Automatic Mapping of OWL Ontologies into Java

Aditya Kalyanpur
Department of Computer Science
University of Maryland,
MD 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.

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 Knowledge Representation (KR) systems themselves and hence cannot be ignored.

For instance, a necessary and sufficient 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, as shown in the agent framework - *HarmonIA*.

2. 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 [1] 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 Resource Description Framework Schema (RDFS) [2] specifications. We extend its applicability by focusing on the Web Ontology Language (OWL) [3], a more expressive logical formalism 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.

3. Related Work

The concept of generating Java code from an ontology is not new. *OntoJava (*previously mentioned in section 2) is a cross compiler that translates ontologies written with *Protégé* [4] and rules written in RuleML [5] into a unified Java object database and rule engine. Similarly, the *Protégé* Bean Generator plug-in [6] 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* [7], 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. This is ensured by providing great engineering flexibility while building and debugging these applications through the use of listeners and specialized exceptions (as explained in section 4.2).

4. The Mappings in Detail

In order to build the instance of a Java class, we use the standard **Java-beans** [8] 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.

4.1 OWL Classes

Every OWL 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 in the next section).

OWL provides us with a set of complex operators (with explicit semantics) for defining classes such as *subClassOf*, *intersectionOf* and *oneOf*. Our approach maps complex OWL Class expressions to Java preserving the underlying semantics as much as possible. A summary of the mappings is shown in **Table 1**.

Note: For a more in-depth look at our mapping framework, with explanations for each type of mapping accompanied by UML diagrams and appropriate examples (with sample code), refer [9]

Table 1: OWL Class 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

	A = complementOf / disjointWith B	interface IntA {
		IntA ABBlocker()}
		interface IntB {
		IntB ABBlocker()}
		(Overridden blocking method – ABBlocker)

	A = oneOf(I1, I2)	Enum A{I1, I2}

4.2 OWL Properties

4.2.1 Domain

As mentioned earlier, all properties without a specified domain have their accessor functions declared in an abstract interface *Thing*, 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 (see **Table 1** for Java implementation of *intersection*) and declare accessor functions for the property inside this interface

4.2.2 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 of only 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*.

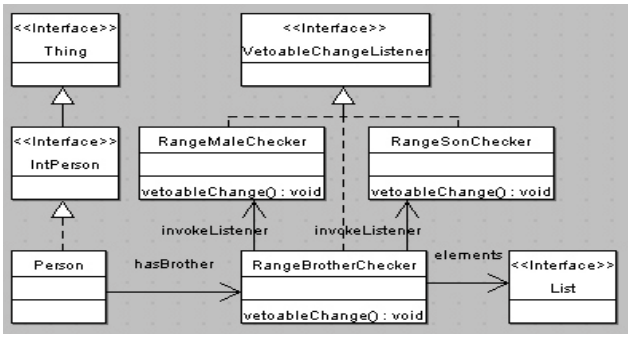

Figure 1: Multiple Range OWL Properties in Java

For properties with multiple-ranges, interpreted as an intersection of all ranges, the single-range mechanism can be directly extended as shown in **Figure 1**. Here, the range of the property *hasBrother* is defined as two separate classes, *Male* and *Son.* 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.

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 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.

Table 2: OWL Property Mappings

	OWL	Java
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	

In addition to possessing basic attributes such as *domain* and *range*, OWL Properties can be defined using a set of reserved keywords such as *Functional*, *Symmetric* and *Transitive* that have predefined semantics. Moreover restrictions can be placed on the Property's value/cardinality (using operators such as *minCardinality* and *hasValue)* in order to define *Anonymous Classes*. Our

framework considers all the semantic nuances of OWL Property expressions while mapping to Java. A summary of the mappings is shown in **Table 2** (a detailed study is provided in [9]).

5. Tying it All Together

In order to demonstrate how the various components of the mapping framework fit together, we consider a simple, yet practical scenario – a domain ontology (in OWL) used to describe *Supply Chain Management* processes in a B2B environment.

Table 3: Fragment of a sample OWL Ontology

OWL Class	Definition / Description
Company	**Primitive Class** **It's associated properties :-** **Datatype** Properties who's domain is this Class and range is **xsd:String** include *businessName*, *contactAddress*, *contactPhone, businessID* (**Functional)** **Object** Properties who's domain is this Class and range is an appropriate OWL Class (can be guessed from the property's name) include *hasBusinessCategory*, *hasWorkOrder*, *hasPurchaseOrder*
Automobile_ Company	subClassOf (*Company)* subClassOf (hasValue(*hasBusinessCategory*, "*…#Automobiles"))*
Product	**Primitive Class** **It's associated properties :-** **Datatype** Properties who's domain is this Class and range is **xsd:String** include *productName*, *productID*.. **Object** Property who's domain is this Class and range is *Company* is *hasMaker*
Buyer	intersectionOf (*Company*, ((>=1) *hasPurchaseOrder*))
Purchase_Order	subClassOf (*Order)* disjointWith (*Work_Order*)
Automobile_Product	subClassOf (*Product*) subClassOf (someValuesFrom(*hasMaker, Automobile_Company*))

Consider a subset of OWL Class definitions in the ontology as described in **Table 3.** The sample ontology contains concepts to represent *Company*, *Product*, *Purchase_Order* etc. that are integral components of a B2B supply-chain management process. Due to a lack of space, some of the related terms such as the concepts *Order* and *WorkOrder* are not displayed in Table 3. The ontology is rich in semantics, employing a suitable combination of OWL operators while defining specific concepts such as *Automobile_Company* and *Automobile_Product*.

We ground this OWL Ontology into Java using techniques described in [9] thereby illustrating the potential of our mapping framework in providing flexible execution capability while simultaneously preserving DL semantics. Space limitations restrict us from providing the complete Java code, however, **Table 4** contains some interesting fragments that we wish to discuss:

Table 4: Fragments of the equivalent Java Code

Java Element	Code

interface *IntProduct*	{..
	void setProductName(List);
	List getProductName();..
	void setHasMaker(List);
	List getHasMaker();..
	..}

interface *IntAutomobileProduct* extends *IntProduct*	{
	// Java class implementing this interface registers the *SVFAutomobileCompanyChecker* Listener on property *hasMaker*
	}

class *SVFAutomobileCompany Checker* implements VetoableChangeListener	{..
	void vetoableChange (PropertyChangeEvent evt) {
	List newValue = evt.getNewValue();
	for (int i=0; i<newValue.size(); i++)
	if (newValue.elementAt(i) instanceof *AutomobileCompany*)
	return;
	throw SomeValuesFromException();
	}
	..}

interface *IntPurchaseOrder* extends *IntOrder*	{.. *PurchaseOrder* PurchaseOrderWorkOrder Blocker() {} ..}

5.1 Discussion

Observing **Table 4,** we see that Java interface *IntProduct* corresponds to the OWL Class *Product* and contains the appropriate accessor methods (set/get) for OWL Properties *productName*, *hasMaker* etc. These accessor methods accept and return arguments of type List. In Java 1.5, we can use generics to define these Lists of type *String* and *Company* respectively. **Note that we need to have a separate Java class embedding each interface in order to create usable Java objects (instances); in this case, Java class *Product* implements the *IntProduct* interface.**

On the same lines as *IntProduct*, we generate Java interface *IntCompany* (not shown in the table) corresponding to OWL Class *Company* with its corresponding property accessors declared within.

Now focusing on the interface *IntAutomobileProduct* defined in **Table 4**, we see that it corresponds to the OWL Class *Automobile_Product*. It extends the *IntProduct* interface thereby inheriting all its property accessor methods (maintaining the *subClassOf* relationship present in the ontology). Moreover, it needs to enforce the OWL restriction (*someValuesFrom*) on property *hasMaker*. For this, a special-purpose listener class called *SVFAutomobileCompanyChecker* is created. As shown in the table, this class implements the *VetoableChangeListener* interface and contains **constraint-checking** code inside the *vetoableChange* method. This code verifies that **there exists at least one element in the List** passed to the accessor method of the concerned property **of type *AutomobileCompany***. This generic constraint checking method can handle other OWL restrictions such as *hasValue, allValuesFrom* etc. Moreover, this listener can also be registered on any other property that has the same restriction enforced in the ontology.

In this case, the listener *SVFAutomobileCompanyChecker* is registered on property *hasMaker* and fired from within its **set** accessor method inside the Java class *AutomobileProduct* that embeds the interface *IntAutomobileProduct*.

Note that an OWL restriction defines an anonymous class and we create intermediate Java interfaces to represent these anonymous OWL Classes. However, in the case of OWL Class *Automobile_Product,* since it's a subclass of OWL Class *Product,* and a restriction on property *hasMaker,* which has domain *Product,* there is no need to create an additional interface (corresponding to the anonymous class) containing the accessor methods for *hasMaker,* since these accessor methods are already declared in *IntProduct*. Our approach considers many optimization mechanisms such as this to keep the number of Java interfaces and classes generated to a minimum (see [9]).

On the same lines as *IntAutomobileProduct,* we generate Java interface *IntBuyer* (not shown in the table) corresponding to OWL Class *Buyer* that is defined by the intersection of OWL Class *Company* and *minCardinality* restriction (>=1) on property *hasPurchaseOrder* (OWL-to-Java mappings for *intersectionOf* and *minCardinality* are similar to the mappings *subClassOf* and *someValuesFrom* respectively, refer [9]).

Finally, we focus on the interface *IntPurchaseOrder* defined in **Table 4** that corresponds to OWL Class *Purchase_Order*. Since its ontological definition has it *disjointWith* OWL Class *Work_Order*, we declare two identically named blocking functions *PurchaseOrderWorkOrderBlocker*() having different return types in each of the respective Java interfaces – *IntPurchaseOrder* and *IntWorkOrder* (latter not shown in the table). In this case, the return type of the blocking method is the corresponding Java class name. Thus, a third interface cannot extend both – *IntPurchaseOrder* and *IntWorkOrder* together without incurring a compiler error, thereby preserving its disjoint semantics.

Thus, to summarize, we have shown how our proposed OWL-to-Java mapping approach maps simple ontological axioms, as well as more complex expressions built using basic 'building block' axioms, into Java, preserving the DL semantics as much as possible.

Now imagine an agent-toolkit that uses the Java API generated from the ontology. It can create instances of Java Classes *Company, Product* etc, and use its accessor functions to populate the instance, at each point, ensuring validity and consistency of the ontological model. In a multi-agent environment sharing a common global ontology, individual agents can communicate with each other seamlessly by interchanging Java Objects (instances) of appropriate Java classes (corresponding to the respective OWL classes), maintaining semantic integrity at all times.

It must be emphasized that given the nature of our framework, very large ontologies (containing >500 classes) cannot be mapped directly into Java without being broken down into smaller, more manageable ontologies. However, given that most domain ontologies in today's software systems (esp. agent-based systems) are small-to-moderately sized, our framework should suffice without significant modification.

6. Testbed Application: *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 [10], thus serves as a natural basis for a tool that implements the work covered in this paper. A preliminary report on the framework appeared in [11]. This paper is a refinement and extension of the ontology technology, and a comprehensive discussion of the *HarmonIA* toolset is in preparation.

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.

7. Future Work

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 [1]) to include advanced property definitions (*owl:InverseFunctionalProperty*), axioms relating individuals (*owl:sameAs* etc) and OWL rules [13], thereby providing for sophisticated A-Box reasoning.

8. Conclusion

We have presented a concise and elegant solution to map an OWL Ontology into the Java Language. 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.

9. References

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

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

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

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

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

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

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

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

[9] Aditya Kalyanpur et al, Detailed Technical Report on Mapping OWL to Java http://www.mindswap.org/~aditkal/OWLtoJava_Report.pdf

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

[11] 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.

[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/