OWLBeans
From ontologies to Java classes

Michele Tomaiuolo, Federico Bergenti, Agostino Poggi, Paola Turci

***Abstract* — The Semantic Web is an effort to build a global network of machine-understandable information. Software agents should be enhanced with tools and mechanisms to autonomously access this information. The objective of this paper is to present a toolkit for extracting a subset of the relations expressed in an OWL document. It generates data structures and artifacts that can be handy for autonomous software agents to access semantically annotated information provided on the web.**

***Index Terms* — Semantic web, ontology, object-oriented systems, autonomous agents, multi-agent systems.**

I. Introduction

Semantic web promises to build a network of machine understandable information [4],[5],[9]. But to become a widespread reality, this vision has to demonstrate innovative applications, and so it is fundamental for its success to have software libraries and toolkits, enabling autonomous software agents to interface this huge source of information.

The OWLBeans toolkit, which is going to be presented in this paper, does not deal with the whole complexity of a semantically annotated web. Instead, its purpose is precisely to cut off this complexity, and provide simple artefacts to access structured information.

In general, interfacing agents with the Semantic Web implies the deployment of an inference engine or a theorem prover. In fact, this is the approach we're currently following to implement an agent-based server to manage OWL ontologies [15].

Instead, in many cases, autonomous software agents cannot, or don't need to, face the computational complexity of performing inferences on large, distributed information sources. The OWLBeans toolkit is mainly thought for these agents, for which an object-oriented view of the application domain is enough to complete their tasks.

The software artefacts produced by the toolkit, i.e., mainly JavaBeans [12] and simple metadata representations used by JADE [10], are not able to express all the relationships that are present in the source. But in some context this is not required. Conversely, especially if software and hardware resources are very limited, it is often preferable to deal only with common Java interfaces, classes, properties and objects.

The main functionality of the presented toolkit is to extract a subset of the relations expressed in an OWL document for generating a hierarchy of JavaBeans reflecting them, and possibly an associated JADE ontology to represent metadata. But, given its modular architecture, it also allows other kinds of conversions, for example to save a JADE ontology into an OWL file, or to generate a package of JavaBeans from the description provided by a JADE ontology.

II. Intermediate model

The main objective of the OWLBeans toolkit is to extract JavaBeans from an OWL ontology. But to keep the code maintainable and modular, we decided to create first an internal, intermediate representation of the ontology. In fact our tool, translating OWL ontologies to JavaBeans or vice-versa, can be viewed as a sort of compiler, and virtually every compiler builds its own intermediate model before producing the desired output. In compilers, this helps to separate the problems of the parser from those of the lexical analyzer and moreover, the same internal representation can so be used to produce different outputs. In the case of the OWLBeans toolkit, the intermediate model can be used to generate the sources of some Java classes, a JADE ontology, or an OWL file. And the intermediate model itself can be filled with data coming from different sources, obtained, for example, by reading an OWL file or by inspecting a JADE ontology.

A. Requirements

The main features we wanted for the internal ontology representation were:

- *Simplicity*: it had to include only few simple classes, to allow a fast and easy traversal of the ontology. The model had to be simple enough to be managed in scripts and templates; in fact, one of the main design goals was to have a model to be passed to a template engine, for generating the code directly from it.

Manuscript received September 27, 2004.

M. Tomaiuolo is with DII, University of Parma, Parco Area delle Scienze 181A, 43100, Parma, Italy (phone: +39 0521 905712; e-mail: tomamic@ce.unipr.it).

F. Bergenti is with DII, University of Parma, Parco Area delle Scienze 181A, 43100, Parma, Italy (phone: +39 0521 905708; e-mail: bergenti@ce.unipr.it).

A. Poggi is with DII, University of Parma, Parco Area delle Scienze 181A, 43100, Parma, Italy (phone: +39 0521 905728; e-mail: poggi@ce.unipr.it).

P. Turci is with DII, University of Parma, Parco Area delle Scienze 181A, 43100, Parma, Italy (phone: +39 0521 905708; e-mail: turci@ce.unipr.it).

- *Richness*: it had to include the information needed to generate JavaBeans and all other wanted artefacts. The main guideline in the whole design was to avoid limiting the translation process. The intermediate model had to be as simple as possible, though not creating a *metadata bottleneck* in the translation of an OWL ontology to JavaBeans. All metadata needed in the following steps of the translation pipeline had to be represented in the intermediate model. Moreover, though it had to be used mainly by template engines to generate JavaBeans, it had to be general enough to allow other uses, too.
- *Primitive data-types*: it had to handle not only classes, but even primitive data-types, as both Java and OWL classes can have properties with primitive data-types as their range.
- *External references*: often ontologies are built extending more general classifications an taxonomies, for example to detail the description of some products in the context of a more general trade ontology. We wanted our model not to be limited to single ontologies, but to allow the representation of external entities, too: classes had to extend other classes, defined locally or in other ontologies, and property ranges had to allow not only primitive data-types and internal classes, but even classes defined in external ontologies.

One of the main issues regarded properties, as they are handled in different ways in description logics and in object oriented systems. While they are first level entities in Semantic Web languages, they are more strictly related to their “owner” class in the latter model. In particular, property names must be unique only in the scope of their own class in object-oriented systems, while the have global scope in description logics. Our choice was to have properties “owned” by classes. This allows an easier manipulation of the meta-objects while generating the code for the JavaBeans, and a more immediate mapping of internal description of classes to the desired output artefacts.

B. Other models

Before deciding to create a specific internal representation of the ontology, we evaluated two existing models: the one provided by Jena [13], which of course is very close to the Semantic Web model, and the one used internally by JADE, which instead is quite close to the object-oriented model.

The first one had the obvious advantage to be the most complete model of the ontology. According to Brooks, “the world is its own best model” [22]. Nevertheless it was too complex for our scopes. For example, we wanted it to be handled by template engines, to generate Java code directly from it.

The other one, used by JADE, had most of the features we desired. But it had some major disadvantages, too. First of all, it cannot easily manage external entities; though ontologies can be organized in hierarchies, it is not possible to define the namespace of classes. Another issue is that the classes of a JADE ontology are distinguished as predicates or concepts, and predicates for example cannot be used as range of properties; this matches the semantics of the FIPA SL language [6], but could be a problem for the representation of generic OWL ontologies, as such distinction does not exist in the language. The third, and perhaps most important, issue is it does not allow exploring the tree of classes and properties from the outside.

The internal field to store classes defined in the *Ontology* class, for example, is marked private; obviously, this is a good choice to encapsulate data, but no accessor methods are provided to get the names of all classes. Other problems regard the *ObjectSchema* class that does not provide a way to get all directly extended super-classes and all locally defined slots. Finally, the *CardinalityFacet* class does not expose minimum and maximum allowed values.

In fact, the JADE ontology model was designed to allow automatic marshalling and un-marshalling of objects from FIPA ACL messages [8], and not to reason about ontology elements.

Obviously, these limitations of the JADE ontology model, proved to be a serious problem when trying to save it in an OWL file, too. This facet will be discussed in more detail in the following sections.

C. Core classes

The intermediate model designed for the OWLBeans toolkit is made of just few, very simple classes. The simple UML class diagram shown in figure 1 describes the whole intermediate model package.

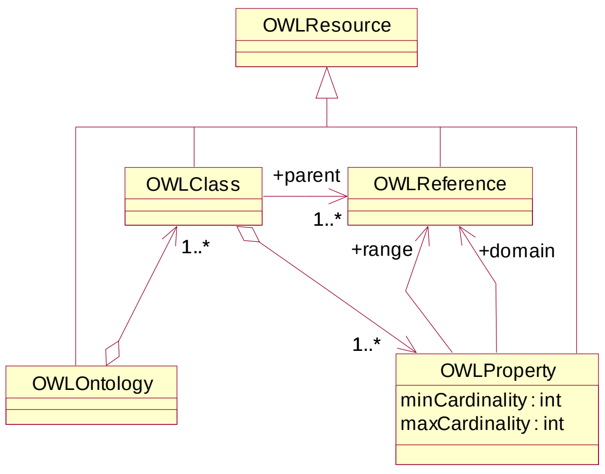

Fig. 1 - Class diagram of the intermediate model

The root class here is *OwlResource*, which is extended by all the others. It has just two fields: a local *name*, and a *namespace*, which are intended to store the same data as resources defined in OWL files. All the resources of the intermediate model – refernces, ontologies, classes and properties – are implicitly *OwlResource* objects.

OwlReference is used as a simple reference, to point to super-classes and range types, and don’t add anything to the *OwlResource* class definition. It is defined to underline the fact that classes cannot be used directly as ranges or parents.

OwlOntology is nothing more than a container for classes. In fact it owns a list of *OwlClass* objects. It inherits from *OwlResource* the name and namespace fields. In this case the namespace is mandatory and is supposed to be the namespace of all local resources, for which in fact it is optional.

OwlClass represents OWL classes. It points to a list of parents, or super-classes, and owns a list of properties. Each parent in the list is a *OwlReference* object, i.e. a name and a namespace, and not an *OwlClass* object. Its name must be searched in the owner ontology to get the real *OwlClass* object. Properties instead are owned by the *OwlClass* object, and are stored in the *properties* list as instances of the *OwlProperty* class.

OwlProperty is the class representing OWL properties. As in UML, their name is supposed to be unique only in the scope of their “owner” class. Each property points to a domain class and to a range class or data-type. Both these fields are simple *OwlReference* objects: while the first contains the name of the owner class, the latter can indicate an *OwlClass*, or an XML data-type, according to the namespace. Two more fields are present in this class: *minCardinality* and *maxCardinality*. They are used to store respectively the minimum and maximum allowed cardinality for the property values. Moreover, a *minCardinality = 0* has the implicit meaning of an optional property, while *maxCardinality = 1* has the implicit meaning of a functional property.

Probably you have already noticed the design choice to have indirect references to *OwlClass* objects in some places, in particular to point to super-classes and to allowed ranges. This decision has two main advantages over direct Java references to final objects: parsing an OWL file is a bit simpler, as references can point to classes that are not yet defined, and above all in this way super-classes and ranges are not forced to be local classes, but can be references to resources defined somewhere else.

III. Pluggable readers and writers

In our toolkit, the intermediate model is used as the glue to put together the various components needed to perform the desired, customizable task. These components are classes implementing one of the two interfaces (*OwlReader* and *OwlWriter*) representing ontology readers and writers, respectively. Not very surprisingly, readers can build an intermediate representation of the ontology, acquiring metadata from different kinds of sources, while writers can use this model to produce the desired artefacts.

The current version the toolkit provides readers to inspect OWL files and JADE ontologies, and writers to generate OWL files, source files of JavaBeans and JADE ontologies.

A very simple, yet handy application is provided, which can be customized with pluggable readers and writers, thus performing all the possible translations. While not pluggable into the main application, other components are implemented to provide additional features. For example, one of them allows to instantiate at runtime a JADE ontology and add classes to it from an intermediate ontology representation. Another component allows to load the generated code for JavaBeans directly into the Java Virtual Machine, using an embedded Java scripting engine. These components can be exploited, for example, by agent-based applications designed to be ontology agnostic, like some of those deployed in the Agentcities network [1],[2].

A. OWL files

Two classes are provided to manage OWL files. *OwlFileReader* allows reading an intermediate model from an OWL file, while *OwlFileWriter* allows saving an intermediate model to an OWL file. These two classes respectively implement the *OwlReader* and *OwlWriter* interfaces and are defined in the package confining all the dependencies from the Jena toolkit.

The latter process is quite straightforward, as all the information stored in the intermediate model can easily fit into an OWL ontology, in particular into a Jena *OntModel* object. But one particular point deserves attention. While the property names in the OWLBeans model are defined in the scope of their owner class, all OWL properties instead are first level elements and share the same namespace. This poses serious problems if two or more classes own properties with the same name, and above all if these properties have different ranges or cardinality restrictions.

In the first version of the OWLBeans toolkit, this issue is faced in two ways: if a property is defined by two or more classes then a complex domain is created in the OWL ontology for it; in particular the domain is defined as the union of all the classes that share the property, using an *owl:UnionClass* element. Cardinality restrictions are specific to classes in both models, and are not an issue. Currently, the range is instead assigned to the property by the first class that defines it, and is kept constant for the other classes in the domain. But this obviously could be incorrect in some cases. Using some class-scoped *owl:allValuesFrom* restrictions could solve most of the problems, but nevertheless difficulties would arise in the case of a property defined in some classes as a data-type property, and somewhere else as an object property.

Another mechanism allows to optionally use the class name as a prefix for the names of all its properties, hence automatically enforcing different names for properties defined in different classes. Obviously this solution is appropriate only for ontologies where names can be decided arbitrarily;

moreover it is appropriate when resulting OWL ontologies will be used only to generate JavaBeans and JADE ontologies, as in this case the leading class name would be automatically stripped off by the *OwlFileReader* class.

The inverse process, i.e. converting an OWL ontology into the intermediate representation, is instead possible only under very restrictive limitations, mainly caused by the rather strong differences between Semantic Web and object oriented languages. In fact, only few, basic features of the OWL language are currently supported.

Basically, the OWL file is first read into a Jena *OntModel* object and then all *classes* are analyzed. In this step all *anonymous classes* are just discarded. For each one of the remaining classes, a corresponding *OwlClass* object is created in the internal representation. Then all *properties* listing the class directly in their domain are considered and added to the intermediate model as *OwlProperty* objects. Here, each defined property points to a single class as domain and to a single class or data-type as range. Set of classes are not actually supported. Data-type properties are distinguished in our model by the namespace of their range, which is *http://www.w3.org/2001/XMLSchema#*. The only handled restrictions are *owl:cardinality*, *owl:minCardinality* and *owl:maxCardinality*, which are used to set the *minCardinality* and *maxCardinality* fields of the new *OwlProperty* object. The *rdfs:subClassOf* element is handled in a similar way: only parents being simple classes are taken into consideration, and added to the model.

All the rest of the information eventually being in the file is lost in the translation.

Inverse conversions are applied when writing an intermediate ontology model into an OWL file. Table 1 provides a synthetic view of these mappings.

OWL	**OWLBeans**
owl:Class	*OwlClass*
owl:ObjectProperty, owl:DatatypeProperty	*OwlProperty*
rdfs:range	*OwlProperty.range*
rdfs:domain	*OwlProperty.domain*
owl:FunctionalProperty	*OwlProperty.maxCardinality*
owl:minCardinality	*OwlProperty.minCardinality*
owl:maxCardinality	*OwlProperty.maxCardinality*
owl:cardinality	*OwlProperty.minCardinality, OwlProperty.maxCardinality*

Tab. 1 – Mappings between OWL/OWLBeans elements

B. *Template engine*

Rather than generating the source files of the desired JavaBeans directly from the application code, we decided to integrate a template engine in our project. This eventually helped to keep the templates out of the application code, and centralized in specific files, where they can be analyzed and debugged much more easily. Moreover, new templates can be added and existing ones can be customized without modifying the application code.

The chosen template engine was Velocity [19], distributed under LGPL licence from the Apache Group. It's an open source project enjoying widespread use. While its fame mainly comes from being integrated into the Turbine web framework, where it is often preferred to other available technologies, as JSP pages, it can be effortlessly integrated in custom applications, too.

Velocity template engine integration is performed through the *VelocityFormatter* class. This class hides all the implementation details of applying desired templates to an intermediate ontology and encapsulates all the dependencies from the Velocity engine. Two different types of templates are allowed, *ontology templates* and *class templates*. While the first ones only need an *OwlOntology* as parameter, the other ones also need an *OwlClass*. Ontology templates are used to generate as output the source code of JADE ontologies, for example. Class templates are instead applied to each *OwlClass* of the ontology to generate a Java interface and a corresponding implementation class, for example.

Currently, the OWLBeans toolkit provides templates to generate the source file for JavaBeans and JADE ontologies. JavaBeans are organized in a common package where, first of all, some interfaces mapping the classes defined in the ontology are written. Then, for each interface, a Java class is generated, implementing the interface and all accessor methods needed to get or set properties.

Creating an interface and then a separate implementing Java class for each ontology class is necessary to overcome the single-inheritance limitation that applies to Java classes. Each interface, instead, can extend an arbitrary number of parent interfaces. The corresponding class is eventually obliged to provide an implementation for all the methods defined by one of the directly or indirectly implemented interfaces.

The generated JADE ontology file can be compiled and used to import an OWL ontology into the JADE framework, thus allowing agents to communicate about the concepts defined in the ontology. The JavaBeans will be automatically marshalled and un-marshalled from ACL messages in a completely transparent way.

Translating an intermediate ontology to Java classes cuts off some details of the metadata level. In particular, no checks are imposed on the cardinality of property values, but only a rough distinction is made to associate non-functional properties (where *maxCardinality* is >1) with a Java *List*, to hold the sequence of values. Moreover, the class of the items of the list is not enforced, so the *range* information associated with the *OwlProperty* object is effectively lost. Instead, generating the JADE ontology does not impose the same loss of *range* and *cardinality* metadata. But nonetheless, the available set of primitive data-types is poor compared to the one of XML Schema, used in the intermediate model.

XSD	Java	JADE
xsd:Boolean	*boolean*	*BOOLEAN*
xsd:decimal, xsd:float, xsd:double	*double*	*FLOAT*
xsd:integer, xsd:nonNegativeInteger, xsd:positiveInteger, xsd:nonPositiveInteger, xsd:negativeInteger, xsd:long, xsd:int, xsd:short, xsd:byte, xsd:unsignedLong, xsd:unsignedInt, xsd:unsignedShort, xsd:unsignedByte	*int*	*INTEGER*
xsd:base64Binary, xsd:hexBinary	*Object*	*BYTE_SEQ UENCE*
xsd:dateTime, xsd:time, xsd:date, xsd:gYearMonth, xsd:gYear, xsd:gMonthDay, xsd:gDay, xsd:gMonth, xsd:duration	*Date*	*DATE*
xsd:string, xsd:normalizedString, xsd:anyURI, xsd:token, xsd:language, xsd:NMTOKEN, xsd:Name, xsd:NCName	*String*	*STRING*

Tab. 2 – Mappings between XSD/Java types

The XML data-types supported by the OWL syntax are listed in [16]. For each of them a corresponding primitive Java or JADE type must be provided. Both these conversions are not zero-cost transformations, as the target types do not express the precise meaning of their corresponding XML Schema types. Table 2 shows these conversions, as they are defined in the default templates; the "*Java*" column indicates the Java data types, while in the "*JADE*" column indicates the name of the corresponding constants defined in the JADE *BasicOntology* class.

C. Scripting engine

An additional template is provided to put the source of all interfaces, classes and JADE ontologies together, into a single stream or file, where the *package* and *imports* statements are listed only once, at the beginning of the whole file. This proves useful to load generated classes directly into the Java Virtual Machine.

In fact, if a Java scripting engine like *Janino* [11] is embedded into the toolkit, it can be exploited as a special class-loader, to load classes directly from Java source files without first compiling them into byte-code. Source files don't even need to be written to the file system first. So, at the end, JavaBeans can be loaded into the Java Virtual Machine directly from an OWL file.

Obviously, pre-compiled application code will not be able to access newly loaded classes, which are not supposed to be known at compile time. But the same embedded scripting engine can be used to interpret some ontology specific code, which could be loaded at run time from the same source of the OWL ontology file, for example, or provided to the application in other ways.

Among the various existing Java scripting engines we tested for integration into the toolkit, currently Janino proves to be the best choice. It is developed as open source project and released under LGPL license. It is an embedded compiler that can read Java expressions, blocks, class bodies and sets of source files. The Java byte-code it generates can be loaded and executed directly into the Java Virtual Machine.

While other similar engines were not able to correctly read the source files produced by the template engine, Janino made its work promptly. For example, Beanshell was not able to parse source files of interfaces with multiple inheritance, which instead is an important feature required by the OWLBeans toolkit. Thanks to its features, and to its clean design, Janino is gaining popularity. Drools, a powerful rule engine for Java, uses Janino to interpret rule scripts, and even Ant and Tomcat can be configure to use Janino as their default compiler.

The possibilities open by embedding a scripting engine into an agent system are numerous. For example, software agents for e-commerce often need to trade goods and services described by a number of different, custom ontologies. This happens in the Agentcities network, where different basic services can be composed dynamically to create new compound services.

To increase adaptability, these agents should be able to load needed classes and code at runtime. The OWLBeans package allows them to load into the Java Virtual Machine some JavaBeans directly from an OWL file, together with the ontology-specific code needed to reason about the new concepts.

D. JADE ontologies

Probably one of most interesting application of the Semantic Web is its use by autonomous software agents, which could use ontologies to reason and manipulate their environment. Their world would be made of resources and services described in ontologies, which would not be supposed to be known a priori, at compile time. The OWLBeans toolkit provides software agents the ability to load ontologies and defined classes at run time, just when they're needed or when they're discovered.

Apart from using the embedded Velocity template engine and the embedded Janino scripting engine to load generated classes at run time into the Java Virtual Machine, another component is provided to instantiate an empty JADE ontology at run time, and populate it with classes and properties read from an OWL file, or from other supported sources.

This proves useful when the agent doesn't really need JavaBeans, but can use the internal ontology model of JADE to understand the content of received messages, and to write the content of messages to send to others. The generated JADE ontology is very similar to the one produced by the Velocity template, but it doesn't need to be compiled, as no source code is generated. Instead Java objects are manipulated to create a new instance of the *Ontology* class containing all the classes and properties of the intermediate model.

The class providing this functionality is defined in the *JadeOwlOntology* class. This class does not implement the *OwlWriter* interface, but extends the *Ontology* class of JADE, adding the ability to read classes from an OWLBeans intermediate model.

Table 3 shows how the entities of one model can be mapped to the other.

Creating and populating a JADE ontology from an intermediate model is quite a straightforward process. In fact an *OwlClass* can be mapped without particular difficulties into a JADE *Schema*, while an *OwlProperty* can easily fit into a JADE *SlotDescriptor* (a private inner class of *ObjectSchemaImpl*, which can be inspected through some public methods of the outer class). The only significant difference is JADE making explicit the *AggregateSchema*, for the range of slots with *maxCardinality > 1*, and having a *TypedAggregateFacet* (i.e. a restriction) to enforce the schema of the single elements. Moreover, in a JADE ontology, *maxCardinality* and *minCardinality* are added to a slot through a *CardinalityFacet*, while in the OWLBeans model, for simplicity, they are two fields of the *OwlProperty* class.

JADE	**OWLBeans**
ObjectSchema	*OwlClass*
SlotDescriptor	*OwlProperty*
SlotDescriptor.schema	*OwlProperty.range*
SlotDescriptor.optionality	*OwlProperty.minCardinality*
CardinalityFacet.cardMin	*OwlProperty.minCardinality*
CardinalityFacet.cardMax	*OwlProperty.maxCardinality*
TypedAggregateFacet.type	*OwlProperty.range*

Tab. 3 – Mappings between JADE/OWLBeans elements

It is interesting to note that JADE defines facets, which are very similar to OWL restrictions, and which instead are missing in the OWLBeans model. This was a precise design choice to make traversing the model easier, without sacrificing needed metadata but probably loosing a bit of generality.

The *JadeReader* class encapsulates all the dependencies from the JADE framework This class does exactly what its name suggests: it "reads" an instance of a JADE ontology, and generates an intermediate model from it. Unfortunately, as we already underlined, JADE ontologies are not designed to be traversed from the outside. To be useful to inspect the content of an ontology, the model JADE uses internally lacks few accessor methods:

- it lacks a method, in the *Ontology* class, to obtain the name of all defined classes;
- it lacks a method in the *ObjectSchema* class to get the name of all defined properties;
- finally it lacks two methods to read minimum and maximum allowed cardinality, in *CardinalityFacet.*

In the implementation of the *JadeReader* class, these limitations are circumvented by using the reflection API of Java to access hidden fields and methods when necessary. Obviously, this solution can only be thought as a temporary, very limited and well documented, patch to allow JADE ontologies to be fully inspected from external code. In fact, since encapsulation is broken, even minimal modifications to the internal state representation of one of the three listed classes would stop *JadeReader* from working. We valued the possibility to export JADE ontologies to OWL files important enough to be released very soon, and thus creating such a patch proved necessary.

Anyway, the proposed modifications to the ontology API of JADE are going to be submitted to the JADE Board and to the JADE community for their introduction into the official distribution. They would make the API useful not only to extract the content of ACL messages, or to compose such messages, but even to inspect the described entities and discover some simple relationships among them. Moreover, they would not break backward compatibility, as just few methods need to be added or made public. Nothing else needs to be changed.

A particularity of the *JadeReader* class is that it silently adds some classes to the ontology it generates. These classes represent some basic FIPA types for ontology classes. FIPA SL in fact distinguishes ontology classes as *concepts*, representing objects of the model, or *predicates*, representing beliefs about the objects. Then there are more specific concepts representing *actions*, i.e. some tasks that agents can be requested to execute. The last basic class that's silently added is a concept for *agent identifiers*, or *AID*s, a class used for assigning unique names to FIPA agents [7]. Figure 2, captured from the Protégé ontology editor [17],[18], shows the hierarchy of the basic FIPA classes.

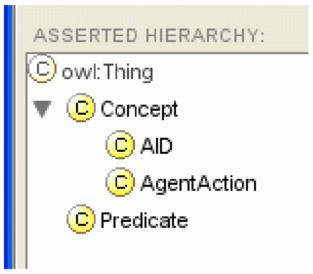

Fig. 2 – Basic FIPA classes

When the JADE ontology is traversed, each one of its schemas is checked for being an instance of a particular basic class and, accordingly, it is placed in the right branch of the generated hierarchy of classes. For example, a *ConceptSchema* class will be mapped into an *OwlClass* class having *"Concept"* among its ancestors, one of the classes added by default to the intermediate ontology soon after its creation. Similarly, a *PredicateSchema* class will instead have "*Predicate*" among its direct parents, or among its ancestors.

IV. Using the Toolkit

A customizable Java application is distributed with the toolkit. Thanks to the modular design of the whole project, this application is very simple, yet allowing to exploit almost all the functionalities of the toolkit. It simply takes the intermediate model produced by a pluggable reader, and feeds with it a pluggable writer. In this way, it can be used to realize all the format conversions made possible by combining available readers and writers. It can be used to generate Java classes from an OWL file, or to save a JADE ontology into an OWL file, or even to generate some JavaBeans adhering the descriptions provided by a JADE ontology.

The application can be execute from a shell, using the following syntax:

```
java it.unipr.aot.owl.Main [-input <input>] [-output <output>] [-package <package>] [-ontology <ontology>] [-imports (true|false)]
```

The optional arguments include the input file, the output folder for generated sources, the name of the package and the one of the ontology, and a flag to process imported ontologies. The last option is currently not yet implemented.

The following subsections show an example of execution. The first subsection shows the input ontology. The following one shows the source code generated by applying the default templates.

A. Input OWL ontology

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns="http://www.owl-ontologies.com/unnamed.owl#"
  xml:base="http://www.owl-ontologies.com/unnamed.owl">
  <owl:Ontology rdf:about="">
    <rdfs:label>Test</rdfs:label>
  </owl:Ontology>
  <owl:Class rdf:ID="AID">
    <rdfs:subClassOf>
      <owl:Class rdf:ID="Concept"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="Price">
    <rdfs:subClassOf rdf:resource="#Concept"/>
  </owl:Class>
  <owl:Class rdf:ID="Tradeable">
    <rdfs:subClassOf rdf:resource="#Concept"/>
  </owl:Class>
  <owl:Class rdf:ID="Predicate"/>
  <owl:Class rdf:ID="Book">
    <rdfs:subClassOf rdf:resource="#Tradeable"/>
  </owl:Class>
  <owl:Class rdf:ID="AgentAction">
    <rdfs:subClassOf rdf:resource="#Concept"/>
  </owl:Class>
  <owl:ObjectProperty rdf:ID="price">
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
    <rdfs:domain rdf:resource="#Tradeable"/>
    <rdfs:range rdf:resource="#Price"/>
  </owl:ObjectProperty>
  <owl:DatatypeProperty rdf:ID="authors">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="#Book"/>
  </owl:DatatypeProperty>
  <owl:FunctionalProperty rdf:ID="currency">
    <rdfs:domain rdf:resource="#Price"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
  </owl:FunctionalProperty>
  <owl:FunctionalProperty rdf:ID="value">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#double"/>
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
    <rdfs:domain rdf:resource="#Price"/>
  </owl:FunctionalProperty>
  <owl:FunctionalProperty rdf:ID="title">
    <rdfs:domain rdf:resource="#Book"/>
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  </owl:FunctionalProperty>
</rdf:RDF>
```

B. Generated Java source code

```
package bookstore;

import jade.util.leap.List;
import jade.content.onto.*;
import jade.content.schema.*;

public interface Price extends jade.content.Concept {

  public double getValue();
  public void setValue(double value);

  public String getCurrency();
  public void setCurrency(String currency);
}

public interface Tradeable extends jade.content.Concept {

  public Price getPrice();
  public void setPrice(Price price);
}

public interface Book extends Tradeable {

  public String getTitle();
  public void setTitle(String title);

  public List getAuthors();
  public void setAuthors(List authors);
}

public class PriceImpl implements Price {

  String currency;
  public String getCurrency() { return currency; }
  public void setCurrency(String currency) { this.currency = currency; }

  double value;
  public double getValue() { return value; }
  public void setValue(double value) { this.value = value; }
}

public class TradeableImpl implements Tradeable {

  Price price;
  public Price getPrice() { return price; }
  public void setPrice(Price price) { this.price = price; }
}

public class BookImpl implements Book {

  String title;
  public String getTitle() { return title; }
  public void setTitle(String title) { this.title = title; }


  List authors;
  public List getAuthors() { return authors; }
  public void setAuthors(List authors) { this.authors = authors; }

  Price price;
  public Price getPrice() { return price; }
  public void setPrice(Price price) { this.price = price; }
}

public class BookstoreOntology extends Ontology {
  public static final String ONTOLOGY_NAME = "Bookstore";

  // The singleton instance of this ontology
  private static Ontology theInstance = new BookstoreOntology();
  public static Ontology getInstance() { return theInstance; }

  // Vocabulary
  public static final String TRADEABLE = "Tradeable";
  public static final String TRADEABLE_PRICE = "price";
  public static final String PRICE = "Price";
  public static final String PRICE_VALUE = "value";
  public static final String PRICE_CURRENCY = "currency";
  public static final String BOOK = "Book";
  public static final String BOOK_TITLE = "title";
  public static final String BOOK_AUTHORS = "authors";

  public void addSlot(ConceptSchema schema, String slot, TermSchema type, int minCard, int maxCard) {
    int optionality = (minCard > 0) ? ObjectSchema.MANDATORY : ObjectSchema.OPTIONAL;
    if (maxCard == 1) schema.add(slot, type, optionality);
    else schema.add(slot, type, minCard, maxCard);
  }

  public void addSlot(PredicateSchema schema, String slot, TermSchema type, int minCard, int maxCard) {
    int optionality = (minCard > 0) ? ObjectSchema.MANDATORY : ObjectSchema.OPTIONAL;
    if (maxCard == 1) schema.add(slot, type, optionality);
    else schema.add(slot, type, minCard, maxCard);
  }

  public BookstoreOntology() {
    super(ONTOLOGY_NAME, BasicOntology.getInstance());

    try {
      PrimitiveSchema stringSchema = (PrimitiveSchema)getSchema(BasicOntology.STRING);
      PrimitiveSchema floatSchema = (PrimitiveSchema)getSchema(BasicOntology.FLOAT);
      PrimitiveSchema intSchema = (PrimitiveSchema)getSchema(BasicOntology.INTEGER);
      PrimitiveSchema booleanSchema = (PrimitiveSchema)getSchema(BasicOntology.BOOLEAN);
      PrimitiveSchema dateSchema = (PrimitiveSchema)getSchema(BasicOntology.DATE);
      ConceptSchema aidSchema = (ConceptSchema)getSchema(BasicOntology.AID);

      // Adding schemas
      ConceptSchema tradeableSchema = new ConceptSchema(TRADEABLE);
      add(tradeableSchema, Class.forName("bookstore.TradeableImpl"));

      ConceptSchema priceSchema = new ConceptSchema(PRICE);
      add(priceSchema, Class.forName("bookstore.PriceImpl"));

      ConceptSchema bookSchema = new ConceptSchema(BOOK);
      add(bookSchema, Class.forName("bookstore.BookImpl"));

      // Adding properties
      addSlot(priceSchema, PRICE_VALUE, doubleSchema, 0, 1);
      addSlot(priceSchema, PRICE_CURRENCY, stringSchema, 0, 1);

      addSlot(tradeableSchema, TRADEABLE_PRICE, priceSchema, 0, 1);

      addSlot(bookSchema, BOOK_TITLE, stringSchema, 0, 1);
      addSlot(bookSchema, BOOK_AUTHORS, stringSchema, 0, -1);

      // Adding parents
      bookSchema.addSuperSchema(tradeableSchema);

    } catch (Exception e) { e.printStackTrace(); }
  }
}
```

V. Conclusions

The OWLBeans toolkit we presented in this paper ease the access to semantically annotated information by software agents. Its main functionality is to generate JavaBeans and other artefacts, that can be used by agents needing just an object-oriented model of their application domain.

Given its modular design, the toolkit is able to process various kinds of input and produce different outputs. So, while the main purpose is to extract relations from an OWL ontology and generate JavaBeans, it can also be used to perform all other conversions allowed by combining available readers and writers.

Possible improvements include a better management of name conflicts that can arise while converting properties from an object oriented system to an ontology, where their scope is not limited to a single class. A new reader should be added to build an ontology model, using Java reflection to analyze a package of Java classes and extract needed metedata.

Above all, some relations among ontologies should be recognized and handled. In fact, having a hierarchy of ontologies, with terms of an ontology referencing terms of parent ontologies, is quite common.

References

[1] *The Agentcities Network*. http://www.agentcities.net/
[2] *Agentcities.RTD*. http://www.agentcities.org/EURTD/
[3] *BeanShell*. http://www.beanshell.org
[4] Berners-Lee, Tim. Hendler, James, Lassila, Ora. *The Semantic Web*, Scientific American, May 2001.
[5] Berners-Lee, Tim *Semantic Web Road map*, September, 1998. Available from http://www.w3.org/DesignIssues/Semantic.html
[6] FIPA spec. XC00008. *FIPA SL Content Language Specification*. Available from http://www.fipa.org/specs/fipa00008/

[7] Available from FIPA spec. XC00023. *FIPA Agent Management Specification*. Available from http://www.fipa.org/specs/fipa00023/

[8] FIPA spec. XC00037. *FIPA Communicative Act Library Specification*. Available from http://www.fipa.org/specs/fipa00037/

[9] Hendler, James, Berners-Lee, Tim and Miller, Eric *Integrating Applications on the Semantic Web*, Journal of the Institute of Electrical Engineers of Japan, Vol 122(10): 676-680, 2002.

[10] *JADE*. Available from http://jade.tilab.it

[11] *Janino*. http://janino.net

[12] *JavaBeans*. http://java.sun.com/products/javabeans/

[13] *Jena Semantic Web Framework*. http://jena.sourceforge.net/

[14] *Java Server Pages*. http://java.sun.com/products/jsp/

[15] *OWL*. http://www.w3c.org/OWL

[16] http://www.w3.org/TR/2003/WD-owl-semantics-20030203/syntax.html

[17] *Protégé Ontology Editor and Knowledge Acquisition System.* http://protege.stanford.edu/

[18] *Protégé OWL Plugin - Ontology Editor for the Semantic Web.* http://protege.stanford.edu/plugins/owl/

[19] *Velocity* http://jakarta.apache.org/velocity/

[20] *W3C Web Pages* on *Semantic Web*. http://www.w3.org/2001/sw/

[21] *XML Schema* http://www.w3.org/XML/Schema

[22] Brooks, R.A., *How to build complete creatures rather than isolated cognitive simulators*, in K. VanLehn (ed.), *Architectures for Intelligence*, pp. 225-239, Lawrence Erlbaum Assosiates, Hillsdale, NJ, 1991.