Automatic mapping of OWL ontologies into Java

Aditya Kalyanpur
Maryland Information and Network Dynamics Lab
University of Maryland, College Park,
Maryland 20742, USA
aditya@cs.umd.edu

Daniel Jiménez Pastor
Department of Computer Science
University of Bath,
Bath BA2 7AY, UK
dani@pitaweb.com

Steve Battle
Hewlett Packard Labs,
Bristol BS34 8QZ, UK
steve.battle@hp.com

Julian Padget
Department of Computer Science
University of Bath,
Bath BA2 7AY, UK
jap@cs.bath.ac.uk

ABSTRACT
We present an approach for mapping an OWL ontology into Java. The basic idea is to create a set of Java interfaces and classes from an OWL ontology such that an instance of a Java class represents an instance of a single class of the ontology with most of its properties, class-relationships and restriction-definitions maintained. We note that there exist some fundamental semantic differences between Description Logic (DL) and Object Oriented (OO) systems, primarily related to completeness and satisfiability. We present various ways in which we aim to minimize the impact of such differences and show how to map a large part of the much richer OWL semantics into Java. Finally, we sketch the *HarmonIA* framework, which is used for the automatic generation of agent systems from institution specifications, and whose OWL Ontology Creation module was the basis for the tool presented in this paper.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features – *patterns, inheritance, constraints.*

General Terms
Design, Languages

Keywords
Semantic Web, Agents, Ontology, OWL, Java, HarmonIA

1. Introduction
1.1 Fundamental Issues in Mapping
It must be stated from the outset there exist fundamental differences in understanding Description Logic based (DL) and Object-Oriented (OO) systems. These differences are inherent in the nature of the KR systems themselves and hence cannot be ignored.

For instance, a class definition in an ontology (DL-based system), which consists of restrictions on a set of properties, implies:

An individual which satisfies the property restrictions, belongs to the class

However, its equivalent class definition in Java (OO system) containing a set of fields with restrictions on field-values enforced through listener functions in its accessor methods implies:

A declared instance of the class is constrained by the field restrictions enforced through the class accessor methods

The above two definitions represent dual views of the same model, and hence are not semantically equivalent. However, we lay aside these differences with the hope that the mapping from DL to OO will serve its main purpose of aiding application development.

2. The Mappings in Detail
In order to build the instance of a Java class, we use the standard **Java-beans** [1] approach to access the values of the properties of the class *(set/get methods)*. In order to maintain class relationships present in the ontology (including multiple inheritance), we use Java **interfaces** to define the OWL Classes. Finally, in order to enforce class-specific restrictions on properties, we use a set of *constraint-checker* classes that register themselves as **listeners** on the property inside the class definition, and are invoked upon property access to enforce the corresponding restriction.

2.1 Identifier Syntax
This is a perennial problem in any language translation task when the source language has a less restrictive identifier syntax than the target language. Thus, all the nominals in the ontology, which are invalid Java variables, are renamed (nominals that start with something other than a letter, underscore or $). In addition, nominals that are reserved Java keywords are also renamed appropriately.

2.2 Classes
2.2.1 OWL Basic Classes
Every OWL [2] class is mapped into a Java Interface containing just the accessor method declarations *(set/get methods)* for properties of that class (i.e. properties whose domain is specified as that class). Using an interface instead of a Java class to model an OWL class is the key to expressing the multiple inheritance properties of OWL, because Java's class language is single inheritance. In Java, an Interface can inherit from many others (unlike classes which can extend only a single class), and hence Interfaces are used in our mapping framework.

Because Java interfaces contain only static variables and abstract methods and are not by themselves instantiable, we define a corresponding Java class that embeds each interface (corresponding to an OWL class) wherein we explicitly define the fields (properties of the class) and implement the accessor methods. Additionally, based on the specified OWL property constraints, special-purpose *listeners* are registered on the corresponding field accessor methods within the Java class in order to enforce these constraints. Finally, every Interface inherits from *Thing*, an abstract Interface, needed to specify empty or unknown domains and ranges (explained later in the properties section – 2.3.1).

In the ensuing sections, we illustrate with the help of examples, mappings from OWL classes to Java using UML [3] class diagrams and fragments of Java code. The notation used in the examples is as follows: OA, OB, OC denote ontology classes, IntA, IntB, IntC represent their corresponding Java Interfaces and A, B, C are the respective Java Classes that embed the Interfaces.

2.2.2 Class axioms

2.2.2.1 equivalentClass

If OWL class *OA is equivalentTo OWL class OB* in the ontology, the corresponding Java interface and class definitions are as follows:

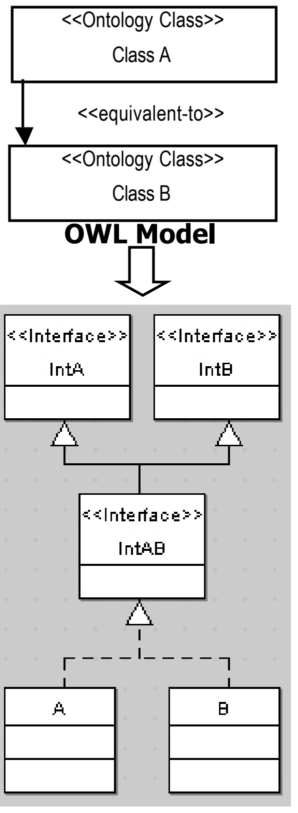
```
interface IntAB extends IntA, IntB, …Thing{}
class A implements IntAB{}
class B implements IntAB{}
```

Thus, an instance of Class A and Class B share the same set of properties and every instance of A is an instance of B and vice versa. Also, note that every interface extends *Thing* for reasons explained later (see 2.3.1)

The **Java UML Class Diagrams can be mapped** as follows:

The upper half of the diagram illustrates a putative entity model corresponding to an ontological declaration. A standard OWL model will in future be established by efforts such as the OMG call for an ontological profile for UML.

The large downward arrow represents a mapping from a standard ontological model into a more convenient (implementable) form. We understand this refactoring process as being described by transformation rules over the UML.

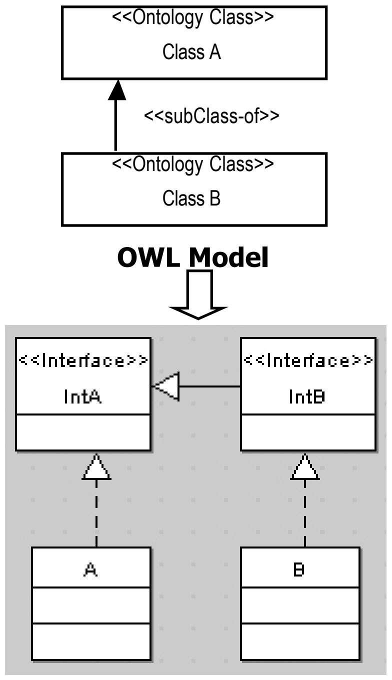


2.2.2.2 subClass

If OWL class *OB is a subClassOf OWL class OA* in the ontology, the corresponding Java Interfaces and classes are defined similarly:

```
interface IntB extends IntA, …Thing{}
class A implements IntA{}
class B implements IntB{}
```

Thus, an instance of Class B inherits all properties from Class A and every instance of B is also an instance of A.



Java UML Model

2.2.3 Enumeration classes

2.2.3.1 oneOf

Suppose OWL class OA can take *oneOf* the following instances only: IA1, IA2

In order to implement this in Java, we define appropriate **public static instances** (IA1, IA2) of the class (A) inside the class-definition itself and hide the constructor function by making it private, as follows:

```
class A {
   public static A IA1 = new A();
   public static A IA2 = new A();
   private void A() {} // hide constructor
 }
```

However, if we take into account the upcoming features of Java 1.5, we could also express the mapping like this:

```
public enum A {IA1, IA2}
```

Note: The above technique (using `enum`) can also be used to handle enumerated *DatatypeProperties* in OWL that define a range of data values.

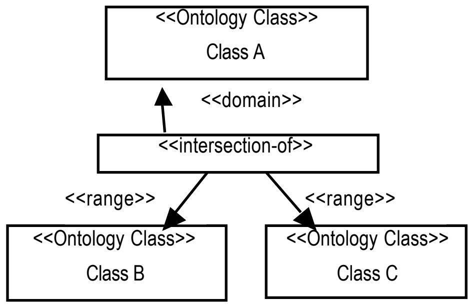
2.2.4 *OWL Logical classes*

2.2.4.1 *intersectionOf*

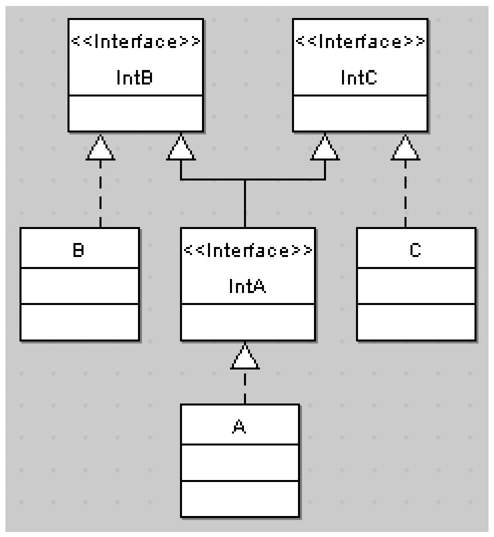
Suppose the OWL ontology specifies *OA = OB intersectionOf OC*, the Java interface and class definitions are simply:

```
interface IntA extends IntB, IntC, …Thing{}
class B implements IntB{}
class C implements IntC{}
class A implements IntA{}
```

Thus, every instance of Class A is also an instance of Class B **and** Class C, and contains (has access to) all the properties present in both the classes.



OWL Model



Java UML Model

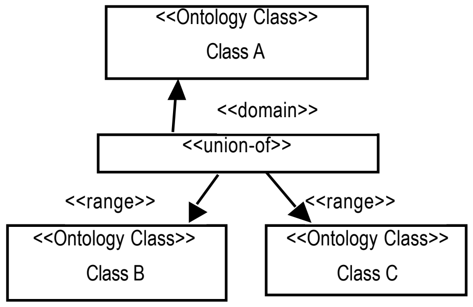
2.2.4.2 *unionOf*

If in the OWL ontology, *OA = OB unionOf OC*, the Java class definitions are rather similar:

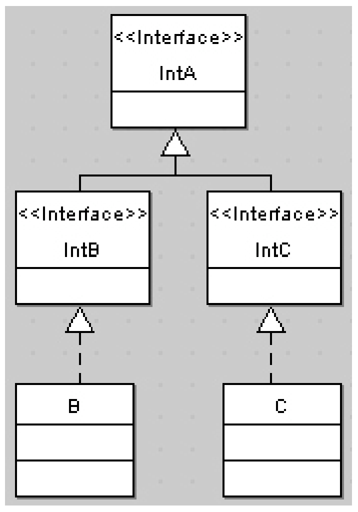
```
interface IntB extends IntA, …Thing{}
interface IntC extends IntA, …Thing{}
class B implements IntB{}
class C implements IntC{}
```

Thus, every instance of Class B **or** of Class C is **also** an instance of Class A. An instance of Class A cannot be constructed directly, it may only be instantiated by an instance of either B or C, and hence we do not create the corresponding embedding class for interface A. In addition, Interface A contains the common methods from Class B and Class C.

The intuition here is that the class OA can be satisfied by an instance of B or C, not that we have something that is *either* B or C. In OO terms, this is the *union pattern.*



OWL Model



Java UML Model

2.2.4.3 *complementOf / disjointWith*

By definition, Java does not allow an object to be declared as an instance of two or more unrelated classes, and hence the semantics of these two DL constructs are implemented by default within the Java programming model.

In order to override this feature in Java and declare a common instance of two disparate classes, the programmer needs to create a third class that is defined as an *intersectionOf* the two respective classes and then create instances of this intersection class (see 2.2.4.1).

In this case, incorporating the *complementOf / disjointWith* semantics requires some mechanism to prevent this intersection class from being created. We introduce the notion of special *blocking functions* that help us solve this problem. The basic idea is to exploit the Java function declaration model whereby function overloading cannot occur with different return types. Thus, to ensure that no interface can simultaneously extend both the interfaces corresponding to the disjoint ontological classes, we declare two identically named functions inside the respective interfaces with different return types.

For example, suppose the OWL ontology has two classes called *Male* and *Female,* which are defined as *complementOf* one another. Inside each of their respective Java Interfaces, we add a blocking function as follows:

```
interface IntFemale {
    IntFemale MaleFemaleBlocker();
}


interface IntMale {
    IntMale MaleFemaleBlocker();
}
```

Since the *MaleFemaleBlocker*() function has different return types in both the interfaces, a third interface cannot extend both the above interfaces together without incurring a compiler error. Thus, the following class does not compile:

```
interface Bisexual
                  extends intFemale, intMale {}
```

This method also accounts for subclass relationships within the OWL ontology (see 2.2.2.2). For instance, any subclass of *Female,* e.g. *Mother,* also inherits the *MaleFemaleBlocker* function (by extending the *intFemale* Interface) and hence cannot co-exist with the Interface *IntMale*.

Note: The technique shown above uses compiler intervention to detect inconsistencies in the ontology definitions, and we are currently investigating more elegant solutions (possibly using new features introduced in Java 1.5) to circumvent this.

2.3 OWL Properties

2.3.1 Domain

As mentioned earlier, all properties without a specified domain have their accessor functions declared in an abstract interface *Thin*g, which is inherited by all Java Interfaces. However, if the domain of the property is specified as the ontology class X, the corresponding Java interface *IntX* contains declarations of accessor functions for the property. In the case of a multiple-domain property (net domain is an 'intersection-of' all the classes specified as the domain), we create an intersection interface (as in section 2.2.4.1) and declare accessor functions for the property inside this interface

2.3.2 Range

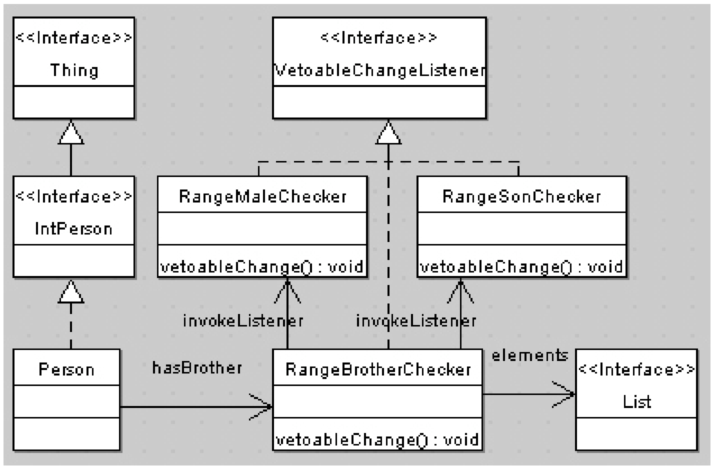
Unless stated otherwise (using appropriate restrictions), every property in OWL assumes multiple-cardinality, and hence the corresponding Java field must be of type *Collection*. Thus, OWL DatatypeProperties can be directly mapped into Java variables of the corresponding data type collection (for e.g., properties with range xsd:String to fields of type String[] etc.), and ObjectProperties to Java variables whose type is the class specified in the property's range. However, while this mechanism works fine for single-range properties, it fails to account for multi-range properties as we now discuss.

In Java, each variable can be only of one type. This contrasts with the permitted multi_range properties in OWL. A possible workaround is to declare the variable of type *List*, which is a collection of generic objects, and to incorporate appropriate checking functions to ensure that only objects in the specified range are assigned to the variable. A natural technique to program this using the Java Beans API is to define a *RangeChecker* class for each multi-range property, which implements the *VetoableChangeListener* interface and that listens to the property change events. The *RangeChecker* class contains the *vetoableChange* method that is invoked each time the property value is set by using the *fireVetoableChange* function call inside the property's accessor function. The *vetoableChange* method then checks whether the type of the object being assigned to the property is in the valid range specified in its definition, if not, it vetoes the change *throwing* the corresponding exception. In our framework, we use a hierarchy of specialized exception classes such as *PropertyRangeException*, *FunctionalPropertyException*, *MinCardinalityException*, *HasValueException* etc, which are all (in some way) subclasses of the standard *PropertyVetoException* class.

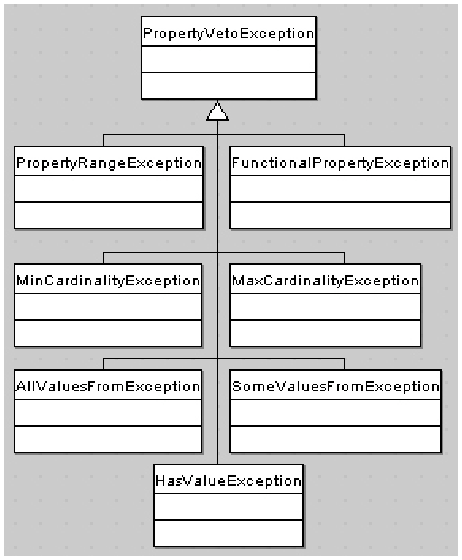
To illustrate this concept, we consider a simple example. Say, we have an OWL ObjectProperty '*hasBrother*' with its range defined as class – '*Male*'. In our Java model, *hasBrother* is defined as a field of type *List*, with an associated *RangeMaleChecker* listener that listens to (and is invoked upon) any property change event. The *vetoableChange* method inside the *RangeMaleChecker* class contains a simple *if-then* statement to check whether object(s) in the List being assigned to the property are instances of class *Male* (using Java *instanceof*), if not, it vetoes the assignment and throws a *PropertyRangeException*. In Java 1.5, we can use generics to define a List of *Male* classes, so a separate listener (*RangeChecker*) would not be needed.

For properties with multiple-ranges (interpreted as an intersection of all ranges), the above mechanism can be directly extended **as shown in the figure below**. Here, the range of the property *hasBrother* is defined as two separate classes, *Male* and *Son,* and hence the corresponding *RangeBrotherChecker* listener invokes the *RangeMaleChecker* **and** *RangeSonChecker* in turn to verify that an instance being assigned to the property satisfies both the type constraints.

Note: Ontology designers should be aware that in the case of multiple-range properties whose range is defined as two or more disjoint classes, the *intersection* semantics for multiple-range ensures that the net range is **void**. Such a conflict would be detected by our Java implementation of *disjointWith* (see section 2.2.4.3)



Multiple Range OWL Properties in Java



Hierarchy of Property Exceptions

This generic range-checking technique can easily handle single-range properties, or properties with multiple alternate ranges that are defined using the *unionOf* operator.

An added incentive for using listeners for property range checking (or even cardinality/value checking as we will see later) is that it allows arbitrary extensions to the ontology, such as additional range axioms for a property, to be integrated seamlessly i.e. by just modifying code inside the *vetoableChange* method of the appropriate listener. Moreover, if desired, it provides the user with the flexibility of dynamically enforcing only a subset of the property constraints by turning on/off the appropriate listeners.

2.3.3 Property definitions:

2.3.3.1 Functional

A functional property of a class in an ontology is one which has a unique value for each instance of the class i.e. if P is a functional property of class X, then given two triples (of the form *Subject-Predicate-Object*) about instance I1 of class X:

I1-P-I2 and I1-P-I3*,*

we can infer that the two objects are equal i.e. I2=I3

(An example of such a property is *SocialSecurityNumber* with domain *Person*, since the same instance of Person cannot have two distinct Social Security Numbers)

In order to implement this feature, we use a *FunctionalChecker* class similar to the *RangeChecker* class (described in section 2.3.2) that listens to property change events and vetoes a change if it is invalid. In this case, we need to check whether all the elements in the List passed to this property through its accessor function are equal. Thus, the *vetoableChange* method of the corresponding *FunctionalChecker* listener class looks like this:

```
void vetoableChange(PropertyChangeEvent evt) {
  List newValue = evt.getNewValue();
  for (int i=0; i<newValue.size()-1; i++)
  if (!newValue.elementAt(i)
     .equals(newValue.elementAt(i+1)))
     throw FunctionalPropertyException();
  return;
}
```

2.3.3.2 Symmetric

A symmetric property P is a property for which it holds that if the pair (x, y) is an instance of P, then the pair (y, x) is also an instance of P. To ensure this condition is satisfied in our Java framework, we use a *SymmetricChecker* class that implements the *PropertyChangeListener* (instead of the *VetoableChangeListener*) interface, listens on the specified property, and contains code to ensure symmetry inside the *propertyChange* method. Thus, for example, if the OWL ObjectProperty '*hasBrother*' has a domain and range of class '*Male*', and is defined as a symmetric property, the *propertyChange* method of the corresponding *SymmetricChecker* listener class looks like this:

```
void propertyChange(PropertyChangeEvent evt) {
  List newValue = evt.getNewValue();
  for (int i=0; i<newValue.size(); i++)
      newValue.elementAt(i)
       .addhasBrother(evt.getSource());
  return;
}
```

Here, we utilize a new accessor function *addhasBrother(Object)* that incrementally adds values to the property List rather than setting its entire contents directly (as is done by standard accessor *sethasBrother(List)*). The function also checks to see if the element to be added is already present in the List before making the change (to prevent duplication/looping)

2.3.3.3 Transitive

When one defines property P to be transitive, it means that if a pair (x, y) is an instance of P and the pair (y, z) is an instance of P, then the pair (x, z) is also an instance of P. To ensure transitivity is maintained for the property in Java, we use a *TransitiveChecker* class (listener), similar to the SymmetricChecker class described above, and write the appropriate code inside the *propertyChange* method. Thus continuing with the previous example, if the same property '*hasBrother*' is defined as transitive instead of symmetric, the code changes to this:

```
void propertyChange(PropertyChangeEvent evt) {
  List newValue = evt.getNewValue();
  for (int i=0; i<newValue.size(); i++)
  if
(newValue.elementAt(i).gethasBrother()!=null)
     evt.getSource().addhasBrother(newValue
     .elementAt(i).gethasBrother());
  return;
}
```

Here we use function polymorphism to create *addhasBrother(List)* that performs the same essential function as *addhasBrother(Object)*, with only the type of the argument being different.

2.3.4 Property Relationships:

Property relationships in the OWL ontology can be maintained in Java using the same principles described in sections 2.3.3.2 and 2.3.3.3. For each relationship, we add **appropriate accessor calls** (as explained in detail below) inside each of the *propertyChanged* method of the listeners of the concerned properties to ensure that the condition of their relationship is satisfied. However, special care needs to be taken to prevent loops from occurring while setting values of related properties together.

2.3.4.1 *equivalentProperty*

If properties P and P' are declared equivalent in the ontology, it implies that every instance pair (x, y) of P, is also an instance pair of P' and vice versa. To ensure equivalence is maintained for the property in Java, we use an *EquivalenceChecker* class (listener), similar to the listeners described in sections 2.3.3.2/3, and write the appropriate code inside the *propertyChange* method. Thus, continuing from the previous example, if property *hasBrother* is defined equivalent to the property *brother* in the ontology, we add the following code inside the *EquivalenceChecker* registered with *hasBrother*:

```
void propertyChange(PropertyChangeEvent evt) {
  List newValue = evt.getNewValue();
  if(!evt.getSource().getBrother()
    .contains(newValue))evt.getSource()
      .setBrother(newValue);
  return;
}
```

We add a similar call to the *sethasBrother* accessor function inside the *EquivalenceChecker* registered with *brother*, to complete the equivalence semantics. Also note that we use a special function *contains* to check whether the *newValue* object is already present in the value of the property before calling its accessor function, in order to prevent the system from going into an infinite loop.

2.3.4.2 *subProperty*

If a property P is defined as a *subProperty* of P' in the ontology, it implies that every instance pair (x, y) of P is also an instance pair of P', but not vice versa. Implementing its semantics in Java is very similar to the equivalence case mentioned above, the only difference being a **unidirectional** call to the accessor function of the super-property from inside the property change listener of the sub-property. Thus, if property *hasBrother* is a *subproperty* of *hasSibling* within the ontology, the code inside the *subPropertyChecker* registered with *hasBrother* looks like this:

```
void propertyChanged(PropertyChangeEvent evt) {
  List newValue = evt.getNewValue();
  if(!evt.getSource().gethasSibling()
    .contains(newValue))evt.getSource()
      .sethasSibling(newValue);
  return;
}
```

2.3.4.3 *inverseOf*

If a property P is defined as the *inverseOf* property P' in the ontology, it implies that if the pair (x, y) is an instance of P, the pair (y, x) is an instance of P'. Again, implementing its semantics in Java is similar to the two cases mentioned above, as is illustrated with an example. If the property *hasBrother* is an *inverseOf* property *isBrotherOf* in the ontology, the code inside the *InverseChecker* registered with *hasBrother* looks like this:

```
void propertyChange(PropertyChangeEvent evt) {
  List newValue = evt.getNewValue();
  for (int i=0; i<newValue.size(); i++) {
    if (!newValue.elementAt(i).getisBrotherOf()
       .contains(evt.getSource()))
    newValue.elementAt(i)
      .setisBrotherOf(evt.getSource());
  }
  return;
}
```

We add a similar loop (repeatedly calling the *sethasBrother* accessor function for each element in the *newValue* List) inside the *InverseChecker* registered with is*BrotherOf*, to complete the inverse semantics.

2.3.5 *Property Restrictions:*

The cardinality and value restrictions on properties are also enforced in the same manner as is done for range checking (described in section 2.3.2) i.e. for each restriction, we add a corresponding *CardinalityChecker* or *ValueChecker* listener on the property in its class definition. The checker class contains a *vetoableChange* method which is invoked each time the property value is set, with the appropriate constraint checking code as described in the sections below.

2.3.5.1 *Cardinality (Min/Max)*

To ensure cardinality constraints are satisfied, we need to count the number of semantically distinct elements in the List being passed to the property in its accessor function. This can be done using a separate function (we call *cardinality(List)*) that runs a loop through all the elements in the List, and removes equivalent elements before returning the final size. The checker class uses this function to perform the comparison with the values specified in the cardinality restrictions.

For example, if a property '*hasBrother'* has an *owl:minCardinality* restriction of '1'and an *owl:maxCardinality* restriction of '3', then the *vetoableChange* method of the corresponding *CardinalityChecker* listener class looks like this:

```
void vetoableChange(PropertyChangeEvent evt) {
  List newValue = evt.getNewValue();
  int number = cardinality(value);
  if (number<1)
    throw MinCardinalityException();
  else if  (number>3)
    throw MaxCardinalityException;
  return;
}
```

2.3.5.2 *HasValue*

A *hasValue* restriction on a property P specified as 'P *hasValue* I' implies that one of the values of the property must be I. In order to verify this, we use the Boolean method *contains (Object I)* on the List being passed to the property in its accessor function. Thus, for example, suppose the OWL property '*hasBrother*' has a *hasValue* restriction to individual '*John*', the *vetoableChange* method of the corresponding *ValueChecker* listener class looks like this:

```
void vetoableChange(PropertyChangeEvent evt) {
   List newValue = evt.getNewValue();
   if (newValue.contains(John)) return;
   else throw HasValueException();
}
```

2.3.5.3 *SomeValuesFrom*

A *someValuesFrom* restriction on property P specified as 'P *someValuesFrom* C' implies that at least one of the values of P must be an instance of Class C. This can be easily verified by running a single loop through all the elements in the List being passed to the property and determining whether any one of the elements is of type C. For example, if the OWL property '*hasBrother*' has a *someValuesFrom* restriction to class '*Athlete',* the *vetoableChange* method of the corresponding *ValueChecker* listener class looks like this:

```
void vetoableChange(PropertyChangeEvent evt) {
   List newValue = evt.getNewValue();
   for (int i=0; i<newValue.size(); i++)
    if (newValue.elementAt(i) instanceof Athlete)
      return;
   throw SomeValuesFromException();
}
```

2.3.5.4 *AllValuesFrom*

An *allValuesFrom* restriction on property P specified as 'P *allValuesFrom* C' implies that all values of P must be instances of class C. The verification for this is very similar to the previous case; the only difference being **all** elements in the List must satisfy the type constraint. Thus, taking the same example as above with the restriction being *allValuesFrom* instead of *someValuesFrom*, changes the constraint checking code to this:

```
void vetoableChange(PropertyChangeEvent evt) {
   List newValue = evt.getNewValue();
   for (int i=0; i<newValue.size(); i++)
     if (!newValue.elementAt(i) instanceof
Athlete)
       throw AllValuesFromException();
   return;
}
```

3. Practical Benefits of Mapping OWL to Java

Grounding the abstract conceptual space of the ontological domain in the execution space of a programming language such as Java has many significant advantages. Primarily, the Java API generated from an ontology (schema) can be used to readily build applications (or agents) whose functionality is consistent with the design-stage specifications defined in the schema. Related work [4] by Andreas Eberhart amply demonstrates this concept. In the paper referenced, the author uses a tool called *OntoJava* to automatically develop a small link recommendation system (Java applet) called '*SmartGuide'* from its RDFS [11] schema specifications. We extend its applicability by focusing on OWL, a more expressive DL than RDFS, while additionally circumventing the need for a separate rule engine. Other benefits of this mapping include the use of any Java IDE to debug (or customize) the application or ontology easily and the use of *javadoc* to generate an online documentation of the ontology automatically.

4. Related Work

4.1.1 Related projects

The concept of generating Java code from an ontology is not new. *OntoJava (*previously mentioned in section 3) is a cross compiler that translates ontologies written with *Protégé* [5] and rules written in RuleML [6] into a unified Java object database and rule engine. Similarly, the *Protégé* Bean Generator plug-in [7] can be used to generate FIPA/JADE compliant ontologies from RDF(S), XML and Protégé projects. A more comprehensive solution, is the frame-based ontology language developed by Poggi *et al* [8], where a distinct ontology language inspired by KIF has been defined for use in the context of JADE, and from which the implementation of agent systems in JADE have been generated.

Our approach tries to learn from and build upon all the projects mentioned above (many of them designed to deliver Java implementations for Agent Systems), using the latest standards (OWL) from the W3C group. Furthermore, we intend to make our tool useful not only in Agent Systems, but also in a wider range of applications (that are derived from their schema specifications), by providing great engineering flexibility while building and debugging these applications through the use of listeners and specialized exceptions (as explained in section 2.3).

4.1.2 HarmonIA

HarmonIA is a framework for the automatic generation of e-organizations from institution specifications, created at the University of Bath. Its *Ontology Creator* module, developed by Daniel Jiménez Pastor generates FIPA/JADE compliant ontologies from OWL ontology specifications using Jena 2 [9], thus serving as a natural basis for a tool that implements the work covered in this paper. A preliminary report on the framework appeared in [10]. This paper is a refinement and extension of the ontology technology, and a comprehensive discussion of the *HarmonIA* toolset is in preparation.

5. Current Status and Future Work

The *Ontology Creator* module in *HarmonIA* is being extended to handle ontological features such as multiple-inheritance among classes, multiple-range properties etc that Java does not support. Our current goal is to complete the *Ontology Creator* module by implementing all the features presented in this paper, thereby providing an automated CASE tool for the mapping of OWL into Java. This will be released open-source in the form of an API, which can be easily embedded into third party applications.

Our future goal is to design a Protégé plug-in based on our OWL-to-Java mappings, which together with the recently released OWL plug-in [12] would help provide a complete graphical framework for creating OWL ontologies and obtaining the equivalent Java UML class representation.

Finally, we wish to further enhance our mapping framework (by possibly borrowing techniques described in [4]) to include advanced property definitions (*owl:InverseFunctionalProperty*), axioms relating individuals (*owl:sameAs* etc) and OWL rules [13], thereby providing for sophisticated A-Box reasoning.

6. Conclusion

We have presented a concise and elegant solution to map an OWL Ontology into the Java Language. A summary of the mappings can be seen in Table 1. OWL has three sub-languages (OWL Lite, OWL DL and OWL Full) and we attempt to deliver a solution for the most expressive - OWL Full - at the expense of completeness, which is inevitably compromised in migrating to the OO domain. Certain OWL constructs (that primarily fall into A-Box reasoning) have not been accounted for, since incorporating their semantics is either forbidden within the Java framework, or requires a significant amount of additional code. However, ignoring them does not reduce the main utility of our mappings, which is aiding systematic application development, as was exemplified by the *HarmonIA* system.

We hope that more research continues in this direction in order to realize practical widely used applications based on Semantic Web technologies.

Table 1: Summary of OWL to Java Mappings

	OWL	**Java**
Basic Class	*A*	`interface IntA` `class A implements IntA`
Class Axioms	*A equivalentClass B*	`interface IntAB extends IntA, IntB` `class A/B implements IntAB`
	B subClassOf A	`interface IntB extends IntA`
Class Descriptions	*A = intersectionOf(B,C)*	`interface IntA extends IntB, IntC`
	A = unionOf(B,C)	`interface IntB/IntC extends IntA`
	complementOf / disjointWith	`Overridden blocking functions`
	A = oneOf(I1, I2)	`enum A{I1, I2}`
Property Associations	*P domain A*	`interface IntA {setP(List); List getP()}`
	range	`VetoableChangeListener` `(range – instanceOf, Functional – equal elements)`
Property Descriptions	*Functional*	
	InverseFunctional	-
	Symmetric	`PropertyChangeListener` `(call appropriate accessor functions inside` *`propertyChanged`*`())`
	Transitive	
Property Relationships	*equivalentProperty*	
	subPropertyOf	
	inverseOf	
Property Restrictions	*cardinality*	`VetoableChangeListener` `(check constraint satisfaction inside` *`vetoableChange`*`())`
	hasValue	
	someValuesFrom	
	allValuesFrom	
Individual Axioms	*sameAs* *differentFrom* *AllDifferent*	-

7. References

[1] Java Beans API: http://java.sun.com/products/javabeans/

[2] Word Wide Web Consortium (W3C). OWL – Web Ontology Language. http://www.w3.org/TR/owl-ref

[3] Object Management Group (OMG). UML – Unified Modelling Language. http://www.omg.org/uml/

[4] Andreas Eberhart, Automatic Generation of Java/SQL based Inference Engines from RDF Schema and RuleML. In I. Horrocks and J. Hendler, editors, Proceedings of the First International Semantic Web Conference (ISWC 2002), Chia, Sardinia, Italy, pages 102--116 , June 2002

[5] Protégé, an editor for ontologies (software package) http://protege.stanford.edu

[6] The Rule Markup Initiative: http://www.dfki.uni-kl.de/ruleml/

[7] Bean Generator plug-in for Protégé: http://gaper.wi.psy.uva.nl/beangenerator

[8] Poggi, F. Bergenti, and F. Bellifemine. An ontology description language for FIPA agent systems. Technical Report DII-CE-TR001-99, University of Parma, 1999.

[9] Jena, HP Labs Semantic Web Toolkit. http://jena.sourceforge.net/

[10] D. Jiménez Pastor and J. Padget. Towards HarmonIA: automatic generation of e-organisations from institution specifications. Workshop on Ontologies in Agent Systems (OAS'03, http://oas.otago.ac.nz/OAS2003), July 2003, Melbourne, Australia.

[11 Resource Description Framework Schema (RDFS) Specifications by the W3C. http://www.w3.org/TR/rdf-schema/

[12] OWL Plug-in for Protégé http://protege.stanford.edu/plugins/owl/

[13] I. Horrocks, P. P. Schneider, H. Boley, S. Tabet, A Proposal for an OWL Rules Language: http://www.daml.org/rules/proposal/