NISTIR 7509

An Ontological Modeling Platform

Vei-Chung Liang
Conrad Bock
XuanFang Zha

National Institute of Standards and Technology
U.S. Department of Commerce

NISTIR 7509

An Ontological Modeling Platform

Vei-Chung Liang
Conrad Bock
XuanFang Zha
Manufacturing Systems Integration Division
Manufacturing Engineering Laboratory

June 2008

U.S. Department of Commerce
Carlos M. Gutierrez, Secretary

National Institute of Standards and Technology
James M. Turner, Deputy Director

An Ontological Modeling Platform

Vei-Chung Liang
Conrad Bock
XuanFang Zha

June 11, 2008

Abstract

The Ontological Modeling Platform described in this paper is a class library for extended ontological operations, to support extension of ontology languages, composition of interconnected elements, and high-level product modeling services, such as specialization of product models. In this report, we explain the structure, implementation, and design decisions of the platform. The platform has an Application Programming Interface (API) architecture that enables it to be implemented on multiple off-the-shelf Ontology Web Language (OWL) APIs. It provides model spaces to distinguish and integrate modeling languages, models, and instances of models. Each model space can only instantiate classes defined from the immediate upper level model space. A tutorial is given to demonstrate how to construct a simple plate-fixing assembly by using the platform.

1 Introduction

An integrated and collaborative product design environment should enable engineers to interact and reach agreement by sharing and exchanging product information and knowledge. The information encoded in product models must be unambiguously understood by all participating parties (for example, engineers), independent of their perspectives, physical locations, and times of contribution.

One approach to providing such product knowledge representations is based on ontologies. Ontologies emerged in applied artificial intelligence as a means for sharing knowledge. An ontology can be informally defined as a description of concepts and relationships that are used in a specific knowledge domain. Latest research in product modeling indicates such a trend towards semantic product data and knowledge modeling for better interoperability using the product ontologies, see related work in [Bock 2007b], especially the composite structure ontology for interconnected elements capturing relationships among parts within assemblies [Bock 2007a]. The semantic web has the potential for richer representations of product concepts and their relations on the web, and thus should provide us with more intelligent and collaborative product services. Today, we have quite impressive results on semantic modeling, manifested by standards that have been adopted, in particular the Resource Description Framework / Schema (RDF/S) and the Ontology Web Language (OWL) [W3C 2007], development frameworks (e.g., Jena, Protégé, TopBraid Composer) [SourceForge 2007a, 2007b; SMI 2007, TopQuadrant 2007], best practice and deployment recommendations [OMG 2007], and many applications [Bock 2007a, 2007b].

In developing product ontologies, it is important to extend ontology languages when their capabilities are needed as part of the product modeling language. This enables reasoners to assist in product modeling problems, such as checking consistency of requirements and designs [Bock 2007b]. For example, it is useful to specialize product models from existing ones. Specialization is already supported by ontology languages and their reasoners. To take advantage of this, product models should be introduced by extending the ontology language, rather than just using the ontology language to define an entirely separate product modeling framework.

Extending an ontology language requires multiple layers of instantiation. Metamodeling techniques and model-driven architectures (MDA) [OMG 2007] use at least three layers (M2, M1, and M0):

- The M2 layer manages vocabularies and provides corresponding rules to instruct how instantiation should be performed.
- The M1 layer manages specific usages of vocabularies, for example, engineering designs.
- The M0 layer manages intended or physical things and behaviors.

The elements in each layer are instantiated from the layer above it (M1 from M2, M0 from M1), and the top two layers can support specialization within them. For example, an element for product models in general is specialized from the ontology language at M2. A particular product model, for example, a design for car, is instantiated from this at M1, where it can be specialized from other product models describing vehicles in general. An intended or actual car is instantiated at M0.

The platform described in this paper is a class library for multi-layer ontological operations, as well as composition of interconnected elements [Bock 2007a], and high-level product modeling services [Bock 2007b]. The platform uses multiple off-the-shelf and customized APIs to support its operations. A composition conceptualization can be constructed from scratch through the existing composition base models and Java programming. Besides using the platform to create composition conceptualizations, the platform can be extended with new base models. The conceptualization in the platform captures an entire application, from modeling language to individual things. A conceptualization may contain several layers of model spaces. An MDA configuration for a conceptualization contains three model spaces (M2, M1, and M0 model spaces). The roles that the model spaces can play are determined by the specific conceptualization.

The models and services supported by the platform require specialized capabilities that are not available in some modeling implementations:

1) Polymorphism between properties and classes. An instance of a property can be coerced to become an instance of a class. At the time of the implementation, only the Jena API has full support for this feature [SourceForge 2007a]. Protégé-OWL 3.2 does not support such feature [SMI 2007].

2) Reification of the RDF/S statements. RDF subject, object, and predicate triples about an OWL statement can be retrieved from a given OWL statement. At the time of the

implementation, only the Jena API has full support for this feature. The OWL API does not support such a feature [SourceForge 2007b].

3) Organization of multiple model spaces. Instantiation policies can be defined for a stack of model spaces. None of the APIs mentioned above provide the operations for organizing multiple model space policies.

The ontological modeling platform described in this paper supports all the above features.

The rest of the paper is as follows. Section 2 introduces the ontological modeling platform including its logical structure, API structure, and implementation architecture. Section 3 provides a quick tutorial for constructing a composition conceptualization from scratch through the existing composition base models and Java programming. Section 4 describes a scenario to extend the platform with new base models and an extension example to include new base models. Section 5 proposes a three model space conceptualization framework and describes its implementation details. Section 6 gives a summary and concluding remarks for the paper.

2 Ontological Modeling Platform

The platform is a class library for the extended ontological operations described in the previous section. This section introduces the ontological modeling platform including its logical structure, API structure, and implementation architecture.

2.1 Logical Structure

The logical architecture of the platform is shown in Figure 1, consisting of conceptualizations, model spaces, and base models. Conceptualizations are frameworks for defining model spaces. They are used in the platform to define modeling languages, models, and instances of models. The model spaces are numbered following the Object Management Group's (OMG) Model-Driven Architecture (MDA) [OMG 2007] conventions. An element in M1 model space is defined as an instance of classes in the M2 model space. The M1 elements are further treated as classes for M0 elements. Thus, an M0 element is an instance of an M1 element.

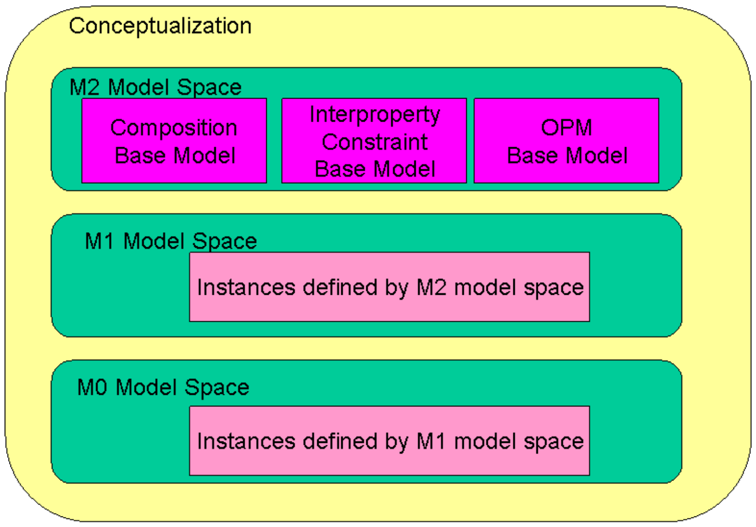
Conceptualization
M2 Model Space
Composition Base Model
Interproperty Constraint Base Model
OPM Base Model
M1 Model Space
Instances defined by M2 model space
M0 Model Space
Instances defined by M1 model space

Figure 1: Architecture of Composition API

The mapping from metalayers to engineering design and manufacturing can be clearly identified. The M2 model space defines the basic constructs (ontology) for describing engineering design concepts in M1 model space. For example, the M1 model space uses M2 constructs to define engineering designs such as a car assembly or configuration. An M0 model space contains product instances (for example, a specific car with a specific engine number, or an intended car as imagined by the designer) that are governed by the design elements in the M1 model space.

Rules and policies applied on an element in the current model space are defined from its upper model space. Rules governing M1 elements are defined in the M2 model space and rules governing M0 elements are defined in the M1 model space.

Each model space may have different behaviors, because of each model space instance plays a different role inside a conceptualization. However, under the current implementation, these model spaces behave in the same way because no extension has been made to each of the model spaces (they are defined by the same model space class definition). Users can define different behaviors for different model spaces if desired. For example, a conceptualization with three model space is shown in Figure 1, only the M2 model space instance contains base models, such as Composition base model [Bock 2007a], Interproperty Constraint base model, and Ontological Product Model (OPM) base model [Bock 2007b], and the M0 model space cannot define its own classes. Another example is that all elements in M0 model space must be instances of elements in M1 model space. The current implementation does not distinguish the behavioral difference in the model space definition. It is the programmer's responsibility to define the behaviors of the model space instances in the specific conceptualization.

2.2 API Structure

The relationship among APIs is depicted in Figure 2. The Jena API [SourceForge 2007a] is at the bottom and provides basic semantic operations to manipulate OWL/RDF models and documents [W3C 2007]. These semantic models can be either in memory or in a persistent storage.

The base model API is between the model space API layer and The Jena API. It provides extended ontological operations and policies for programmers to model interconnected elements [Bock 2007a], such as composition relationships, or other extensions with rules defined in the policies. For example, an instantiation of a connector in the composition API may require several checks before the created instance can be asserted in the model (for example. to create or load the ontology model, to retrieve class definitions, to instantiate individuals accordingly). This API also uses an adapter design pattern [Gamma 1995] for the future extension so that other OWL APIs besides Jena can be used.

The model space API manages the model spaces that constitute a conceptualization. For example, a typical three-model-space conceptualization may have composition operations to instantiate model spaces, and to define and instantiate elements in the spaces. In this API, method calls activate modeling policies registered for elements in the model space to identify parts and connections in model spaces, and check for proper contextualization.

The conceptualization API is used to manage model spaces, for example, to create them. A conceptualization can have many model spaces, but in the examples of this paper, only three are needed, one for each metalayer described in Sections 1 and 2.1.

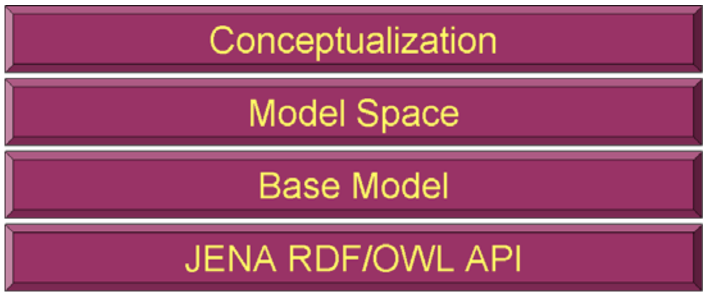

Figure 2: API Relationships

2.3 Implementation Organization

The platform uses multiple packages to organize the architecture (see Figure 3). Currently, these packages are implemented in Java and are only for the Jena API. Porting the implementation to other semantic web APIs, such as the OWL API, is a topic for future work. The packages include:

gov.nist.comp.builder: This package provides several base model builder interfaces for constructing and saving the base models with various construction options (for example, construct models by importing or by direct construction).

gov.nist.comp.builder.impl.jena: The package provides Jena implementation of the builder interfaces.

gov.nist.comp.model: This package provides interfaces that define various base models as Java types (for example, Composition base model, inter-property constraint, and OPM base model). These java types define strings that represent the Uniform Resource Identifiers (URIs) for the elements used in base models.

gov.nist.comp.model.jena: This package provides Java classes for the base model Jena implementation. These base model classes also contain prefix tables and policy tables.

The prefix table stores the prefixes that are used in the base model schema. The policy table stores policies that can instruct model space to properly construct, modify, and destroy classes and properties.

gov.nist.comp.model.app: This package provides a conceptualization abstract class and several concrete conceptualization classes that demonstrate how to describe application-specific conceptualization. In this document, we use a plate-fixing assembly as an example. See Section 3 below.

gov.nist.comp.modelspace.jena: The package provides a model space class defining the behavior on a model space.

gov.nist.comp.policy: This package provides policy interfaces.

gov.nist.comp.policy.jena: This package provides the Jena implementation of the policy interfaces defined in the gov.nist.comp.policy package. Only few elements from base models need predefined policies. Other elements can just use the ordinary OWL/Jena operations.

gov.nist.comp.vocabulary: This package provides vocabulary constants generated from Jena's schemagen. The generated constants allow the programmers to have quick access to the elements defined in the base models.

gov.nist.comp.Main: This package provides a Preferences class that allows the platform to store bootstrap information.

These packages are designed to support the following engineering information modeling and meta-modeling tasks:

Instantiate, import, or construct conceptualizations.

Within the conceptualization, specify and construct model spaces and meta-model spaces.

Within each model space, define, create and instantiate classes and individuals to describe a model using composition extension or inter-property constraints.

Define composition extension policies and other constraints that are used to govern elements in base models.

Serialize and de-serialize OWL models.

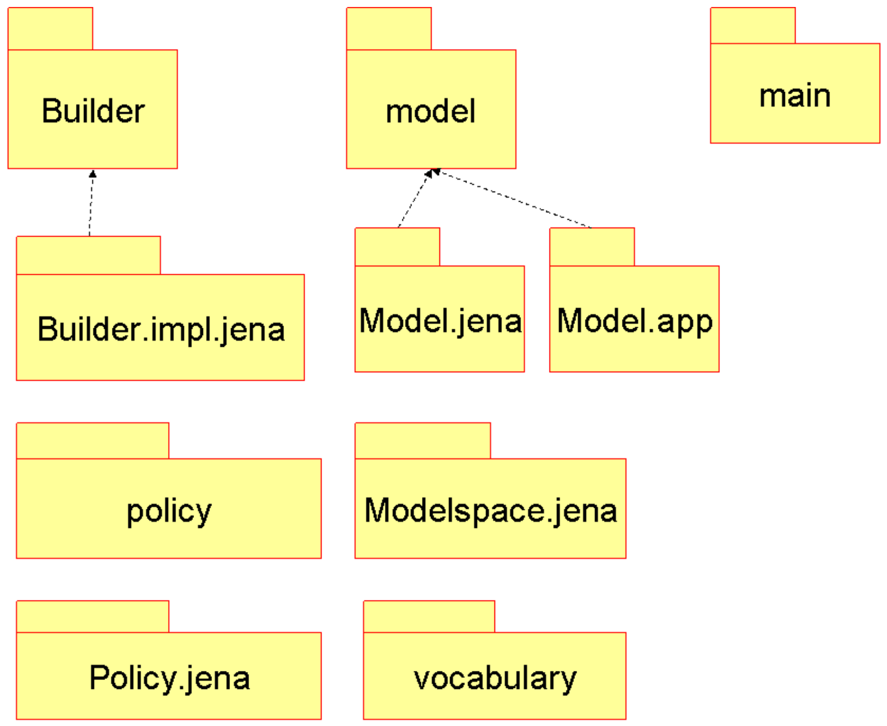

Figure 3: Ontological Modeling Platform Packages

3 A Tutorial for Constructing Conceptualizations

This quick tutorial depicts how one can use the existing composition base models to construct a conceptualization from scratch through Java programming. The example we used is the plate-fixing assembly, in which two plates are held in a fixed relative position with a fastener.

Source code for a complete example can be found in Plate2Conceptualization.java, in the gov.nist.comp.model.app package. It is highly recommended to go over the tutorial with the complete example in order to understand the constructing mechanism. A more detailed explanation of possible extensions will be described later in this document.

To run the tutorial from the package is straightforward:

1. Unzip the package to a folder that is different from your eclipse workspace.
2. Start eclipse.
3. Right click on the package explorer panel and select “import…” from the context menu.
4. Choose “Existing Projects into Workspace” as your import source type and click next.
5. Find the folder containing the unzipped package as your root directory and select the project Interproperty2.
6. Click finish to start import.

You can also use the archive file directly to perform the import. Once you import the project, you can add all the jar files from Jena’s lib subdirectory or you can create a handle for these jar files. Please refer to the eclipse manual [Eclipse 2007].

3.1 Planning a Conceptualization

The example goes through the following common practice in engineering information modeling:

1. **Decide a namespace URI and a prefix for the conceptualization.** In this tutorial, the namespace URI for the plate assembly is file://comp/plate#. Note that the hash sign is used because it is a common practice in the Semantic Web community to define a namespace. The prefix for this conceptualization is "plate." The URI prefix is crucial information because Jena XML serialization needs to use the prefix information to convert a common URI to a legal XML name[1].
2. **Decide what kind of conceptualization is going to be used**. Conceptualization can be configured with many model spaces. A typical configuration is a three model space conceptualization. The tutorial conceptualization is a three model space conceptualization. That is, it contains M2, M1, and M0 model spaces.
3. **Decide what base models are going to be included in the M2 model space.** In this tutorial, only composition base model is going to be included.
4. **Decide what elements are going to be constructed in the M1 model space.** In this tutorial, a plate class, a plate assembly class, two part-properties for two plate roles (hasPlate1, hasPlate2), a medium level property-that-connects (FIXES), and a plate connector need to be defined.
5. **Decide what elements are going to be constructed in the M0 model space.** In this tutorial, two plate individuals, a plate assembly individual, and their corresponding properties need to be instantiated.

3.2 Creating Conceptualization According to the Plan

A simple conceptualization can now be constructed with the following actions according to the plan.

1. **(Carrying out Decision 1)** The prefix and namespace URI will be used in the action item 6 when we instantiate our plate assembly conceptualization instance.
2. **(Carrying out Decision 2)** The conceptualization for a particular design is realized through specialization. That is we define a java class that specializes a pre-defined conceptualization template. Inside the gov.nist.comp.model.app package, there is a ThreeModelSpaceConceptualization that can be used for this tutorial. Thus, we define a new Java class called Plate2Conceptualization for the plate assembly.
3. **(Carrying out Decision 3)** Since the ThreeModelSpaceConceptualization is an abstract class, three abstract methods have to be implemented. The constructM2Space is implemented to describe what base models are going to be included.

[1] The OWL community recommends using hash sign "#" at the end of a namespace string and thus file://comp/complm#Connector becomes an identifier. However, such an identifier is not legal in XML. The JENA API replaces such a namespace with an auto-generated prefix (for example. j.0 or j.1) to avoid the XML serialization error. To avoid the hard-to-understood auto-generated prefix, a prefix is required in the current implementation.

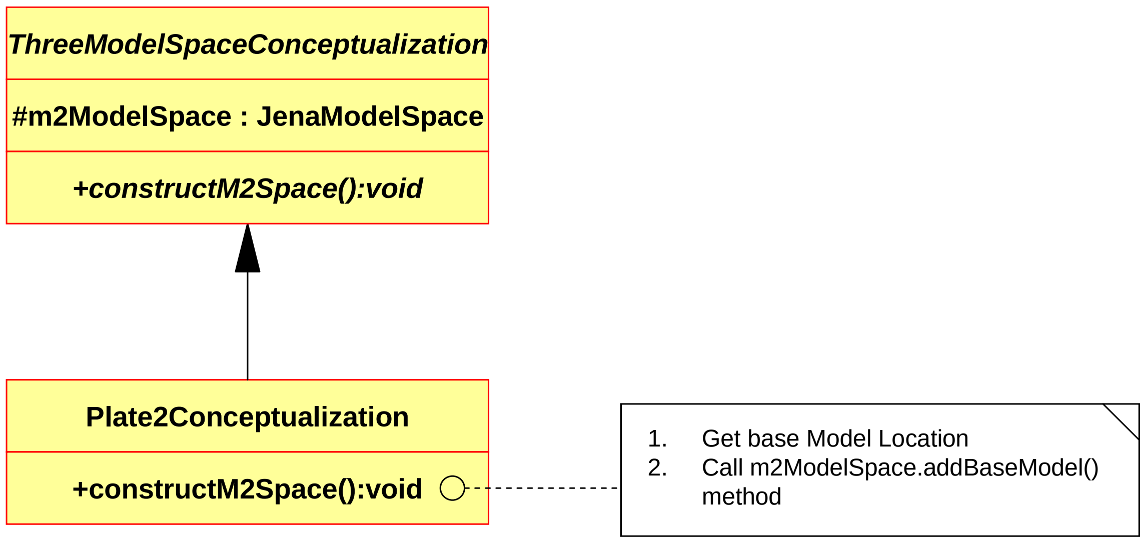
ThreeModelSpaceConceptualization
#m2ModelSpace : JenaModelSpace
+constructM2Space():void
Plate2Conceptualization
+constructM2Space():void
1. Get base Model Location
2. Call m2ModelSpace.addBaseModel() method

Figure 4: Constructing M2 Model Space

Figure 4 demonstrates how base model can be added by calling the addBaseModel method. Note that the m2ModelSpace attribute is inherited from the ThreeModelSpaceConceptualization class that refers to the M2 model space in this conceptualization. Three arguments for the addBaseModel method are a base model class, the base model importing URI, and base model schema location. A composition base model class contains the policies that composition model carries. The policies can thus be registered to the model space through its carrying class. The base model importing URI defines the URL that is going to be used for the base modeling importing. The schema location specifies the physical location of the base model OWL file.

4. **(Carrying out Decision 4)** The inherited constructM1Space method is implemented to describe what elements are going to be used in the M1 model space (See Figure 5). The m1ModelSpace inherited data member refers to the M1 model space in this conceptualization.

   A Plate class is created by calling the createCurrentLevelClass method. In Jena, all resources should be created by using their full names. Thus, we have to use the baseURI of m1ModelSpace and use the Java String concatenation operation '+' or StringBuffer to construct full names. Note that the baseURI is a public field of JenaModelSpace, which is different and independent from the baseURI of the conceptualization. A conceptualization will have its own baseURI. The createCurrentLevelClass method call is used to create not only the part OWL class, Plate, but also the assembly OWL class, Plate-Fixing.

```
protected void constructM1Space(){
            //create the plate class
OntClass plate = m1ModelSpace.createCurrentLevelClass(m1ModelSpace.baseURI +
"Plate");

OntClass plateFixing =
m1ModelSpace.createCurrentLevelClass(m1ModelSpace.baseURI + "Plate-Fixing");

//because we know that hasPlate1 is also a property. we can make it as a
property
OntProperty hasPlate1 = m1ModelSpace.instantiateFromUpperLevelClass(
      PartPart.PART_PROPERTY.getURI(),
      m1ModelSpace.baseURI + “hasPlate1").asProperty();
            hasPlate1.addDomain(plateFixing);
            hasPlate1.addRange(plate);

OntProperty hasPlate2 = m1ModelSpace.instantiateFromUpperLevelClass(
      PartPart.PART_PROPERTY.getURI(),
      m1ModelSpace.baseURI + "hasPlate2").asProperty();
            hasPlate2.addDomain(plateFixing);
            hasPlate2.addRange(plate);

Individual fastenerC = m1ModelSpace.instantiateFromUpperLevelClass(
      PartPart.LM_CONNECTOR.getURI(),
      m1ModelSpace.baseURI + "FastenersConnector");

m1ModelSpace.addProperty(
      plateFixing, PartPart.HAS_CONNECTOR, fastenerC);

OntProperty fixs = m1ModelSpace.instantiateFromUpperLevelClass(
      m2ModelSpace.getUpperLevelClass(
            OPM.BEHAVIOR_PROPERTY.getURI()).getURI(),
            m1ModelSpace.baseURI + "Fixes").asProperty();
            fixs.addDomain(plate);
            fixs.addRange(plate);

m1ModelSpace.addProperty(
      fastenerC, PartPart.PROPERTY_THAT_CONNECTS, fixs);

List<OntProperty> domainPath = new ArrayList<OntProperty>();
domainPath.add(hasPlate1);

Iterator<OntProperty> ir = domainPath.iterator();
RDFList rList = m1ModelSpace.getOntModel().createList();
while(ir.hasNext()){
      rList = rList.cons(ir.next());
}

m1ModelSpace.addProperty(
      fastenerC, PartPart.DOMAIN_END_OF_CONNECTOR, rList);

List<OntProperty> rangePath = new ArrayList<OntProperty>();
rangePath.add(hasPlate2);
ir = rangePath.iterator();

RDFList dList = m1ModelSpace.getOntModel().createList();
      while(ir.hasNext()) dList = dList.cons(ir.next());

m1ModelSpace.addProperty(
      fastenerC, PartPart.RANGE_END_OF_CONNECTOR,dList);
}
```

Figure 5: Constructing M1 Model Space

Two part-properties, hasPlate1 and hasPlate2, are created by calling the instantiateFromUpperLevelClass method. The difference between the createCurrentLevelClass and instantiateFromUpperLevelClass is that the former uses OWL class to be the upper level class and the latter requires explicit specification of the name of the upper level class. The createCurrentLevelClass is equivalent to instantiateFromUpperLevelClass with OWL.Class.getURI() as the first argument. The instantiateFromUpperLevelClass takes two arguments: one is the upper level class full name and the other is the individual full name. The full name of the upper level class can be accessed by using the Composition class. Composition class is created by Jena's schemagen tool. Every element that is defined in the OWL schema file fed into the schemagen will become either a static instance of OntClass or OntProperty defined in the output class file. Thus, the full name of the part-property can be retrieved by calling Composition.PART_PROPERTY.getURI().

Note that the method will return an Individual of type PART_PROPERTY but hasPlate1 and hasPlate2 should be manipulated as an OntProperty. Thus, we call asProperty() to convert Individual instance to OntProperty instance.[2]

With all the classes and properties defined, we can now create a connector to regulate the connection. A FastenersConnector can be created as a medium level connector by using the instantiateFromUpperLevelClass method. Once the connector is created, we can use the addProperty method to associate the connector with the assembly. The addProperty takes three arguments: the domain, the property, and the range. In order to make the plate assembly to have a FastenrsConnector, the property in the argument list will be Composition.HAS_CONNECTOR.

Three values of a connector have to be assigned for FastenersConnector. They are:

a PROEPRTY_THAT_CONNECT,

a DOMAIN_END_OF_CONNECTOR, and

a RANGE_END_OF_CONNECTOR.

The PROPERTY_THAT_CONNECT in this example is a medium level [Bock 2007a] PROPERTY_CLASS instance, Fixs. Because of the policy mechanism, this Fixs will also be defined as a subtype of Statement automatically and the programmer does not have to worry about it. This Fixs can be treated as a property by using the asProperty() method. Since it is an OntProperty instance, setDomain and setRange can be used to set the domain and range types. With Fixs defined, the addProperty method can be used to add it as the value for the PROPERTY_THAT_CONNECT for the FastnersConnector.

The values of the DOMAIN_END_OF_CONNECTOR and RANGE_END_OF_CONNECTOR properties have to be assigned by programmer in the current implementation. Since their values have to be of type RDFList, instances of RDFList have to be constructed first. One uses the m1ModelSpace's getOntModel() to get an OntModel reference and use it to call createList() to create an RDFList instance.

A Java ArrayList can be first constructed to establish the proper order for each path. An iterator of the list can be used to assign elements inside the ArrayList to the RDFList. This is accomplished by using the RDFList's cons method. Finally, addProperty method is used again to associate these RDFList instances to the FastenersConnector.

[2] `Individual` and `OntProperty` are JENA classes that are used to represent OWL individual and property respectively. The `asProperty()` is the JENA OWL type polymorphic mechanism which can convert an individual to a property. Other polymorphic mechanism includes `asClass()`, `asIndividual`(), etc. Please see another note for detail.

5. **(Carrying out Decision 5)** The inherited constructM0Space is implemented to describe what elements are going to be in the M0 model space. In this implementation, m0ModelSpace is used to refer to the M0 model space (see Figure 6).

   Three individuals, two plates and one assembly, can be created by calling the instantiateFromUpperLevelClass of the m0ModelSpace instance. Note that full name string, m1ModelSpace.baseURI + "Plate", must be used to retrieve the class reference in the upper model space; and full name string, m0ModelSpace.baseURI + "Plate1Indiv", must be used for the created individual in the current model space.

   By calling the addProperty, two plate individuals can be defined to belong to the assembly individual. Using the getUpperLevelProperty, one can retrieve a reference for the Fixs property defined in the M1 model space. Using this reference, two plate individuals can be connected together. Note that a reified statement will also be created automatically due to the policy specified for the PROPERTY_CLASS.

```
protected void constructM0Space(){

Individual plateIndiv1 =
m0ModelSpace.instantiateFromUpperLevelClass(m1ModelSpace.baseURI + "Plate",
m0ModelSpace.baseURI+"Plate1Indiv");

Individual plateIndiv2 =
m0ModelSpace.instantiateFromUpperLevelClass(m1ModelSpace.baseURI + "Plate",
m0ModelSpace.baseURI+"Plate2Indiv");

Individual plateFixing1 =
m0ModelSpace.instantiateFromUpperLevelClass(m1ModelSpace.baseURI + "Plate-
Fixing", m0ModelSpace.baseURI+"Rivet-Plate-FixingIndiv1");

//also get the fixes property reference point.
OntProperty fixesProperty =
m0ModelSpace.getUpperLevelProperty(m1ModelSpace.baseURI + "Fixes");


m0ModelSpace.addProperty(plateFixing1,
m0ModelSpace.getUpperLevelProperty(m1ModelSpace.baseURI+"hasPlate1"),
plateIndiv1);

m0ModelSpace.addProperty(plateFixing1,
m0ModelSpace.getUpperLevelProperty(m1ModelSpace.baseURI+"hasPlate2"),
plateIndiv2);

m0ModelSpace.addProperty(plateIndiv1, fixesProperty , plateIndiv2);
}
```

Figure 6: Constructing M0 Model Space

6. **(Carrying out Decision 1)** To create conceptualization, one uses the conceptualization class constructor to instantiate an instance (See Figure 7).

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

Plate2Conceptualization pcon =
new Plate2Conceptualization("file://comp/plate#", "plate");

      //pcon.printConceptualization(false);
pcon.saveConceptualization("PlateModelSpace", true);
pcon.saveConceptualizationInOneFile(null);
}
```

Figure 7: Instantiate Conceptualization

The base namespace URI and namespace prefix are passed to the constructor. The constructor will generate the base URI for each model space by inserting "M2", "M1", and "M0" strings before the hash sign of the base URI. The prefixes for the model spaces are generated in the same fashion.

In this code snippet, an instance of Plate2Conceptualization is created. The Plate2Conceptulaization is a subclass of ThreeModelSpaceConceptualization. A ThreeModelSpaceConceptualization is a special type of Conceptualization containing three model spaces.

The printConceptualization method can be called to print out the OWL model on the screen or saveConecptualization method can be called to save all three model spaces to owl files. The "PlateModelSpace" is the base string that will be passed to the saveConecptualization method to generate "PlateModelSpaceM2.owl", "PlateModelSpaceM1.owl", and "PlateModelSpaceM0.owl" files representing the corresponding model space. The output will be three OWL documents saved in the root execution directory.

This simple tutorial explains how one can use the platform to construct a semantic engineering information model. The next section explains the design decisions made in more detail.

4 An Extension Example of Including New Base Model

Besides using the Ontological Modeling Platform to create composition conceptualizations, another scenario is to extend the platform with new base models. In this example, steps of how to add a new OPM base model are explained. Usually, a meta-model OWL file is ready to be integrated with the rest of the API. If the OWL file is not ready, programmers must create a builder class for constructing the schema. In this example, both approaches are explained. Similar to the previous tutorial, a plan of actions have to be defined first.

4.1 Planning for Adding a New Base Model

The plan for adding an OPM base model to the platform is the following:

1. **Check that the OPM OWL file is ready to be used.** Sometimes, the OWL file is not validated or checked. A valid OWL file must be prepared for the rest of the tasks. One

way to check or validate the OWL file is to dump the file to an OWL valuator. An online service provided by WonderWeb can be found on the Internet[3].

Another way to validate OWL files is to use the Jena API's tool to reprint the input OWL file. This will activate the checker. One can also use OWL editors to perform checking. Note that not all OWL editors are supporting the property-class concept [Bock 2007a]. Currently, TopBraid composer supports such a capability [TopQuadrant 2007].

2. **Create an OPM base model interface** that will hold all the URIs defined in the OWL file. These URI strings become handy for the rest of the tasks.
3. **Create a Jena OPM base model class** that implements the OPM base model interface. The implementation will allow the base model class to access the URI strings. This base model class will also hold the prefix table and policy table.
4. **Extend JenaPolicy class if necessary.** The created policy class will be stored as in the previously created Jena OPM base model class' static block.
5. **Create a base model builder[4]**. A base model builder is a Java class that can be used to create a base model from scratch. The idea is to manually script the construction process in the builder's constructSchema method.
6. **Update the Preference class** so that the program will be able to learn where the OPM base model owl file is located. Currently, it is stored at the bin/rdf directory.
7. **Use Jena's schemagen to create a vocabulary file for OPM base model.** This vocabulary file will also be handy for later tasks.

4.2 Creating OPM Base Model Extension

1. **(Carrying out item 2)** An OPM base model interface is created in the gov.nist.comp.model package. Since this interface can also be implemented for Protégé 4.0, we use Java's generic type [Bracha 2004] [Java 2007] to make it flexible enough for future extension. Figure 8 shows a snippet of the OPM base model definition. All strings are defined in the static nested interface NAMES[5]. All the other base model interfaces follow this style. Although this style somewhat cumbersome, it distinguishes programming constants and OWL-related constants.

   Three important string constants are also needed to be defined. They are IMPORT_URI, NAMESPACE, and PREFIX. All the base model interfaces require these three constants. The IMPORT_URI is used when one wants to import this base model schema to a model. Note that IMPORT_URI is defined as the same as the base URI only without a hash sign. The NAMESPACE is used to record the base URI of this base model. Once it is defined, it can be used to define other elements of this base model. The PREFIX defines the xmlns prefix of the base model. It will only be used when one wants to register the prefix to the model.

[3] http://phoebus.cs.man.ac.uk:9999/OWL/Validator

[4] Note that the base model builder is not used anywhere else in the current implementation. One can consider this step as an exercise.

[5] http://java.sun.com/docs/books/tutorial/java/javaOO/nested.html

```
public interface OPMBaseModel<T> extends BaseModel<T> {

public static interface NAMES {
public static String IMPORT_URI = "file://comp/compopm";

public static String NAMESPACE = "file://comp/compopm#";
public static String PREFIX = "compopm";

public static String BEHAVIOR_C = NAMESPACE + "Behavior";

public static String BEHAVIOR_PROPERTY_PC = NAMESPACE +
      “BehaviorProperty";

public static String REQUIREMENT_C = NAMESPACE + "Requirement";

public static String REQUIRED_P = NAMESPACE + "required";

public static String REQUIRED_BY_P = NAMESPACE  + "requiredBy";

public static String ON_ARTIFACT_P = NAMESPACE + "onArtifact";

public static String REQUIRED_BEHAVIOR_P = NAMESPACE +
      "requiredBehavior";

public static String REQUIRED_FORM_P = NAMESPACE + "requiredForm";

public static String FUNCTIONAL_REQUIREMENT_PROPERTY_C = NAMESPACE +
      "FunctionalRequirementProperty";

public static String REQUIREMENT_ON_ARTIFACT_PROPERTY_C = NAMESPACE +
      "RequirementOnArtifactProperty";

public static String REQUIREMENT_WITH_ENVIRONMENT_PROPERTY_C =
      NAMESPACE + "RequirementWithEnvironmentProperty";

public static String FORM_C = NAMESPACE + "Form";

public static String GEOMETRY_C = NAMESPACE + "Geometry";

public static String MATERIAL_C = NAMESPACE + "Material";

public static String ASSEMBLY_PROPERTY_P = NAMESPACE +
      "AssemblyProperty";

public static String RELATION_END_C = NAMESPACE + "RelationEnd";

public static String ARTIFACT_C = NAMESPACE + "Artifact";

public static String FUNCTION_C = NAMESPACE + "Function";
}}
```

Figure 8: Fragment of OPM Base Model

A naming convention for string identifiers is used:

_C: OWL class
_P: OWL property
_PC: property-class

For example, if the string is referring to an OWL class, a '_C' will be attached to the end of the identifier. For example, BEHAVIOR_C refers to a Behavior OWL class.

2. **(Carrying out item 3)** A Jena OPM base model class is defined in the gov.nist.comp.model.jena package. It will extend JenaBaseModel class and implement the OPMBaseModel<OntModel>. Note that an OntModel type is supplied as a type argument to the OPMBaseModel. In the first static block in Figure 9, we added used prefixes to the table. Note that the MF prefix is used for referring to the property-class used in a medium and full level of the composition base model [Bock 2007a]. Since no policy is defined for this base model yet, the second static block remains blank.
3. **(Carrying out item 5)** Before create a Jena OPM base model builder, an OPM base model builder interface is defined in the gov.nist.comp.builder package. This interface is an extension of BaseModelBuilder interface. It is overriding the return type of getBaseModel method to an OPMBaseModel.

   A Jena OPM base model builder implementing OPMBaseModelBuilder is defined in the gov.nist.comp.builder.impl.jena package. There are two methods that need to be implemented inside the JenaOPMBaseModelBuilder. One is importSchema and the other is constructSchema. The importSchema will create a model with an imports statement. The constructSchema will construct the model from scratch. Please refer to the source code for more details.

```
public class JenaOPMBaseModel extends JenaBaseModel implements
            OPMBaseModel <OntModel>{

      public static String POLICY_FILE = "JenaOPMBaseModel.rules";

      public JenaOPMBaseModel(OntModel m){
            this.ontModel = m ;
      }

public static HashMap<String, String> prefixTable =
      new HashMap<String, String>();

public static HashMap<String, JenaPolicy> vocabularyPolicies =
      new HashMap <String, JenaPolicy>();

      static{
            prefixTable.put(PartPartBaseModel.NAMES.MF_NAMESPACE_PREFIX,
                      PartPartBaseModel.NAMES.MF_NAMESPACE);
            prefixTable.put(OPMBaseModel.NAMES.PREFIX,
                      OPMBaseModel.NAMES.NAMESPACE);
      }

      static {
      }
}
```

Figure 9: Jena OPM Base Model

4. **(Carrying out item 6)** New properties have to be defined in the Preference class pointing to the location of the compopm.owl file. This can be done by creating new property handles, OPM_SCHEMA_FILE and OPM_SCHEMA_FILE_VALUE, in the Preference class (see Figure 10).

```
public static final String OPM_SCHEMA_FILE = "opm.schema";
public static final String SCHEMA_FOLDER = "schema.folder";

//default values for the schema file location keys
private static final String INTERPROPERTY_SCHEMA_FILE_VALUE =
"InterpropertyConstraint.owl";

private static final String INHERIT_PARTPART_SCHEMA_FILE_VALUE =
"inherit_partpart.owl";

private static final String OPM_SCHEMA_FILE_VALUE = "compopm.owl";
```

Figure 10:Property Definitions in Preference Class

The value can be assigned to the handle by the following line in the Preference's initialize method:

```
preferences.setProperty(OPM_SCHEMA_FILE, OPM_SCHEMA_FILE_VALUE);
```

Figure 11: Using setProperty in Preference initialize Method

5. **(Carrying out item 7)** The compopm.owl file can be passed to the Jena's schemagen by using the following command: run jena.schemagen -i compopm.owl –a file://comp/compopm -c opmowlSchemagen.rdf –o OPM.java. An OPM vocabulary file will be created. The compopm.owl and opmowlSchemagen.rdf have to be put in the same directory as the run.bat is located; usually it is in the Jena root directory.

Once these actions are taken, one can modify the conceptualization class to include the new base model by calling addBaseModel in the JenaModelSpace.

5 Three Model Space Conceptualization Framework

The conceptualization in the Ontological Modeling Platform captures an entire application, from modeling language to individual things. A conceptualization may contain several layers of model spaces[6]. A typical configuration for a conceptualization contains three model spaces. That is, M2, M1, and M0 model spaces. Each of these model spaces has different responsibilities. For example, M2 model space manages vocabularies and provides corresponding rules to instruct how instantiation should take place. M1 model space manages specific usages of vocabularies, for example, engineering designs. The M0 model space manages intended or physical things.

However, under the current implementation, these three model spaces are all instantiated from the same model space class. The roles that the model spaces play are determined by the

[6] We consider the conceptualization as a framework because it predefines how to reuse the model spaces and low-level API to create programs for various engineering applications.

specific conceptualization. Thus, instead of the three model space conceptualization, one can create a two model space conceptualization or even a four model space conceptualization as long as their responsibilities and roles are defined clearly in the conceptualization.

5.1 *Conceptualization Class*

The conceptualization in the Composite Platform captures an entire application, from modeling language to individual things. Besides configuration operations such as creating model spaces, it also contains all the engineering design operations such as create connections and destroying connections, and checking for consistency.

5.1.1 Construct Model Spaces in Conceptualization

Usually, there will be three model space instances defined in a conceptualization that represents M2, M1, and M0 model space. In the ThreeModelSpaceConceptualization class, three protected data members are created so that its subclasses can reuse these data members (see Figure 12). These model space references will be instantiated by calling the JenaModelSpace's constructor in the ThreeModelSpaceConceptualization's constructor. Currently, the JenaModelSpace constructor will construct an in-memory model, which might be changed to a persistent storage in the future implementation.

```
protected JenaModelSpace m2ModelSpace =  null;
protected JenaModelSpace m1ModelSpace = null;
protected JenaModelSpace m0ModelSpace = null;
```

Figure 12 Three Model Space Instances Defined in a Conceptualization class

The Plate2Conceptualization must implement the abstract constructM2Space method to customize the M2 model space. In this snippet, the method sends a message to m2ModelSpace instance and asks it to add a base model by importing its schema from a local file. In the current practice, M2 model space contains only the base models, that is, the meta-models. These meta-models are pre-defined in OWL files. Thus, the constructM2Space method just needs to call the JenaModelSpace's addBaseModel method to import these pre-defined schemas to the model space.

The addBaseModel method takes a Java base model class and two URI strings as its arguments. The Java base model class is used to retrieve a prefix table and a vocabulary policy table. These two tables are added to the M2 model space so that M2 model space will have the proper prefixes registered and it will understand how to instantiate a given class. The prefix table defines the prefixes that have been used in a base model. In the composition model, we have complm, complmf, compf, and compmf. Jena will not auto detect these prefixes even if they are specified in the XML entity statement. A static predefined prefix table is accessible from JenaCompositionBaseModel; thus, we can just use it directly. The vocabulary policy table will be explained in the next section.

The other two strings specify the import URI and the location of the local schema. Note here that the CompositionBaseModel has predefined the import URI for the composition schema file. The value is a pseudo URI (file://comp). Since it is a pseudo URI, the second piece of information, the local location of the schema file, must be supplied to the addBaseModel method. This information has been predefined in a Java property file. The property file, compgen.properties, is located in root execution directory. One can also change the content of the property file with a text editor. Various schema files come with the Composition API. In the current implementation, we are using the NO_INHERIT_COMPOSITION_SCHEMA_FILE, which means that it contains a composition model without inheriting and importing from the inter-property constraint model.

5.1.2 Serialize Models to Files

The current implementation allows one to dump the RDF/XML serialization on the screen or save to the file. One can call the printConceptualization in the ThreeModelSpaceConceptualization to obtain serialization on the screen. The method has a boolean flag which allows one to printout the whole model or just the base model.

Other methods such as saveConceptualization and saveConceptualizationInOneFile provide various options on saving to files. The saveConceptualization has a parameter that will generate either four files or three files for the model spaces and conceptualization itself. The saveConceptualizationInOneFile will generate a lumped file containing all elements from all three model spaces.

5.2 M2 Model Space

Importing base models M2 model space contains base models and their corresponding vocabularies and semantic constraints. A base model is a set of predefined vocabularies that describe the most abstract modeling concepts and contextualization. For example, the concept of connection can be included in a composition base model and the concept of constraint can be included in an inter-property constraint base model. Usually, concepts defined in the base model are not instances but subtypes of the OWL class. Doing so allows the concepts can be instantiated at the M0 model space.

Vocabulary constraints It is designed that base models containing class definitions can only be loaded from files or database through the import mechanism defined by the underlying semantic language implementation, such as The Jena API or OWL API. The semantic and vocabulary constraints are associated with the base model. Their references should be established properly inside the base model so that the base model can refer to these constraints whenever needed.

No instantiation operations M2 model space should not provide any instantiation services. That is, one cannot instantiate any instances in the M2 model space. The instantiation must be realized in the lower model spaces.

No definition operations M2 model space should not allow any further definition operations. It should not allow a program to redefine and add new definitions to this model space. It is a read-only model space.

5.3 M1 Model Space

Importing base models M1 model space uses M2 model space as its base model, so that no importing mechanism except model library importing will be called in place. However, the serialization of the M1 model space must ensure that the M2 model space is properly included as an imported model[7].

[7] Currently, the subModel operations in JENA do not include an imports statement.

Vocabulary constraints M1 model space does not create new vocabulary constraints, it only obtains vocabulary constraints from the upper model space. Constraints are obtained only when such constraints are available when instantiating a specific class from the upper model space. That is, if a class defined in the M2 model space has no further constraints, no constraints will be recorded in the M1 model space.

Instantiation operations M1 model space can only instantiate classes defined in the M2 model space. M1 model space must ensure the vocabulary and semantic constraints associated with these classes are properly enforced on the instantiated resources. Some constraints will also be recorded at the M1 level, so that when the M1 resources are instantiated in the M0 model space, these recorded constraints can be enforced again.

Property-value assignment Tuples capturing properties defined in the M2 model space can be created at the M1 level. For example, one can specify a design concept plate-assembly to have a plate-connector by assigning hasConnector with plate-assembly as domain value and plate-connector as range value.

Redefine and define new concepts Unlike the M2 model space, M1 model space allows programs to define new concepts as instances of classes or properties from its upper level model space, namely, the M2 model space. For example, one can create a property and assign its domain and range types to the property. Although not defined explicitly, the OWL concepts such as Class and Property are considered also upper level concepts that can be instantiated in the M1 model space. Thus, one can create a new design concept as an instance of an OWL Class.

5.4 M0 Model Space

Importing base models M0 model space used M1 model space as its base model, so that no importing mechanism except instance library importing will be called in place. However, the serialization of the M0 model space must ensure that the M1 model space is properly included as an imported model.

Vocabulary constraints Similar to M1 model space, the M0 model space does not create new vocabulary constraints but only obtains vocabulary constraints from the upper model space. The difference in the M0 model space, it is not necessary to record the obtained constraints. These constraints are kept in the M1 model space and are accessed when instantiation takes place.

Instantiation operations M0 model space can only instantiate classes defined in the M1 model space. M0 model space must ensure the vocabulary and semantic constraints associated with these classes are properly enforced on the instantiated resources.

Property-value assignment Tuples capturing properties defined in the M1 model space can be created at the M0 level. For example, one can specify a plate-assembly-1 instance to have a part instance by assigning hasPart1 with plate-assembly-1 as domain value and part instance as range value.

No definition operations M0 model space should not provide any definition operations. That is, one cannot create new classes or define new properties in the M0 model space.

5.5 Policy

Instance creation is not a straightforward task. Different instances may require extra care. For example, in the medium level of the composition extension, an instance of a property-class must be a type of Composition.PROPERTY_CLASS and also a subtype of RDF.Statement. In order to allow JenaModelSpace to handle the variety of instantiation, these variations are encapsulated in the so-called Policy class.

A policy can apply extra constraints on the processed elements. For example, it can define additional actions that need to be taken for certain resource instantiation. In general, for the current work, we can think of two kinds of policies. One is the action policy that will take actions on the model such as adding new statements. The other is a consistency checking policy that will try to validate and return a derivation path for a given resource.

5.5.1 Policy Definition

Any base model that needs special consideration must implement the Policy interface. For The Jena API, there is a JenaPolicy class. The JenaPolicy is a class without implementation but it is not an abstract class. The reason it is designed in this way is that it allows programmers to specialize only a portion of the methods for a base model.

Current implementation defines policy at the base model level. For the composition extension, a JenaPartyPartPolicy class defines the policies for all the elements in the composition extension. For OPM there is a JenaOPMPolicy class defined. One may be able to extend this to allow the policy to work on a finer level, that is, to register only certain elements for their own policies in their corresponding base model classes. This is not implemented in the current composition platform.

5.5.2 Policy Registration

The policies are registered through a static block in the base model classes. For example, in the JenaCompositionBaseModel class, the JenaCompositionPolicy is registered for the property-class (see Figure 13).

```
static {

vocabularyPolicies.put(
      CompositionBaseModel.NAMES.MF_PROPERTY_CLASS_PC,
      JenaCompositionPolicy.getInstance());
}
```

Figure 13: Vocabulary Policy

Later on, the policy table will be passed on to the model space's policy table. Thus, in the JenaModelSpace, addPrefixAndPolicyFromBaseModel will be called when addBaseModel is called. Note that since currently we only allow M2 model space to add base model, we can only register the policies to the M2 model space. This design restriction is not really implemented but is a guideline. One may break this restriction by calling the addBaseModel from other model space.

These policies reside in the vocabulary policy table and can be accessed from the model space that defines them. For example, the JenaPropertyClassPolicy is defined to support adding the additional RDF.Statement during its instantiation. This policy is stored in the M2 model space during the addBaseModel operation. The policy will be activated when it is needed.

5.5.3 Policy Activation

```
public Individual instantiateFromUpperLevelClass(
            String upperClassURI, String individualURI){

Individual returnValue = null;

JenaPolicy policy = this.upperModelSpace.getPolicy(upperClassURI);

if(policy != null){
      Individual[] idv = new Individual[1];

      policy.createIndividual(this, upperClassURI, individualURI, idv);

      this.addPolicy(individualURI, policy);

      returnValue = idv[0];

} else {
      returnValue = ontModel.getIndividual(individualURI);

      OntClass returnClass = ontModel.getOntClass(upperClassURI);

      if(returnValue == null && returnClass !=null)

            returnValue =
                  ontModel.createIndividual(individualURI, returnClass);
      }
 return returnValue;
}
```

Figure 14: Activating Policy in Instantiation Method

In Figure 14, if the policy of a given upper class URI can be found in the vocabulary policy table, the model space will use the policy method to construct the instance. Otherwise, it will just use the ordinary Jena OWL API to construct the resource. Currently, only two methods, instantiateFromUpperLevelClass and createCurrentLevelSubClass, perform policy activation.

6 Conclusion

In this report, we describe the structure, implementation, and design decisions for creating an ontological modeling platform for product modeling. It supports extended ontological operations for composition of interconnected elements and high-level product modeling services. The platform adapted a multiple API architecture that allows further extensions on various available OWL APIs (for example, the Jena or OWL API). The logical structure provides the concept of model space to separate design concepts at different design stages. Each model space can only instantiate new instances of classes defined from the immediate upper level model space. A tutorial has also been given to demonstrate how one can construct a simple plate assembly by using the platform.

7 Acknowledgements

The authors would like to thank the funding support from U.S. National Institute of Standards and Technology/U.S. National Research Council Postdoctoral Research Associateships Program from 2005 to 2007.

Disclaimer

References

[Bock 2007a] Bock, C., "Part-part Relations in an RDF/S and OWL Extension," U.S. National Institute of Standards and Technology Interagency Report 7507, 2008.

[Bock 2007b] Bock, C., Zha, X., "Ontological Product Modeling For Collaborative Design," submitted to Advanced Engineering Informatics, 2008

[Bracha 2004] Bracha, G., "Generics in the Java Programming Language," http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf, 2004.

[Eclispse 2007] Eclipse Foundation, "Eclipse - an open development platform," from http://www.eclipse.orghttp://www.eclipse.org/, 2007.

[Gamma 1995] Gamma, E., Helm, R., Johnson, R., Vliddides, J., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional, 1995.

[OMG 2007] Object Management Group "Model-Driven Architecture," http://www.omg.org/mda, 2007.

[SourceForge 2007a] SourceForge, "Jena – A Semantic Web Framework for Java," http://jena.sourceforge.net/index.html, 2007.

[SourceForge 2007b] "OWL API," http://sourceforge.net/projects/owlapi, 2007.

[SMI 2007] Stanford Medical Informatics, "What is Protégé OWL?" http://protege.stanford.edu/overview/protege-owl.html, 2007

[TopQuadrant 2007]. "TopBraid Composer," http://www.topbraidcomposer.com, 2007.

[W3C 2007] World Wide Web Consortium, "OWL Web Ontology Language Overview," http://www.w3.org/TR/owl-features/, 2007.