A Java and OWL based Approach for System Interoperability

AGOSTINO POGGI
Dipartimento di Ingegneria dell’Informazione
Università degli Studi di Parma
Viale G.P. Usberti 181A, Parma, 43100
ITALY
agostino.poggi@unipr.it    http://www.ce.unipr.it/people/poggi

Abstract: - Semantic information is assuming more and more importance both for the development of knowledge-based applications and for supporting the interoperability among different applications. This paper presents a software framework that tries to simplify the development of such kinds of applications by providing the possibility of working on models for representing such semantic information and implementations of such models that can be easily used by software developers without any knowledge about semantic models and languages. This software library allows to represent a domain model through Java interfaces and annotations and then to use such a representation for automatically generating a Java implementation of such a domain model. Moreover, it provides the interoperability with other applications both automatically mapping the Java domain representation in an OWL ontology, and providing an automatic translation of each object obtained from the domain model representation in an OWL string representation.

Key-Words: - Knowledge-based systems, software interoperability, Java annotations, reflection, code generation, OWL.

1 Introduction

Nowadays, the use of semantic information is assuming more and more importance both for the development of applications and for supporting the interoperability among different applications. Domain models are increasingly specified as formal ontologies through the use of a semantic Web language (e.g. OWL [10]), but such models remain difficult to be utilized in applications developed through the used software languages and libraries. In fact, the mapping of such models into the code of a typical application development language often is not possible because of the different expressive power of the modeling and the implementation language. Moreover, when it is possible, the obtained implementation is too complex to be used by the large part of software developers.

However, the development of domain models that represent semantic information is very difficult without the use of a semantic language. To cope with this problem, a possible direction is to integrate usual programming techniques with some meta-programming techniques. In particular, the Java programming language supports meta-programming through annotations and reflection [7]. In fact, while annotations allow the decoration of the Java code with new concepts and idioms, reflection allows to retrieve the information associated with annotations and then to use them for either modifying the usual execution of the Java code or for building new Java code.

In this paper, we present a software framework, called JOSI (Java and OWL for System Interoperability), whose goal is to simplify that tries to simplify the development of the software libraries for managing the data that implement the domain models shared by the systems of a distributed enterprise application. The next section introduces related work on the use of annotations for the development of software and on the mapping between OWL ontologies and Java code. Section three presents the JOSI software framework. Finally, section four concludes the paper sketching some future research directions.

2 Related Work

The idea of using Java annotations for extending the Java language is not new and several research teams worked in that direction. AspectJ [9] is probably the first important work that shows how Java annotation can provide a meta-programming layer on the top of Java programming structures. In particular, AspectJ is an aspect-oriented extension of the Java programming language that uses Java annotations for realizing declaring aspects, pointcuts, and advices. Andreae et al. [2] proposed a software framework that supports pluggable type systems in
the Java programming language by the definition of custom constraints on Java types through Java annotations. AVal [11] is a software framework for the definition and checking of rules for programs written by using an attribute domain-specific built on the top of Java. This software framework allows the validation of such kinds of program through a set of predefined Java annotations. Moreover, it allows to the users of the framework to add new annotations to provide new kinds of validation. Bordin & Vardanega [5] used Java annotations to embed in the source code a declarative specification of the required concurrent semantics and then for producing the source code that implements the declared concurrent semantics. Cimadamore & Viroli [6] proposed a software framework that tried to simplify the seamless integration of Prolog code into Java applications taking advantage of Java annotations to incorporate the declarative features of Prolog into Java programs.

A lot of work has been done also towards the mapping of OWL ontologies into Java code and vice versa. The first important work that shows the partial translation of OWL ontologies in Java code is the Protégé Bean Generator [1]. In particular, it transforms Protégé frame-based ontologies into Java source code for developing JADE agents [4]. RDFReactor [15] is a toolkit for dynamically accessing an RDF model through domain-centric methods (getters and setters). In particular, it allows the access to the RDF model through a set of proxy objects that provide the methods for querying and updating the RDF elements. A more sophisticated approach was presented by Kalyanpur et al. [8]. This approach deals with issues as multiple inheritance by mapping OWL classes in Java interfaces. However, there is not a software tool which takes advantage of this approach for mapping OWL ontologies into Java code. SeRiDA [3] is a methodology for enabling a three-tier mapping along ontologies, object-oriented java beans and relational database. In particular, it allows the generation of both an object-oriented and a relational model starting from a domain conceptualization expressed in OWL. This methodology has been experimented by realizing a software tool that generates programming interfaces as enterprise Java beans and Hibernate object-relational mappings from OWL ontologies. Quasthoff & Meinel [13] presented a mechanism that allows application developers, with limited knowledge about RDF and OWL, to easily map arbitrary Java classes and interfaces to corresponding OWL concepts by using Java annotations. In particular, this mechanism has already been experimented in the development of a social network application testing new access control mechanisms on user-generated content with the help of Semantic Web rules [14].

3 Software Framework Overview

JOSI (Java and OWL for System Interoperability) is a software framework that tries to simplify the development of the software libraries for managing the data that implement the domain models shared by the systems of a distributed enterprise application.

The main features of such software library are: i) a strict separation between the representation of a domain model and its implementation, ii) an automatic generation of an implementation of the representation of the domain model in different programming languages; iii) an automatic generation of an OWL ontology from the representation of a domain model and vice versa, and iv) the possibility of using an OWL string representation of the domain model data to support the interoperability between systems implemented in different programming languages and so the possibility of translating domain model data to OWL string representations and vice versa.

JOSI is implemented in Java and takes advantage of Java interfaces and annotations to build a representation of a domain model and uses Java reflection to drive the processing of the information maintained by such interfaces and annotations for generating the source code of the classes that define the concrete implementation of the domain model.

The following subsections will describe how a domain model is represented through Java interfaces and annotations, how the Java classes providing a concrete implementation of such a domain model are generated from such interfaces and annotations, and how enable an application to use a concrete implementation of a domain model.

3.1 Representing a Domain Model

A domain model is represented by a set of Java interfaces. Each domain entity is represented by a Java interface (from here called entity interface) that defines the two methods for reading and modifying its attributes. Moreover, an additional Java interface (from here called factory interface) provides both some general information about the domain model and the factory methods for the creation of the Java classes which implement the different entity interface. Figure 1 and 2 show some entities of two domain models represented through the use of Java interfaces and annotations.
To support the creation of the implementation of such entities, each Java interface is enriched by some Java annotations and constant declarations.

The two annotations: @Getter and @Setter are applicable to the entity interface methods and define the reading and modifying methods of a specific attribute. The type of the attribute is identified by both the return type of the reading method and the type of the argument of the modifying method (of course they need to identify the same type). In particular, the value of any attribute must be: a Java primitive data, an instance of the String class, an instance of a class implementing an entity interface, or an array of the previous kinds of value.

### Fig. 1. A simple naming domain model.

The four annotations: @Abstract, @Immutable, @OneOf and @Singleton are applicable to the entity interfaces. The first annotation identifies an abstract entity, i.e., an entity that does not have any direct implementation. The second annotation identifies an entity that has an immutable implementation, i.e., the interface cannot define methods that modify the value of its attributes and the implementation of its reading methods will be defined to return either the value of an attribute (if it is an immutable value) or a copy of the value (if it is a mutable value). The third annotation is used for identifying entities that have an extensional description (e.g., an enumeration). Finally, the forth annotation is used for the definition of some special entities that can be represented by a single class object.

Often the use of an implementation of a domain model inside an application needs the availability of operations for the comparison and ordering of their entities. In a Java implementation, such operations can be performed by implementing the compareTo, equals and hashCode methods. The annotation @Comparator is introduced for this scope. In fact, it identifies the sequence of attributes on which the previous three methods must work.

The two annotations: @Name and @Version are applicable to the model interfaces: the first annotation indicates the name associated with the domain model and the second annotation identifies the version of the model.

### Fig. 2. Two entities of a domain model describing the life-cycle of a software agent.

Finally, the annotation @Binding is associated with a factory method of a model interface. This
annotation identifies the attribute that each argument of the factory method will initialize.

3.2 Constraining Attribute Values
In a domain model often is necessary both to restrict the value that some attributes can assume and to establish a relationship between the attributes of some entities. It is done by associating some additional annotations to the reading methods of the entity interfaces.

The four annotations: @AllValueFrom, @SomeValuesFrom, @Cardinality and @HasValue define the most known constraints that OWL applies to the properties of an ontology. In particular, the first annotation constrains the values of an attribute to belong to specific type (of course, an implicit constraint of such a kind, is defined when the reading and modifying methods of an attributed are defined. However, an additional constraint can be added by imposing that the values of an attribute must belong to a subtype of the declared attribute type). The second annotation imposes that some of the values of an attribute must belong to a specific type (of course, such a type must be a subtype of the declared attribute type). The third annotation imposes that an attribute can have either a fixed number of values or a variable number of values defined by a minimum and/or a maximum value. Finally, the forth annotation imposes that an attribute must always contain some values (in this case, for the limited set of value types that can be associated with the attributes of an annotation, the values of such constraints are defined through constant variables and the annotations refer to the names of such constant variables).

In some cases can be necessary to impose that an attribute does not have duplicated values and that its values are maintained ordered: the two annotations: @Set, and @Ordered impose the previous two constraints (in particular, the second constraint is implemented either by using the natural ordering between values or the ordering defined by the compareTo method built through the @Comparator annotation introduced above).

Finally, the three annotations: @InverseOf, @Symmetric, and @Transitive define the most known constraints that OWL applies to the relationship between properties of an ontology. The first annotation defines an inverse relationship between attributes. The second annotation defines a symmetric relationship between the entities that have such kind of attribute. Finally, the third annotation defines a transitive relationship between the entities that have such kind of attribute.

3.3 Generating a Domain Implementation
A domain model representation, defined as described in the previous sections, contains all the information for building an implementation of such a domain model. This implementation is realized by an annotation processor that builds a Java class for each Java interface of the model.

The result of such an annotation processor is a set of Java files. Each Java file contains the source code of a class that implement an interface of the domain model representation. Moreover, each class that implements an entity interface provides a method for building an OWL string representation of an entity class instance, and each class that implements a model interface provides a method for building an entity class instance from its OWL string representation.

The annotation processor used for generating the domain model implementation is composed by two software modules. The first module, called processing module, extracts the information from the domain model representation, generates an intermediate representation and then calls the second module. Then the second module, called generation module, builds the domain model implementation from the intermediate representation.

The intermediate representation is based on a two level tree where the root object maintains the information about the model interface and each leaf object maintains the information about an entity interface.

The processing module is independent from the implementation of the generation module because it calls a generation module by a Java interface and the generation module implementation is a parameter of the processing module constructor. Therefore, is very easy to provide different implementations of a domain model representation by defining new generation modules able to process in different ways the intermediate representation built by the processing module. In particular, the current release of the software framework provides another generation module that builds an OWL ontology from the domain model representation.

3.4 Developing an Application
The use one or more domain model implementations inside an application is very simple. In fact, the JOSI software framework
provides a class, called DataStore, which has the duty of both maintaining the information about the different domain models available for the current application and providing the access to their implementation through the creation of an instance of the class that implements their domain interface. Therefore, after the creation of an instance of the DataStore class, the code of the application can create instances of any class implementing an entity interface defined in one of the domain models available.

The JOSI software framework supports both the realization of distributed applications and a semantic interoperability between systems. In fact, it allows the realization of distributed applications where either all the programs are written in Java and so the communication between such programs is mainly based on the exchange of instances of the classes that implement the entity interfaces of the domain models involved in the application, or the programs are written in different programming languages and so the communication between such programs is only based on the exchange of OWL strings that represent the entities of the domain model involved in the application.

4 Conclusion

This paper presented a software framework, called JOSI (Java and OWL for System Interoperability), that has the goal of simplifying the development of the software libraries for managing the data that implement the domain models shared by the systems of a distributed enterprise application.

This software library allows to represent a domain model through Java interfaces and annotations and then to use such a representation for automatically generating a Java implementation of the domain model. Moreover, it provides the interoperability with other kinds of systems both automatically mapping the Java domain representation in an OWL ontology and providing an automatic translation of each object defined by the domain model representation in an OWL string representation.

We are using the JOSI software framework for the development of the models and then the implementations of the data necessary for supporting the basic interactions among the components of a distributed system realized through the HDS software framework. Moreover, JOSI was experimented for defining the domain models of some applications in the fields of distributed information sharing and social networks.

Current and future research activities are dedicated, besides to continue the experimentation of the current implementation of JOSI, to: i) the development of a software generation module that allows the automatic generation of a C++ implementation from a JOSI model representation, ii) the generation of a JOSI model representation from an OWL ontology compliant with such a representation, iii) the generation of OWL ontologies compliant with such a representation from OWL ontologies that contain classes and properties that cannot be defined through the annotations defined in JOSI, iv) the introduction of new annotations for increasing the expressive power of the JOSI model representation.

References:


