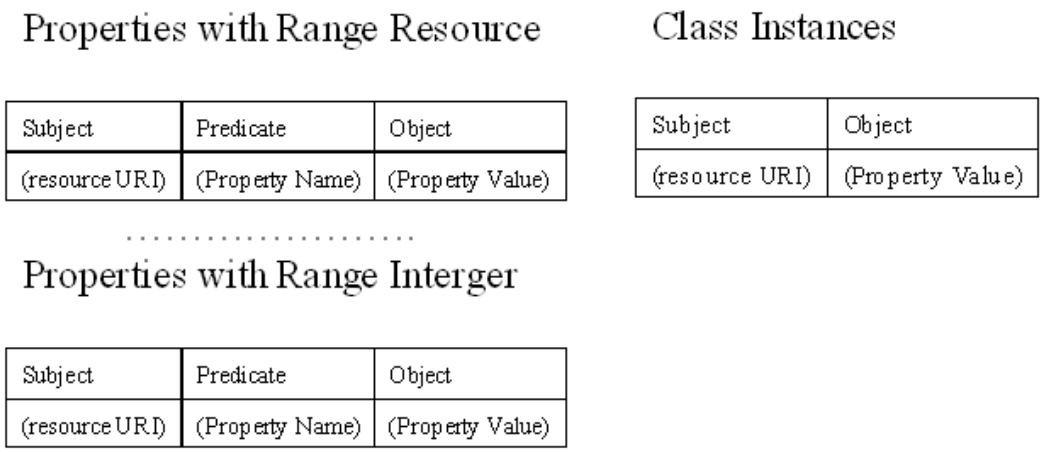


Figure 3- 4

The persistence storage API supports a Fastpath capability for SPARQL queries that dynamically generates SQL queries for SPARQL Basic Graph Patterns to perform many SPARQL queries within the database engine [11].

3.5 SPARQL Query Design

SPARQL query language is an extension of RDQL query language. SPARQL contains capabilities of querying required and optional graph patterns along with their conjunctions and disjunctions. Moreover, Extensible value testing and constraining queries are supported by SPARQL through source RDF graph. The results of SPARQL queries can be results sets or RDF graphs. Jena supports SPARQL query language.

In Jena, SPARQL query language implements inquiry function through matching graphic mode. The simple graphic mode is triples which allow variables match one of three elements in triples. According to this pattern, we can find subject, predicate or object of any triples in RDF/OWL file. SPAQRL query language for Ontology is just like SQL for relational database. In my thesis project, different kinds of SPARQL syntaxes can perform as different function to meet query requirements

Here are some examples used for METOC data request owl file:

1) Find all subclass of a known class

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl:<http://www.w3.org/2002/07/owl#>
SELECT    ?SubClass
    WHERE {?SubClass rdfs:subClassOf rdfs:knownclassname}
```

2) Find which Data time group in METOC data start at date of 2009-10-21

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX DataRequest:
< http://www.semanticweb.org/ontologies/2009/0/Ontology1233418213187.owl>
SELECT    ?DateTimeGroup
    WHERE {? DateTimeGroup DataRequest: start_at DataRequest:2009-10-21}
```

3) Find out names with character “r” or “i” of all processes

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
```

```
PREFIX DataRequest:
< http://www.semanticweb.org/ontologies/2009/0/Ontology1233418213187.owl>
SELECT    ?Process ?name
      WHERE {? Process DataRequest: has_name ?name.
                  FILTER regex(?name, "r", "i")
 }
```

CHAPTER 4 IMPLEMENTATION

4.1 Understanding of Business Processes

In this chapter, I will describe the implementation using an example of a business process of METOC (meteorological and oceanographic) data requests. Due to the complexity of this implementation, I have a partner named Sirisha Tankasala who helped me accomplish the front-end part of this project. My understanding of this METOC data request is represented in a set of object diagrams such as the one shown in Figure 4-1 and Figure 4-2. These diagrams were produced by a true domain expert.

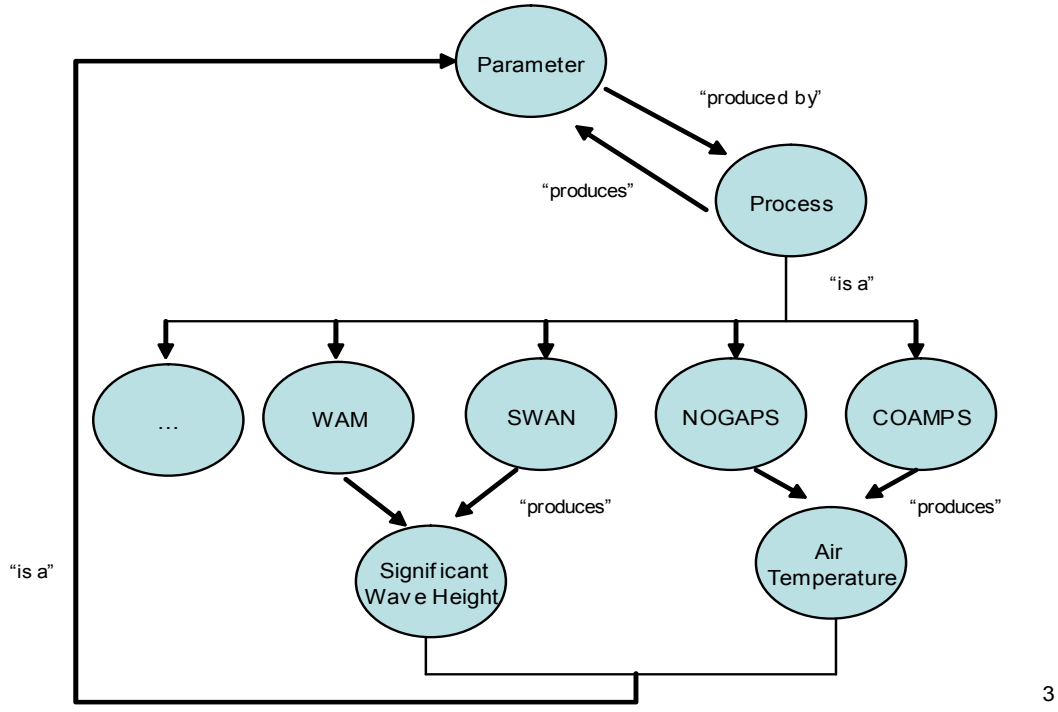


Figure 4- 1

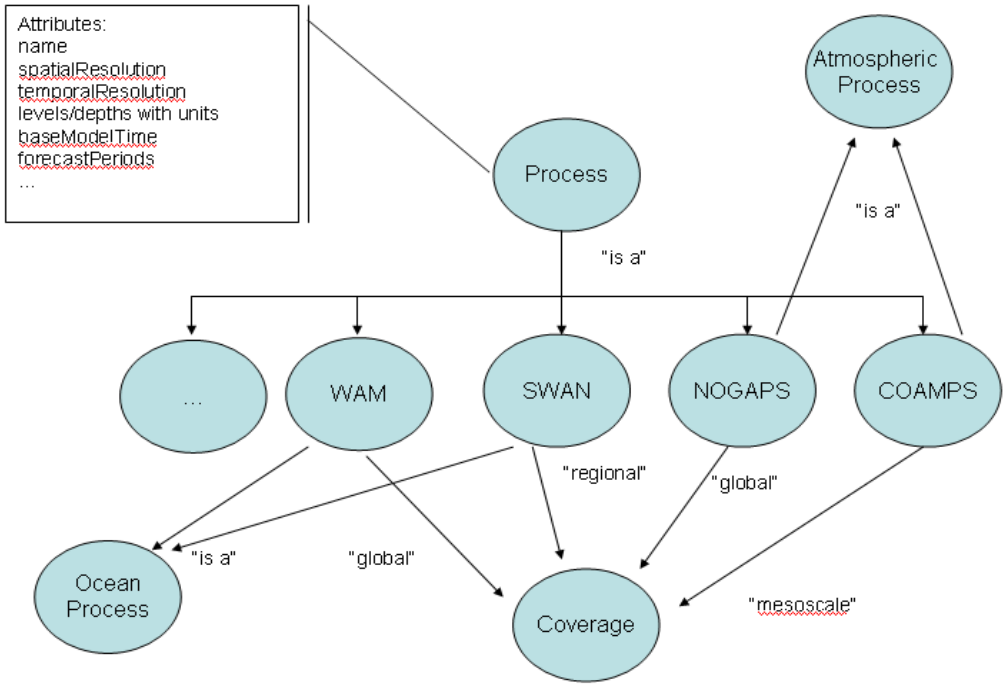


Figure 4- 2

The first step of my task is to depict the relationships given in the object diagrams into Ontology. Naturally, these high-level relationships will fall into the BO category. On the other hand, many terminologies used by the object diagrams are actually referred to TOs in related domains.

4.2 The Ontology and OWL file

I used Protégé to construct an ontology derived from all the object diagrams. The class diagram shown in Figure 4-3 illustrates the top level classes. Figure 4-4 shows the subclass of class data_request. The property diagram shown in Figure 4-5 illustrates the relationships of class Process and Parameter. Using protégé, I exported the ontology into an OWL file. Figure 4-6 shows a fragment of an OWL file generated by Protégé.

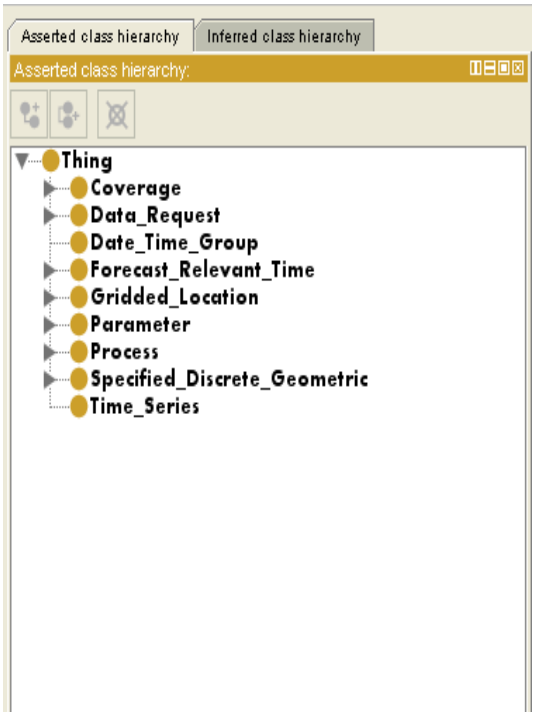


Figure 4- 3

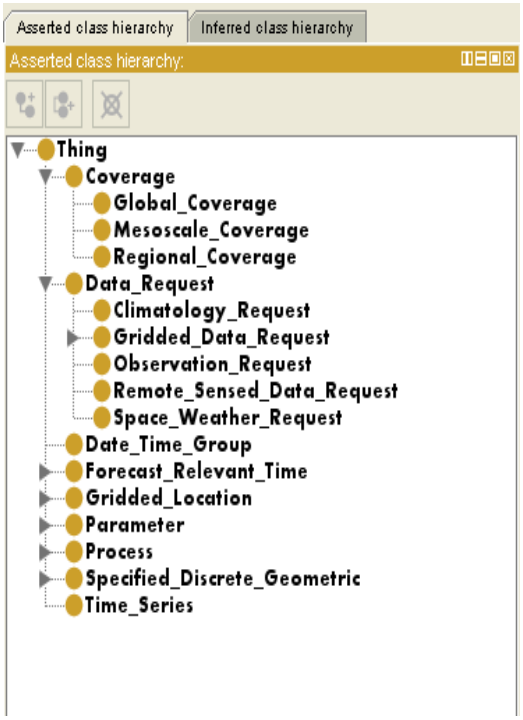


Figure 4- 4

4.3 Implementation of Customized Java library using Jena API

Having obtained the OWL file of ontology, I then implemented a Jena library using the Jena API. The purpose of this customized library is to support our GUI generation process.

Here are some example code fragments.

```
//Generate Ontology Model for access
Model model =
        ModelFactory.createDefaultModel();
FileInputStream file = new FileInputStream(uri);
in = new InputStreamReader(file, "UTF-8");
model.read(in,null);
….

// Generate OntoClass for each Ontology Class in OWL files
public OntClass createOntClass(String name)   {
        String NS=model.getBaseModel().getNsPrefixURI("");
        OntClass r =model.getOntClass( NS + name) ;
        return r;
    }
…..
//get ObjectProperties (relationship) of each ontology class
    public OntProperty[] getObjProperties(OntClass c){
        ArrayList<OntProperty> objProperty= new ArrayList<OntProperty>();
        for(Iterator j=c.listDeclaredProperties();j.hasNext();){ //list all the Object Properties
            OntProperty p=(OntProperty) j.next();
              if(p.isObjectProperty()) {
                objProperty.add(p);
              }
        }
        OntProperty[] OntPro = (OntProperty[])objProperty.toArray(new OntProperty[0]);
        return OntPro;
    }
```

4.4 Generate tree-like view GUI selector

We have two different ways to access the owl file data and load the ontology model. The first one is load from file which is fast to read files directly from the memory. The file path can either be a local directory or an URL address. The other way is to generate the ontology model by loading from a relational database.

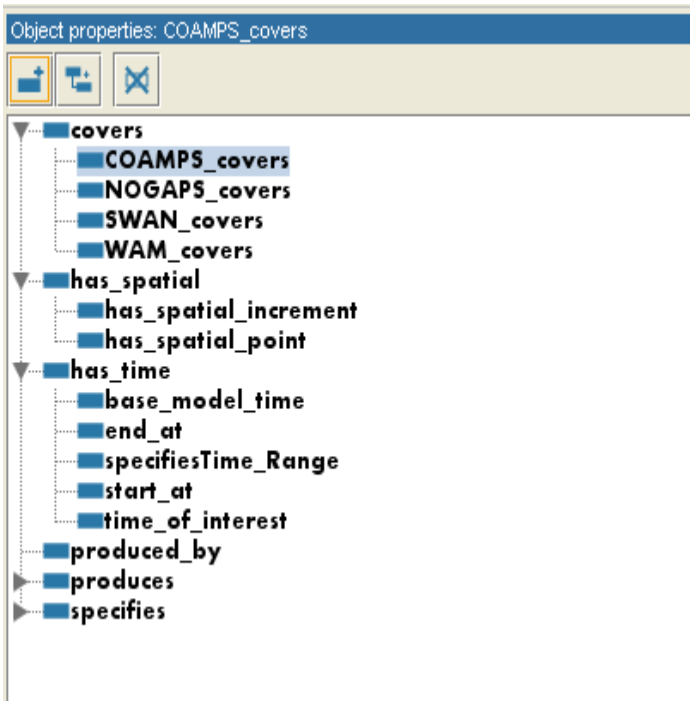


Figure 4- 5

```
    <!-- http://www.semanticweb.org/ontologies/2009/0/Ontology1233418213187.owl#COAMPS_covers -->
     <owl:ObjectProperty rdf:about="#COAMPS_covers">
          <rdfs:domain rdf:resource="#COAMPS"/>
          <rdfs:range rdf:resource="#Mesoscale_Coverage"/>
          <rdfs:subPropertyOf rdf:resource="#covers"/>
     </owl:ObjectProperty>

<!--http://www.semanticweb.org/ontologies/2009/0/Ontology1233418213187.owl#COAMS_produces_air_temperature
-->
     <owl:ObjectProperty rdf:about="#COAMS_produces_air_temperature">
          <rdfs:range rdf:resource="#Air_Temperature"/>
          <rdfs:domain rdf:resource="#COAMPS"/>
          <rdfs:subPropertyOf rdf:resource="#produces"/>
     </owl:ObjectProperty>

  <!-- http://www.semanticweb.org/ontologies/2009/0/Ontology1233418213187.owl#NOGAPS_covers -->
     <owl:ObjectProperty rdf:about="#NOGAPS_covers">
          <rdfs:range rdf:resource="#Global_Coverage"/>
          <rdfs:domain rdf:resource="#NOGAPS"/>
          <rdfs:subPropertyOf rdf:resource="#covers"/>
     </owl:ObjectProperty>
```

Figure 4- 6

4.4.1 Memory view of owl file

In this memory version, we instantiate a model by loading contents from an OWL/RDF file. The OntModel class is an extension of the Jena basic model class using the ontology model recursively. After it is loaded up, the OntoModel contains all the information of this Ontology. We will use a regular Java data structure to convert from OntoModel to a Jtree node. Therefore, this tree node is created beginning with the root node and ending at the leaf nodes. Finally, one Jtree node has the structure of the relationships of all nodes which are originally in OntoModel.

For the front-end display, we use the Jenkov JSP Tree Tag library to display the classes and sub classes in a tree format. The tag library consists of a tree model api, and the tree tags to display the tree model in the JSP page. First of all, a tree instance is created. The tree display contains information about what nodes are expanded and selected. This information is not kept in the tree nodes.

```
ITree tree = new Tree();
OntClass c = onto1.createOntClass(“Data_Request”);
ITreeNode root = new TreeNode(c.getLocalName(), c.getLocalName(), "root");
```

Secondly, a root tree node or the parent node is created and it is passed to the drilldown method where the children are added to the tree.

```
ITreeNode display = drilldown(c,root);
```

The core algorithm contains in method drilldown(`OntClass c,ITreeNode root`) which is to find out all subclasses, Object Properties and Data Properties of each particular

node in OntoModel. SubClasses represent all the children classes under a class. Object Properties represent the relationship between two or several different nodes. And Data Properties represent the attributes of each node. One entire tree node contains all the subclasses and also all the relationships. The advantage of that is the node which has that relationship along with it, makes the view more comprehensive by human beings.

There are two main processes in this method:

1. Show all range classes related to the current node

   a. Fetch the object properties list corresponding to the current OntClass node

```
OntProperty[] properties = onto.getObjProperties(c);
```

   b. If the list is not empty, iterate through the list and for each object get the corresponding range class. The Range Classes and property are added as children to the current node.

```
OntClass[] classes = onto.getRangeClasses(properties[i]);
  if (classes.length != 0)
 {
    for (int j = 0; j < classes.length; j++) {
        if (classes[j] != null && !root.getId().contains(classes[j].getLocalName())) {
            ITreeNode childone = new TreeNode(root.getId()+"/"+classes[j].getLocalName(),
            classes[j].getLocalName(),"rangeClass");
            //childone.setType("rangeClass");
            root.addChild(childone);
            drilldown(classes[j],childone);
        }
    }
}
```

2. show all the subclasses

```
OntClass[] subClasses = onto.getSubClass(c);
```

3. Repeatedly call the drilldown(classes[j], childone); This loop will terminate when the leaf node is reached.

After the entire tree node is generated, we use the tag library to build up a tree view. The tree model is stored as a session attribute, the tags has to know the name of the session attribute to build up the tree. This is an example:

```
<%@ taglib uri="/WEB-INF/treetag.tld" prefix="tree" %>
<tree:tree tree="tree.model">
</tree:tree>
```

The tree tag is responsible for finding the tree and iterating the visible nodes. It doesn't do any layout of the tree by itself, nor display any of the data in the tree model.[12] We have to specify that using other specialized tags placed between the start <tree:tree ...> tag and the end </tree:tree> tag. These nested tags need to know how to find the visible tree nodes as they are iterated by the <tree:tree ...> tag. Therefore we must specify inside the <tree:tree ...> tag where to make the tree nodes available. This is done with the node attribute:

```
<tree:tree tree="tree.model" node="tree.node">
</tree:tree>
```

Now the <tree:tree ...> tag will store the nodes iterated as a request attribute under the key "tree.node". In this way, the entire tree can be displayed. Figure 4-7 shows a resulting tree view.

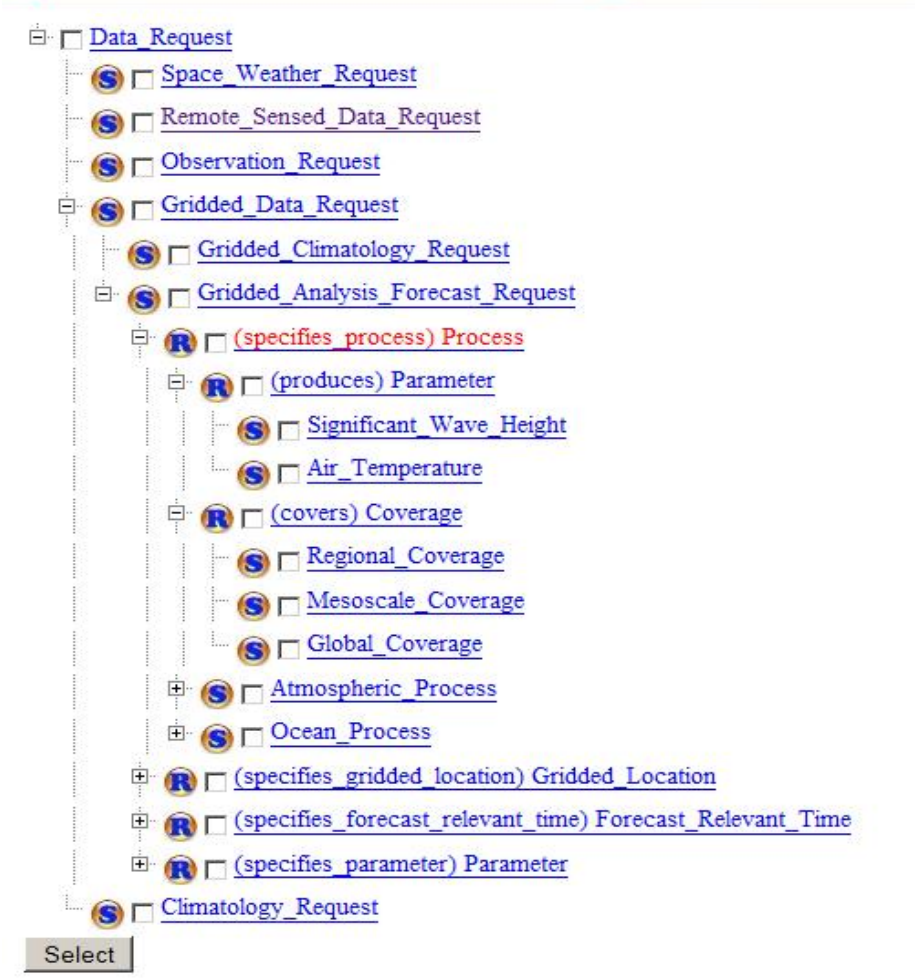


Figure 4- 7

4.4.2 Database persistent view of OWL file

When the ontology is huge that is involved with thousands of classes and instances, it is not a good choice to keep the file in hard disk. So we have to use database as the persistent layer to keep data. Also using this persistent layer of relational database, it offers us an efficient way to use the query language either with general SQL or specific query language of ontology SPARQL. It is according to the Jena2 persistence subsystem, it implements the Model interface to offer persistence for models through use of a back-end relational database engine. The Jena2 database layout uses a denormalized schema where

literals and resource URIs are stored directly in statement tables. The Jena2 layout enables faster insertion and retrieval for fine-grained API operations at the cost of storage over a normalized triples and nodes schema [13]. So besides generating ontology from local file path, there is another way to generate an ontology model by loading from database:

The first one is load the ontology model from local host directory. It generates the model depending on a file path.

```
public static OntModel createDBModelFromFile(IDBConnection con,
                String name,   String filePath) {
        ModelMaker maker = ModelFactory.createModelRDBMaker(con);
        Model base = maker.createModel(name);
        OntModel newmodel = ModelFactory.createOntologyModel(
                getModelSpec(maker), base);
        newmodel.read(filePath);

   return newmodel;
}
```

The second way is to load ontology model from database. We have several tables including a statement table, a literals table and a resources table which allow us to load an ontology model through Jena API.

```
public static OntModel getModelFromDB(IDBConnection con, String name) {
        ModelMaker maker = ModelFactory.createModelRDBMaker(con);
        Model base = maker.getModel(name);
        OntModel newmodel = ModelFactory.createOntologyModel(
        getModelSpec(maker), base);

        return newmodel;
}
```

4.4.3 Column View of OWL file

Compared to the class tree view given by Protégé, an enhancement of our tree is to include the relationships and the nodes in range together in a single tree view. While this addition provides users with handy prompts, it induces loops into the graph. A node and its sub tree may appear twice or even more in a drilldown path. We have applied many ways to truncate redundant path. At the beginning, we have a hashtable to record which node has been visited by the drilldown method. Using this way, we can avoid redundant path. However, this leads to a limitation that one node can only display once in the tree view. Sometimes, a node should be shown in different paths. Then we used a string which contains all the path names from root to the current node. In this way, we did avoid duplicated paths in the tree view. However, when the ontology is large, we encountered the problem of memory outage. So we have also produced a column view as another user interface.

In the column version, we do not generate the whole tree. Only a path is displayed. At any moment, no more than three levels are shown. Initially, we only show the first level of the hierarchy. When the user selects a particular node, it shows the sub-hierarchy of this subclass in the second level column. Upon a selection of a node in the middle column, the sub-hierarchy of the chosen node will be displayed in the third (the right-most) column. If the user selects a node in the right-most column and the chose node is not a leaf node, then the whole display will be shifted toward left. On the other hand, if the user wants to retrospectively review the parent node of the left-most column, the “shift right” button can be clicked.

The demonstration is shown in Figure 4-8 as below:

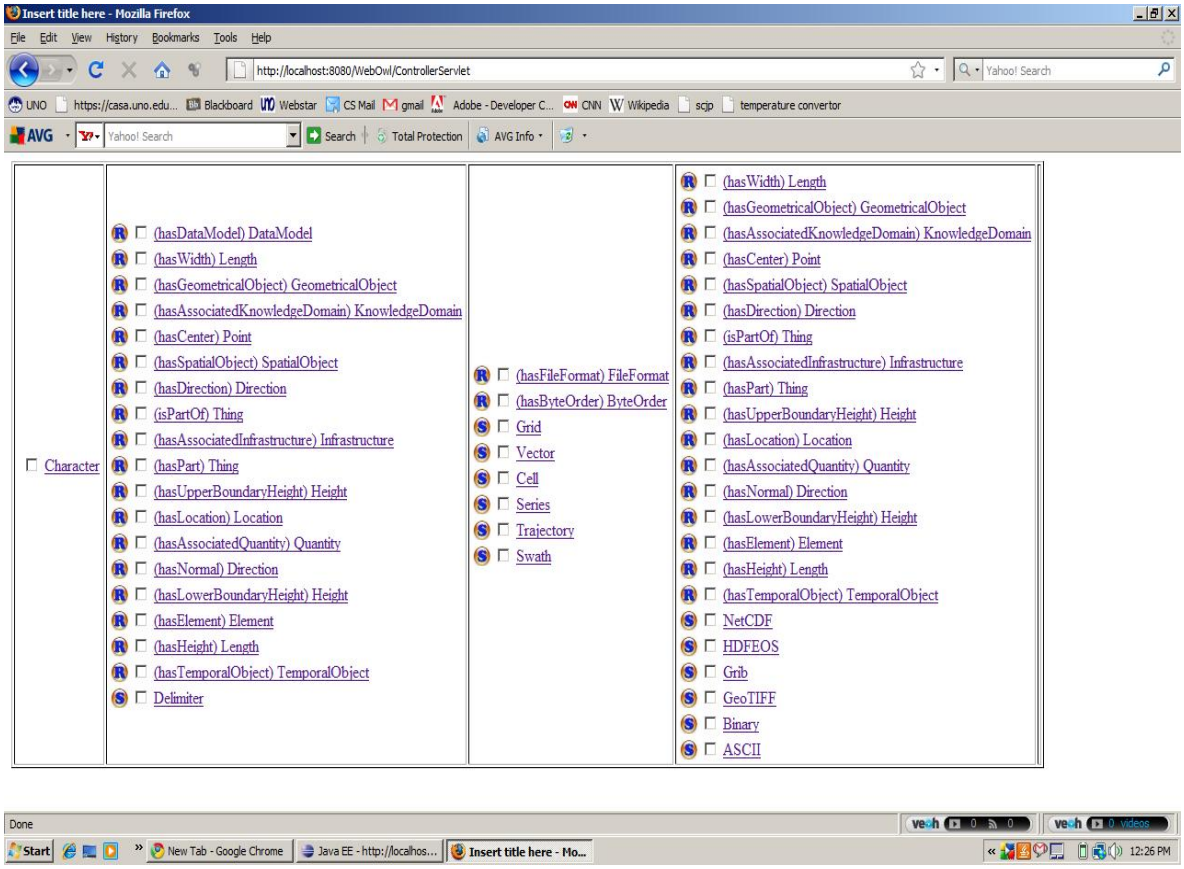


Figure 4- 8

After experts select the nodes, it will generate a customized GUI for end users, which is shown in Figure 4-9.

Typicaly, a task would require data items from many different classes; the generated GUI often contain many panels representing specific criteria for the data request.In our implementation, the GUIs are HTML forms in which every entry is identified by the corresponding ontology class identifier and its attribute name. The HTML forms enable the user to submit inputs to a service that triggers the other request generation.

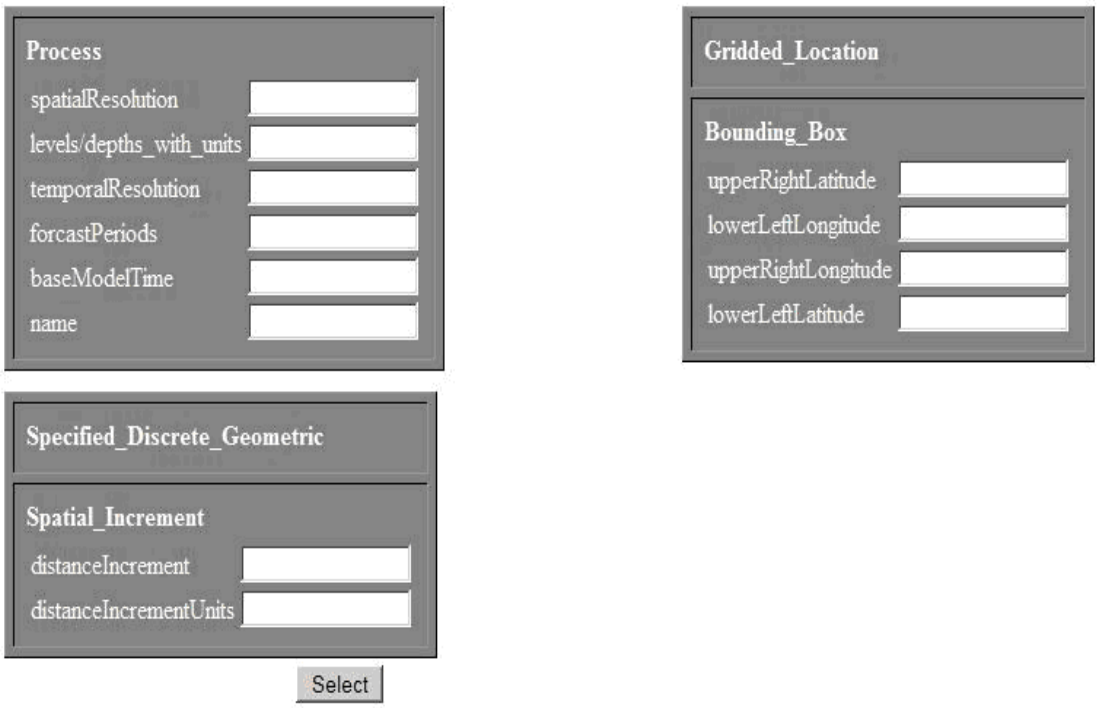

Process
spatialResolution
levels/depths_with_units
temporalResolution
forcastPeriods
baseModelTime
name
Gridded_Location
Bounding_Box
upperRightLatitude
lowerLeftLongitude
upperRightLongitude
lowerLeftLatitude
Specified_Discrete_Geometric
Spatial_Increment
distanceIncrement
distanceIncrementUnits
Select

Figure 4- 9

CHAPTER 5 RELATED WORK

There are a number of ontology-based products. Here I have chosen two popular ones as examples to show the unique features of our implementation.

Protégé, an open resource ontology editor developed by Stanford University is one of the most popular tools used extensively. For comparison, Protégé is not only an ontology editor tool which can help users to generate Ontology file but also can be used as an Ontology viewer to browser Ontology files. In its console window, it has separate views of the hierarchy classes and the hierarchy classes' relationships.

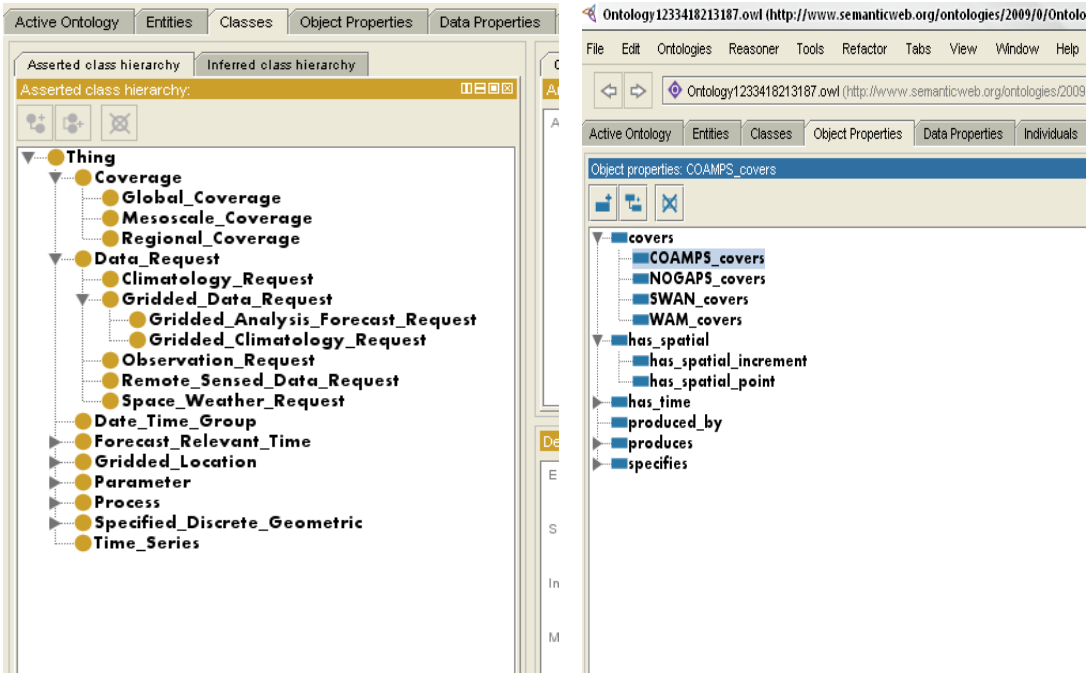


Figure 5- 1

As shown in Figure 5-1, Protégé doesn't have a direct view between each class to tell users what is the relationship of two related classes. However, in our implementation, we combine classes and relationship properties together. Only within one tree view, users can find out either the classes or the relationships. Also some of the relationships we do not show once in the entire tree view. Due to the necessary of some ontology files, we give

each subclass a sub-hierarchy tree to present its own part, which is much convenient for domain experts to figure out the relationship of each node and help them to select the correct node to generate the GUI for end users.

Jspace[14] is based on ontology technology to browse, search, and discover information in complex, very large data sets. It also has the column display. The significant difference from Jspace and our implementation is that Jspace needs its own configuration file format for input data. It can not work well with a regular OWL file. Therefore for a general user, he/she has to learn how to create a configuration file first, then he/she can browse or search data through Jspace toolkit. However, for our implementation, we have a common input file entry just like protégé, any owl file can work fine just by loaded from a file path or a URL. Users don’t need to worry about some other configurations.

CHAPTER 6 CONCLUSION AND FUTURE WORK

A significant advantage our approach is to utilize the widely supported mechanism for knowledge retrieval and management – ontology. Using our ontology-based GUI generator, a domain expert can browse the data schema through his/her own professional field, choose attribute fields according to needs, and make a highly customized GUI for end users, without knowing any programming techniques. Our approach has established a systematic methodology for domain experts to specify the business needs in an unambiguous manner.

The interface for the domain expert is a tree view structure that shows not only the domain taxonomy categories but also the relationships between classes. By clicking the checkbox associated with each class, the expert indicates his/her choice of the needed information. These choices are stored in a metadata document in XML to store for later use. Since every class and every attribute in the class has been formally specified in the ontology, the various customized GUI can be generated automatically due to the choice of domain experts.

Our approach also can contribute to development of many other applications dealing with complex data in different specialized fields. In this thesis, I just used the METOC data requests as a case study to demonstrate our basic idea of this prototype. Actually, another purpose of this project is to apply our approach to other domains such as bioinformatics and network security, in which complete ontologies have been established already. Applying our approach to development for education and training is in our plan for the near future.

REFERENCE

[1] Navy METOC Introduction

http://hcs-metoc.apl.washington.edu/forecasting_challenges.html

[2] Joint METOC Data Administration Website http://www.cffc.navy.mil/metoc/

[3] Berners-Lee, Tim; James Hendler and Ora Lassila (May 17, 2001). "The Semantic Web".*Scientific American Magazine*. 2008-03-26 Wikipedia

http://www.sciam.com/article.cfm?id=the-semantic-web&print=true.

[4] W3C Semantic Web Activity http://www.w3.org/2001/sw/

[5] OWL Web Ontology Language Overview http://www.w3.org/TR/owl-features/

[6] T. R. Gruber. A Translation Approach to Portable Ontology Specications. Knowl. Acquis., 5(2):199-220,1993.

[7] Protégé website http://protege.stanford.edu/

[8] Michael Spahn, Joachim Kleb, Stephan Grimm, Stefan Scheidl Supporting Business Intelligence by Providing Ontology-Based End-User Information Self-Service 2006

[9] Dieter E.Jenz, President, Jenz & Partner GmbH. Ontology-Based Business Process Management. First Edition Nov, 2007

[10] Yannis Theoharis, Vassilis Christophides, and Grigoris Karvounarakis , Benchmarking Database Representations of RDF/S Stores, 2005

[11] Steve Harris IAM, University of Southampton, UK SPARQL query processing with conventional relational database systems, 2005

[12] Prize Tags : Tree Tag User Guide Jenkov development 2004 www.jenkov.com

[13] Kevin Wilkinson, Craig Sayers, Harumi Kuno, Dave Reynolds .Efficient RDF Storage and Retrieval in Jena2, 2003

[14] JSpace, http://clarkparsia.com/jspace/

APPENDIX

1. Controller.java

```
package display;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Hashtable;

import com.hp.hpl.jena.ontology.OntClass;
import com.hp.hpl.jena.ontology.OntProperty;
import com.jenkov.prizetags.tree.impl.Tree;
import com.jenkov.prizetags.tree.impl.TreeNode;
import com.jenkov.prizetags.tree.itf.ITree;
import com.jenkov.prizetags.tree.itf.ITreeNode;

public class Controller {

    Hashtable<Object, Integer> visitedProperty = new Hashtable<Object, Integer>();
    Hashtable<OntClass, Integer> visitedClass = new Hashtable<OntClass, Integer>();
    private static String SHOW_SUB_CLASSES = "showSubClasses";
    OntAccess onto1;
    ITreeNode display;
    OntoClass currentSelectedComponent;
    HashMap<String,ArrayList<String>> dataPropertyList;
    OntoClass selectedNodeInMainTree;
    boolean isSelected=false;
    ArrayList<String> visitedNodes;
    ITree tree;

    public Controller() {
        dataPropertyList = new HashMap<String,ArrayList<String>>();
        visitedNodes = new ArrayList<String>();
        tree = new Tree();

        // Construct the tree.
        //onto1 = new OntAccess();
        onto1 = new OntAccess("e:/owl/ka.owl",true);
        OntClass c = onto1.createOntClass("Student");
        ITreeNode root = new TreeNode(c
                .getLocalName(),c
                .getLocalName(),"root");
        display = drilldown(c, root);
        tree.setRoot(display);
```

```
        System.out.println("root node-->"+display.toString());
    }


    public ITreeNode drilldown(OntClass c,ITreeNode root) {

        // Display ObjectProperties and RangeClasses
        ArrayList<String> attrs = onto1.getAttributesForCurrentNode(c);
        if(attrs != null && attrs.size()>0){
            dataPropertyList.put(c.getLocalName(), attrs);
            root.setType(root.getType().concat("P"));
        }
        OntProperty[] properties = onto1.getObjProperties(c);
        if (properties.length != 0) {
            for (int i = 0; i < properties.length; i++) {

                //if (root.getId().contains(s)) {
                    //if (properties[i].hasInverse()
                        //     && !visitedProperty.containsKey(properties[i]
                            //             .getInverse())) {
                        visitedProperty.put(properties[i], 1);
                        //ITreeNode child = new TreeNode(properties[i].getLocalName(),
                        //             properties[i].getLocalName(),"property");
                        //child.setType("property");
                        //root.addChild(child);
                        OntClass[] classes = onto1
                                .getRangeClasses(properties[i]);
                        if (classes.length != 0) {
                            for (int j = 0; j < classes.length; j++) {
                                if (classes[j] != null
&& !root.getId().contains(classes[j].getLocalName())) {
                                    ITreeNode childone = new
TreeNode(root.getId()+"/"+classes[j].getLocalName(),
                                                classes[j].getLocalName(),"rangeClass");
                                    //childone.setType("rangeClass");
                                    root.addChild(childone);
                                    drilldown(classes[j],childone);
                                }
                            }
                        }
                    //} //if hasInverse
                /*else if (!properties[i].hasInverse()
                            && !visitedProperty.containsKey(properties[i])) {
                        visitedProperty.put(properties[i], 1);
```

```
                        //ITreeNode child = new TreeNode(properties[i].getLocalName(),
                        //          properties[i].getLocalName(),"property");
                        //child.setType("property");
                        //root.addChild(child);
                        OntClass[] classes = onto1
                                .getRangeClasses(properties[i]);
                        if (classes.length != 0) {
                            for (int j = 0; j < classes.length; j++) {
                                if (classes[j] != null) {
                                    ITreeNode childone = new
TreeNode(root.getId()+"/"+classes[j].getLocalName(),
                                            classes[j].getLocalName(),"rangeClass");
                                    //childone.setType("rangeClass");
                                    root.addChild(childone);
                                    drilldown(classes[j], childone);
                                }
                            }

                        }
                    }*/
                //}if root.get
                //visitedProperty = new Hashtable<Object, Integer>();

            }// for
        }

        // Display Subclasses
        OntClass[] subClasses = onto1.getSubClass(c);
        if (subClasses.length != 0) {
/*          OntoClass subClassdirectory = new OntoClass(
                    "subclass");
            root.add(subClassdirectory);
*/          for (int i = 0; i < subClasses.length; i++) {
                if (subClasses[i] != null) {
                    // display all the sub classes, and solicit user's choice
                    // among these classes, the user's choice will be
                    // choiceClass;
                    ITreeNode child1 = new
TreeNode(root.getId()+"/"+subClasses[i].getLocalName(),
                            subClasses[i].getLocalName(),"subClass");
                    root.addChild(child1);
                    drilldown(subClasses[i], child1);
                }
            }
```

```
            }

            return root;
        }


        /*public void getXmlFromObject(Object obj) throws JAXBException, Exception {
            Class[] classes = new Class[3];
            classes[0] = DataPropertyList.class;
            classes[1] = DataPropertyClass.class;
            classes[2] = DataProperty.class;
            final JAXBContext jc = JAXBContext.newInstance(classes);
            final Marshaller m = jc.createMarshaller();
            m.marshal(obj, new File("sample.xml"));
        }*/

        public static void main(String[] args) {
            // Schedule a job for the event-dispatching thread:
            // creating and showing this application's GUI.

            javax.swing.SwingUtilities.invokeLater(new Runnable() {
                public void run() {
                    Controller cntr = new Controller();
                }
            });
        }

}
```

2. ControllerServlet.java

```
package display;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * Servlet implementation class ControllerServlet
 */
public class ControllerServlet extends HttpServlet {
	private static final long serialVersionUID = 1L;

	/**
	 * Default constructor.
	 */
	public ControllerServlet() {
		super();
		// TODO Auto-generated constructor stub
	}

	protected void processRequest(HttpServletRequest request,
			HttpServletResponse response) throws ServletException, IOException {
		Controller cntr = new Controller();
		request.getSession().setAttribute("ontologyClass", cntr.tree);
		request.getSession().setAttribute("attributesList", cntr.dataPropertyList);
		RequestDispatcher rd = getServletConfig().getServletContext().getRequestDispatcher(
				"/displayowl.jsp");
		if(rd != null){
			rd.forward(request, response);
		}
	}

	/**
	 * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse
	 *		response)
	 */
	protected void doGet(HttpServletRequest request,
			HttpServletResponse response) throws ServletException, IOException {
```

```
        processRequest(request, response);
    }

    /**
     * @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse
     *          response)
     */
    protected void doPost(HttpServletRequest request,
            HttpServletResponse response) throws ServletException, IOException {
        processRequest(request, response);
    }

}
```

3. DataProperty.java

```
package display;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlType;


@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "DataProperty", namespace = "", propOrder = {
    "dataProperty",
    "dataValue"
})

public class DataProperty {
    @XmlElement(name = "dataProperty", namespace="")
    private String dataProperty;
    @XmlElement(name = "dataValue", namespace="")
    private String dataValue;

    public String getDataProperty() {
        return dataProperty;
    }
    public void setDataProperty(String dataProperty) {
        this.dataProperty = dataProperty;
    }
    public String getDataValue() {
        return dataValue;
    }
    public void setDataValue(String dataValue) {
        this.dataValue = dataValue;
    }
    public String toString() { return dataProperty+"="+dataValue;
  }

}
```

4. DataPropertyClass.java

```
package display;

import java.util.ArrayList;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlType;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "DataPropertyClass", namespace="", propOrder = {
        "classUri",
    "dataProperties"
})

public class DataPropertyClass {

    @XmlElement(name = "dataProperties", namespace="")
    private ArrayList<DataProperty> dataProperties;
    @XmlElement(name = "classUri", namespace="")
    private String classUri;

    public String getClassUri() {
        return classUri;
    }
    public void setClassUri(String classUri) {
        this.classUri = classUri;
    }
    public ArrayList<DataProperty> getDataProperties() {
        return dataProperties;
    }
    public void setDataProperties(ArrayList<DataProperty> dataProperties) {
        this.dataProperties = dataProperties;
    }


}
```

5. DataPropertyList.java

```
package display;

import java.util.ArrayList;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;

@XmlRootElement(name = "DataPropertyList", namespace="")
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "DataPropertyList", namespace="", propOrder = {
    "dataPropertyClass",
})

public class DataPropertyList {
    @XmlElement(name = "dataPropertyClass", namespace="")
    private ArrayList<DataPropertyClass> dataPropertyClass;

    DataPropertyList(){
        this.dataPropertyClass = new ArrayList<DataPropertyClass>();
    }

    public ArrayList<DataPropertyClass> getDataPropertyClass() {
        return dataPropertyClass;
    }

    public void setDataPropertyClass(ArrayList<DataPropertyClass> dataPropertyClass) {
        this.dataPropertyClass = dataPropertyClass;
    }


}
```

6. GetFromDB.java

```
package display;

import java.sql.*;
import java.io.*;
import java.util.*;
import java.sql.SQLException;
import com.hp.hpl.jena.db.*;
import com.hp.hpl.jena.rdf.model.*;
import com.hp.hpl.jena.ontology.*;
import com.hp.hpl.jena.db.impl.*;
public class GetFromDB {
	public static IDBConnection connectDB(String DB_URL, String DB_USER,
			String DB_PASSWD, String DB_NAME) {
				return new DBConnection(DB_URL, DB_USER, DB_PASSWD, DB_NAME);
			}

	public static OntModel createDBModelFromFile(IDBConnection con, String name,
			String filePath) {
			  ModelMaker maker = ModelFactory.createModelRDBMaker(con);
			  Model base = maker.createModel(name);
			  OntModel newmodel = ModelFactory.createOntologyModel(
		getModelSpec(maker), base);
			  newmodel.read(filePath);
			  return newmodel;
		}

	public static OntModel getModelFromDB(IDBConnection con, String name) {
		ModelMaker maker = ModelFactory.createModelRDBMaker(con);
		Model base = maker.getModel(name);
		OntModel newmodel = ModelFactory.createOntologyModel(
	getModelSpec(maker), base);
		return newmodel;
	}

	public static OntModelSpec getModelSpec(ModelMaker maker) {
		OntModelSpec spec = new OntModelSpec(OntModelSpec.OWL_MEM);
		spec.setImportModelMaker(maker);
		return spec;
	}

	public OntModel test() {
		/*String DB_URL = "jdbc:oracle:thin:@137.30.122.107:1521:onto";
		String DB_USER = "scott";
```

```
        String DB_PASSWD = "lion";
        String DB = "Oracle";
        String DB_DRIVER = "oracle.jdbc.OracleDriver";*/
        String DB_URL = "jdbc:mysql://137.30.122.107:3306/ontology";
        String DB_USER = "root";
        String DB_PASSWD = "840118";
        String DB = "MySQL";
        String DB_DRIVER = "com.mysql.jdbc.Driver";
        OntModel model=null;
        IDBConnection con=null;
        try {
            Class.forName("com.mysql.jdbc.Driver");


        String filePath = "file:D:/java/WebTree0525/Ontology-Roy-rdf-v0-0303.owl";
        con = GetFromDB.connectDB(DB_URL,DB_USER, DB_PASSWD, DB);
        System.out.println(con);

        //GetFromDB.createDBModelFromFile(con, "Roy",filePath);
        model = GetFromDB.getModelFromDB(con, "Roy");
        //GetFromDB.SimpleReadOntology(model);

        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
       /* finally{
            try {
                con.close();
            } catch (SQLException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }*/
        return model;

}

public static void SimpleReadOntology(OntModel model) {
        for (Iterator i = model.listClasses(); i.hasNext();) {
            OntClass c = (OntClass) i.next();
            System.out.println(c.getLocalName());
        }
}
public static void main(String[] args){
```

```
        GetFromDB g=new GetFromDB();
        g.test();
    }

}
```

7. OntAccess.java

```
package display;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Iterator;
import com.hp.hpl.jena.ontology.*;

//import com.hp.hpl.jena.rdf.model.Model;
//import com.hp.hpl.jena.rdf.model.ModelFactory;
import com.hp.hpl.jena.rdf.model.*;
import java.awt.*;
import java.awt.Container;

import javax.swing.*;
import javax.swing.tree.*;

/**
 * @author fangfang
 *
 */
public class OntAccess {

	/**
	 * @param filepath
	 */
	//public String filePath ;
	OntModel model;


	/**
	 * constructor load ontology
	 */
	public OntAccess(String URI, boolean type) {
		//GetFromDB g=new GetFromDB();
		//model=g.test();
		model = ModelFactory.createOntologyModel();	//create an ontology model
		loadModel(URI,type);
	}

	/**
```

```
 *
 * Load the OWL file
 */
private void loadModel(String uri,boolean type) {

      if(type) {    //true equals reading from local path
          InputStreamReader in;
          try {
                FileInputStream file = new FileInputStream(uri);
                in = new InputStreamReader(file, "UTF-8");
                model.read(in,null);
                in.close();
          } catch (FileNotFoundException e) {
                System.out.println("can not open");
                System.exit(0);
          } catch (IOException e) {
                e.printStackTrace();
                System.exit(0);
          }
      }
      else{
           model.read(uri,null);
      }

}
/**
 * Give all the node in OWL
 */
public String[] getAllnodes(){
      ArrayList<String> rootclass= new ArrayList<String>();
      for(Iterator i=model.listHierarchyRootClasses();i.hasNext();){
           rootclass.add(((OntClass)i.next()).getLocalName());
      }
      String[] d=(String[])rootclass.toArray(new String[0]);
      return d;
}
/**
 *    Initialization
 */
public OntClass createOntClass(String name)    {
      String NS=model.getBaseModel().getNsPrefixURI("");
      OntClass r =model.getOntClass( NS + name) ;
      return r;
}
```

```
/**
 * list all the Subclass names
 */
public String[] getSubClasses(OntClass c)
{
	ArrayList<String> subclass= new ArrayList<String>();
	for(Iterator i= c.listSubClasses(true);i.hasNext();){
		subclass.add(((OntClass) i.next()).getLocalName());

	}
	String[] d = (String[])subclass.toArray(new String[0]);
	return d;
}
/**
 * list all the Subclass Ontclass
 */
public OntClass[] getSubClass(OntClass c)
{
	ArrayList<OntClass> subclass= new ArrayList<OntClass>();
	for(Iterator i= c.listSubClasses(true);i.hasNext();){
		OntClass p=(OntClass) i.next();
		subclass.add(p);
	}
	OntClass[] Ontsubclass = (OntClass[])subclass.toArray(new OntClass[0]);
	return Ontsubclass;
}

/**
 * get every object property of class c
 */
public OntProperty[] getObjProperties(OntClass c){
	ArrayList<OntProperty> objProperty= new ArrayList<OntProperty>();
	for(Iterator j=c.listDeclaredProperties();j.hasNext();){ //list all the Object Properties
		OntProperty p=(OntProperty) j.next();
		if(p.isObjectProperty()){
			objProperty.add(p);
			//System.out.println(objProperty.get(i).toString());
		}
	}
	OntProperty[] OntPro = (OntProperty[])objProperty.toArray(new OntProperty[0]);
	return OntPro;
}
```

```
/**
 * get the range classes of a property
 */
public OntClass[] getRangeClasses(OntProperty p){
	ArrayList<OntClass> rangeClass= new ArrayList<OntClass>();
	for(Iterator j=p.listRange();j.hasNext();){
		OntClass range =(OntClass) j.next();
		rangeClass.add(range);
	}

	OntClass[] Ontrangeclass = (OntClass[])rangeClass.toArray(new OntClass[0]);
	return Ontrangeclass;
}

/**
 * return all the attributes of class c
 */
public ArrayList getAttributes(OntClass c) {
	ArrayList attribute= new ArrayList();
	if(c != null){
	for(Iterator j=c.listDeclaredProperties();j.hasNext();){
		OntProperty p=(OntProperty) j.next();
		if(p.isDatatypeProperty())
			attribute.add(p);
	}
	}
	return attribute;
}
/**
 * return the range of attributes of class c
 */
public ArrayList getAttributesType(OntClass c) {
	ArrayList attribute= new ArrayList();
	if(c != null){
	for(Iterator j=c.listDeclaredProperties();j.hasNext();){
		OntProperty p=(OntProperty) j.next();
		if(p.isDatatypeProperty()){
			for(Iterator i=p.listRange();i.hasNext();){
				OntClass range =(OntClass) i.next();
				attribute.add(range);
			}
		}
	}
	}
```

```
        return attribute;
    }

        public ArrayList<String> getAttributesForCurrentNode(OntClass c) {
        ArrayList<String> attribute= new ArrayList<String>();
        if(c != null){
        for(Iterator j=c.listDeclaredProperties(true);j.hasNext();){
            OntProperty p=(OntProperty) j.next();
            if(p.isDatatypeProperty())      {
                String dataPropertyUri = p.toString();
                String dataProperty = dataPropertyUri
                        .substring(dataPropertyUri.lastIndexOf('#') + 1);
                attribute.add(dataProperty);
            }
        }
        }
        return attribute;
    }



    /**
     *    return all individuals of class c
     */
    public ArrayList getIndividuals(OntClass c){
        ArrayList individual=new ArrayList();
        if(c !=null){
            for(Iterator j=c.listInstances();j.hasNext();){
                OntResource in = (OntResource) j.next();
                individual.add(in);
            }
        }
        return individual;
    }
    /**
     * list all super Classes
     */
    public OntClass[] getsuperclass(OntClass c){
        ArrayList<OntClass> superClass= new ArrayList<OntClass>();
        for(Iterator j=c.listSuperClasses();j.hasNext();){
            OntClass s =(OntClass) j.next();
            if(s.getLocalName()!=null          &&          !s.getLocalName().contains("Resource")
&& !s.getLocalName().contains("Thing"))
                superClass.add(s);
```

```
        }
        superClass.add(c);
        OntClass[] Ontsuper = (OntClass[])superClass.toArray(new OntClass[0]);
        return Ontsuper;
    }

    /*public static void main(String[] args) {
        OntAccess onto1=new OntAccess();

        OntClass c=onto1.getSubClass("WAM");
        OntProperty[] d=onto1.getObjProperties(c);
        OntClass[] ont=new OntClass[5];
        for(int i=0;i<d.length;i++)
        {
            if(d[i]!=null)
            {
                //System.out.println(d[i].getLocalName());
                ont=onto1.getRangeClasses(d[i]);
                for(int j=0;j<ont.length;j++)
                {
                    if(ont[j]!=null)
                        System.out.println(ont[j].getLocalName());
                }
            }
        }

    }*/
    public static void main(String[] args) {
        //OntAccess onto=new OntAccess();
        OntAccess onto = new OntAccess("e:/SAO.owl",true);
        //String[] d=onto.getAllnodes();
        OntClass c=onto.createOntClass("Entity");
        //OntClass[] a=onto.getAttributes(c);
        OntClass[] d=onto.getSubClass(c);
        for(int i=0;i<d.length;i++)
        {
            if(d[i]!=null)
            System.out.println(d[i].getLocalName());
        }


    }

}
```

VITA

Fangfang Liu was born in Jinzhou, Liaoning Province China. She earned a Bachelor Degree in Computer Science as major and English as minor during her study at Dalian University of Technology, China from 2002 to 2007. She was enrolled in the graduate program in Computer Science Department at the University of New Orleans in September 2007.