An Ontology Support for Semantic Aware Agents

Michele Tomaiuolo, Paola Turci, Federico Bergenti, and Agostino Poggi

Università degli Studi di Parma
Dipartimento di Ingegneria dell'Informazione
Viale delle Scienze, 181A – 43100 – Parma
{tomamic, turci, bergenti, poggi}@ce.unipr.it

Abstract. The work presented in this paper is an attempt to bridge two co-existing realties: Semantic Web and Multi-Agent Systems. Semantic aware agents will be able to interoperate in a semantic way as well as to produce and consume semantically annotated information and services. Agents should be enhanced with tools and mechanisms in order to autonomously achieve these strategic and ambitious objectives. In this paper, we focus on what we consider the central issue when moving towards the vision of semantic multi-agent systems: the ontology management support. Due to the heterogeneity of resources available and roles played by different agents of a system, a one-level approach with the aim of being omni comprehensive seems to be seldom feasible. In our opinion, a good compromise is represented by a two-level approach: a light ontology management support embedded in each agent and one or more ontology servers, providing a more expressive and powerful support.

Keywords: Semantic web, multi-agent systems.

1 Introduction

One of the most important challenges in agent research is the realization of truly semantic aware agents, i.e. agents that are able to interoperate in a semantic way as well as to produce and consume semantically annotated information and services, supporting automated business transactions. To achieve this goal, researchers can take advantage of semantic Web technologies and, in particular, of OWL and its related software tools.

In this paper, we concentrate on what we consider the central theme when moving towards the vision of semantic multi-agent systems: the management and exploitation of OWL ontologies. We present a two-level approach, coping with the issues of managing complex ontologies and providing ontology management support to lightweight agents.

In the next section, we examine the rationale of embedding a light ontology support in each agent of a multi-agent system. Agents refer to this ontology support when they express the content of ACL messages, e.g. the domain concepts and the relationships that hold among them. Section 3 describes the implemented library providing agents with the aforementioned two-level ontology management support.

M. Kolp et al. (Eds.): AOIS 2005, LNAI 3529, pp. 140 – 153, 2006.

Finally, Section 4 gives some concluding remarks and presents our future research directions on ontology management in multi-agent systems.

2 A Perspective on Object-Oriented vs. OWL DL Model

The scenario in which our research is situated is characterized by different domain knowledge modelling techniques and by different needs. On the one hand, there is the semantic Web and OWL [1], the most recent development in standard ontology languages. On the other hand, the popularity of the Java language for the development of multi-agent systems pushes the need for having an ontology representation more in line with the object-oriented model.

The idea behind our two-level approach originates from the awareness that agents seldom need to deal with the whole complexity of a semantically annotated Web. Our objective is hence to cut off this complexity and provide each agent with simple artefacts to access structured information. These simple artefacts are based on Java technology.

At this point a crucial question arises: are the semantics implied by the object-oriented paradigm powerful enough? A comparison between the two models (object-oriented model, e.g. the Java data model, and OWL DL) is compelling in order to understand similarities and differences, and furthermore to evaluate the feasibility of using an object-oriented representation of the ontology. As a matter of fact, the language used to build an ontology influences the kind of details that one can express or takes into consideration.

Restricting only to the semantics of the object-oriented model, i.e. without considering the possibility of defining a meta-model, what we are able to express is a taxonomy among classes[1].

Briefly, we can rephrase the object-oriented model as follows. An instance of a class refers to an object of the corresponding class. Attributes are part of a class declaration. Objects are associated with attribute values describing properties of the object. An attribute has a name and a type specifying the domain of attribute values. All attributes of a class have distinct names. Attributes with the same name may, however, appear in different classes that are not related by generalization. Methods are part of a class definition and they are used to specify the behaviour and evolution of objects[2]. A generalization is a taxonomic relationship between two classes. This relationship specializes a general class into a more specific class. Generalization relationships form a hierarchy over the set of classes.

As far as OWL is concerned, it provides three increasingly expressive sublanguages designed for use by specific communities of implementers and users. Here we focus mainly on OWL DL (called simply OWL in the following), based on $\mathcal{SHIQ}$ Description Logics. OWL benefits from years of DL research and can rely on a well defined semantics, known reasoning algorithms and highly optimized implemented reasoners.

[1] We focus on the semantics of the so called "class based" model.

[2] The dynamic properties of the model are not dealt with in this paper, focussed on the structural aspects, even if they constitute an important part of the model.

OWL, as the majority of conceptual models, relies on an object centred view of the world. It allows three types of entities: concepts, which describe general concepts of things in the domain and are usually represented as sets; individuals, which are objects in the domain, and properties, which are relations between individuals.

At first glance OWL looks like an object-oriented model. Indeed, they are both based on the notion of class: in the object-oriented model, a class provides a common description for a set of objects sharing the same properties; in OWL, the extent of a class is a set of individuals.

Behind this resemblance, there is however a fundamental and significant difference between the two approaches, centred on the notion of property.

Individual attributes and relationships among individuals in OWL are called properties. The property notion appears superficially to be the same as the attribute/component in the object-oriented model. But, looking deeply to the DL semantics on which OWL DL is based, we can see that the two notions are fairly different. Formally [2], considering an interpretation I that consist of a set Δ^I (the domain of the interpretation) that is not empty and an interpretation function $\cdot^I$, to every atomic concept A is assigned a set $A^I \subseteq \Delta^I$ and to every atomic role R a binary relation $R \subseteq \Delta^I \times \Delta^I$. By means of the semantics of terminological axioms, we can make statements about how concepts and even roles are related to each other (e.g. $R^I \subseteq S^I$ inclusion relationship between two roles). What is clear is that roles in DL, and therefore OWL DL properties, are first-class modelling elements. Most of the information about the state of the world is captured in OWL by the interrelations between individuals. In other words, data are grouped around properties. For instance, all data regarding a given individual would usually be spread among different relations, each describing different properties of the same individual.

Differently, the object-oriented representation relies on the intentional notion of class, as an abstract data type (partially or fully) implemented [3], and on the extensional notion of object identifier. An object is strictly related and characterized by its own features including attributes and methods. In other words, data are grouped around objects, thought of as a collection of attributes/components.

As a consequence, in OWL it is possible to state assertions on properties that have no equivalent in the object-oriented semantics. Properties represent without any doubt one of the most problematic differences between OWL and object-oriented models.

To conclude, we can say that grounding the conceptual space of the ontological domain to a programming language such as Java has several obvious advantages but also some limitations. What we intend to do in next sub-section is an analysis of the weaknesses of the object-oriented representation compared to OWL, and to verify if its expressive power is powerful enough to capture the semantics of the agent knowledge base. In this study, we take into consideration that agents do not often need to face the computational complexity of performing inferences on large, distributed information sources; rather, they often simply need to produce and validate messages that refer to concepts of a given ontology.

2.1 Mapping OWL to Java

During the past years, much research work has been devoted to deal with the comparison between OWL and UML [4-5]. Among these, some considered the

mapping related to a particular object-oriented programming language: Java. Focussing on these, we can essentially identify two major directions followed by the research community in order to express the OWL semantics using the Java language.

1. The definition of a meta-model that closely reflect the OWL syntax and semantics. Examples are the modelling APIs of Jena [6-7] and OWL API [8-9]. The latter consists of a high-level programmatic interface for accessing and manipulating OWL ontologies. Its aim is to implement a highly reusable component suitable for applications like editors, annotation tools and query agents.
2. The use of the Java Beans API [10] to realize a complete mapping between the two meta-models. In particular, to cope with the central issue, i.e. the property-preserving transformation, [10] defines suitable *PropertyChecker* classes in order to support the semantics of the property axioms and restrictions. However, in our opinion, this approach lacks an explicit meta-model and therefore the corresponding explicit information. Moreover, it cannot be supported by a reasoner because of the impracticality of implementing one.

Our approach differs from those listed above since it aims at offering a two-level support: the most powerful one is based on Jena; the other is based on the object-oriented semantics.

When establishing a correspondence between two models, it is important to understand what the purpose of the mapping is. For example, the aim of having a full mapping and preserving the semantics is satisfied when using the Jena toolkit, whereas it is too strong in the case of the lightweight support. In the latter case, we decided to relax this constraint and consider a partial mapping, required only to be consistent (in the sense that it does not preserve semantics but only semantic equivalence [4]). This means that there is a one-to-one correspondence between instances of one model and the instances of the other model that preserves relationships between instances. This lets us use, for example, renaming and redundancy in order to achieve this goal, as in the use of interfaces in Java in order to express multiple inheritance.

For the sake of clarity and in order to avoid a lengthy dissertation, in the following we consider only the more salient aspects of the mapping, analysing commonalities as well as dissimilarities, and ending, in the successive sub-section, by delineating the application sphere of our approach.

Every OWL class is mapped into a Java interface containing the accessor method declarations (getters and setters) for properties of that class (properties whose domain is specified as this class). Then, for each interface, a Java class is generated, implementing the interface. Creating an interface and then separately implementing a Java class for each ontology class is necessary to overcome the single-inheritance limitation that applies to Java classes. In OWL, there is a distinction between named classes (i.e. primitive concepts), for which instances can only be declared explicitly, and defined classes (i.e. defined concepts), which specify necessary and sufficient conditions for membership. Java does not support this semantics and so only primitive concepts can be defined. In the following we refer only to named classes.

Individuals in OWL may be an instance of multiple classes, without one being necessarily a subclass of another. This is in contrast with the object-oriented model:

an object could get the properties of two classes only by means of a third one which has both of them in its ancestors. A workaround is thus to create a special subclass for this notion.

Considering the *terminological axioms* used to express how classes are related to each other, the only one that has an equivalent semantics in Java is the OWL synopsis *intersectionOf* (mapped as an interface which implements two interfaces). The *unionOf* OWL synopsis can be mapped in Java defining an interface as a super-interface of two interfaces but, in order to ensure the semantic equivalence, it is compulsory to prevent the implementation of the super-interface.

The constructs asserting completeness or disjointness of classes are those which characterized more OWL, from the point of view of the "open-world" assumption, i.e. modelling the state of the world with partial information. In OWL, classes are overlapping until disjointness axioms are entered. Moreover, generalization can be mutually exclusive, meaning that all the specific classes are mutually disjoint and/or complete, meaning that the union of the more specific classes completely covers the more general class. In Java, there is no way of expressing it and other similar properties (e.g. *equivalentClass*); the representation of the world that we can state using this model can only refer to a "closed-world" assumption. This obviously constitutes a limitation when one cannot assume that the knowledge in the knowledge base is complete.

Regards properties, since they are not first-class modelling elements in Java, it is not possible to create property hierarchies and to state that a property is symmetric, transitive, equivalent or the inverse of another property. Properties can be used to state relationships between individuals (*ObjectProperty*) or from individuals to data values (*DatatypeProperty*). DatatypeProperties can be directly mapped into Java attributes of the corresponding data type and ObjectProperties to Java attributes whose type is the class specified in the property's range. In OWL there are constraints that can be enforced on properties:

1. Cardinality constraints state the minimum and maximum number of objects that can be related;
2. The "domain" constraint limits the individuals to which the properties can be applied;
3. The "range" constraint limits the individuals that the property may have as its value.

Java accessor methods could ensure that cardinality constraints be satisfied. This information, however, is implicit and embedded in the class source code and it would not become known to a possible reasoner and therefore it would be most likely of no use.

Concerning the domain restriction, if the property domain is specified as a single class, the corresponding Java interface contains declarations of accessor methods for the property. In the case of a multiple domain property, there are two possible alternatives:

1. The domain is an *intersection-of* all the classes specified as the domain; to cope with this we create an intersection interface (see above).
2. Multiple alternative domains are defined using the *unionOf* operator; we can cope with this creating a union interface but with the limitations expressed above.

Finally, in relation to the range restriction, our approach fails to account for multi-range properties, since variables in Java can be only of one type.

It clearly emerges, from the previous analysis, that the Java language expressiveness is lower even than OWL Lite but, despite this, in our view, it is still valuable with respect to the common agent needs.

2.2 Reasoning About Knowledge

Although DLs (and hence OWL DL) and object-oriented models have a common root in class-based models, they were developed by different communities and for different purposes. The different target applications significantly affect the expressiveness of the languages and consequently the reasoning services that can be performed on the corresponding knowledge base.

The object model only permits the specification of necessary conditions for the class (i.e. the definition of the properties that must be owned by objects belonging to a specific class) that are not sufficient to identify a member of the class. The only way to associate an instance to a class is therefore to explicit assert its membership. As a consequence some basic reasoning services lose their importance and significance (e.g. knowledge base consistency, subsumption and instance checking). A fairly common complex reasoning service, i.e. classification, also plays a marginal role in an object-oriented model. In fact, in DL, the terminological classification consists in making explicit the taxonomy entailed by the knowledge base. Whereas the classification of individuals has its role in DL, since individuals can be defined giving a set of their properties and therefore objects' classes, membership can be dynamically inherited.

The previous remarks lead us to consider the aspect that differentiates even more between the two models, that is the divergent assumption on the knowledge about the domain being represented - open vs. closed world assumption. Indeed while a DL-based system contains implicit knowledge that can be made explicit through inference, a system based on an object-oriented model exhibits a limited use of entailment. Inheritance may represent a simple way of expressing implicit knowledge (a class inherits all the properties of its parents without explicit specifying it). Another way is to represent part of the information within methods (e.g. initialization methods), but this implicit information is not (or hard) available to a potential reasoner.

If we consider the knowledge base as a means of storing information about individuals, an interesting complex reasoning task is represented by retrieval. Retrieval (or query answering) consists in finding all the individuals in the knowledge base in a concept expression. The information retrieval task plays a leading role in a knowledge base centred on an object-oriented representation.

3 System Architecture

The concrete implementation of the proposed system is a direct result of the evaluations set out in the previous sections. The proposed two-level approach to

ontology management is implemented as a framework providing the following functionality:

1. Light support: to import OWL ontologies as an object-oriented hierarchy of classes;
2. Ontology Server: to provide the centralized management of shared ontologies.

3.1 OWLBeans

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

In general, interfacing agents with the Semantic Web implies the deployment of an inference engine or of a theorem prover. In fact, this is the approach we are currently following to implement an agent-based server to manage OWL ontologies. Instead, in many cases, autonomous agents cannot (or do not need to) face the computational complexity of performing inferences on large, distributed information sources. The OWLBeans framework 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 framework, i.e. mainly JavaBeans and simple metadata representations used by JADE [11], are not so expressive as OWL-DL. 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, attributes and objects. Its main functionality is to extract a subset of the relations expressed in an OWL document for generating a hierarchy of JavaBeans, and possibly for creating a corresponding JADE ontology to represent metadata. However, given its modular architecture, it also provides other functionality, e.g. to save a JADE ontology into an OWL file or to generate a package of JavaBeans from the description provided by a JADE ontology.

Intermediate ontology model. In order to keep the code maintainable and modular, we decided to base the framework on an internal, intermediate representation of the ontology. This intermediate model can be alternatively used to generate the sources of some Java classes, a JADE ontology or an OWL file. The intermediate model itself can be filled with data obtained, e.g. by reading an OWL file or by inspecting a JADE ontology.

The main design goals of the internal ontology representation were:

1. *Simplicity*: it had to include only few simple classes to allow a fast and easy introspection 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 be directly used by a template engine to generate the code.
2. *Expressiveness*: it had to include the information needed to generate JavaBeans and all other desired artefacts. The main guideline 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.

3. *Primitive data-types*: it had to handle not only classes, but even primitive data-types, since both Java and OWL classes can have properties using primitive data-types as their range.
4. *External references*: ontologies are often built extending more general classifications and taxonomies. For example, an ontology can 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 may extend other classes, defined locally or in other ontologies, and property ranges may allow not only primitive data-types and internal classes, but also classes defined in external ontologies.

One of the main issues related to properties, since they are handled in different ways in description logics and in object-oriented systems (see the previous section). For the particular aims and scope of OWLBeans, property names must be unique only in the scope of their own class in object-oriented systems, while they have global scope in description logics. Our choice, in the internal model design, 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.

The intermediate model designed for the OWLBeans framework is made of just a few, very simple classes. The simple UML class diagram shown in Fig. 1 describes the main classes of the intermediate model package.

The root class 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 model – refernces, ontologies, classes and properties – are implicitly *OwlResource* objects.

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

OwlOntology is nothing more than a container for classes. 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 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 an *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

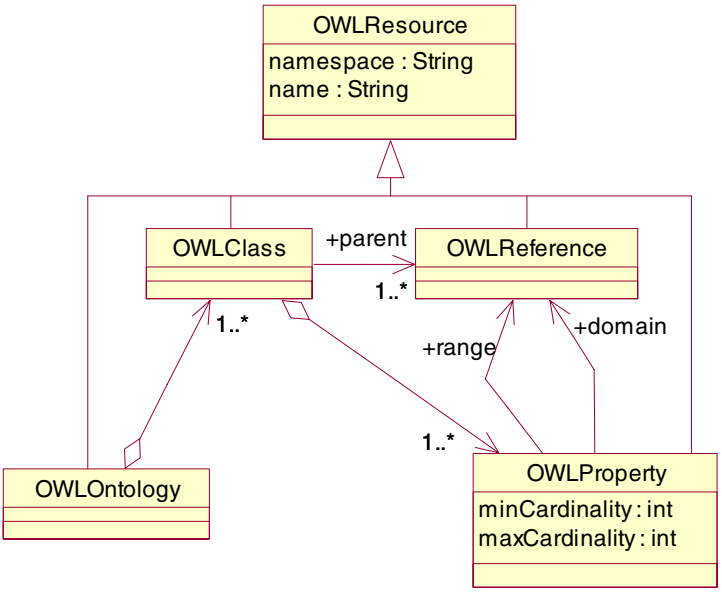

Fig. 1. Class diagram of the intermediate model

used to store respectively the minimum and maximum allowed cardinality for the property values. A *minCardinality* = 0 has the implicit meaning of an optional property, while *maxCardinality* = 1 has the implicit meaning of a functional property.

It is worth pointing the unusual treatment of indirect references to *OwlClass* objects. This decision has two main advantages over direct Java references to final objects. Parsing an OWL file is a bit simpler, since references can point to classes that are not yet defined. Furthermore, in this way, super-classes, domain and ranges are not forced to be local classes, but can be references to resources defined somewhere else.

In our framework, 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 the *OwlReader* or the *OwlWriter* interface, representing ontology readers and writers, respectively. While readers can read an intermediate representation of the ontology, acquiring metadata from different kinds of sources, writers, instead, can use this model to produce the desired artefacts.

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

Reading OWL Ontologies. 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 direct process, i.e. converting an OWL ontology into the intermediate representation, is possible only under quite restrictive limitations, mainly caused by

the rather strong differences between the OWL data model and the object-oriented model. In fact, only few, basic features of the OWL language are supported.

Basically, the OWL ontology is first read into a Jena *OntModel* object and then all classes are analysed. 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 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: *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 remaining information in the OWL file is lost in the translation, as it does not fit into the desired object-oriented model.

Generating JavaBeans. 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 helped to keep the templates out of the application code, and centralized in specific files, where they can be analysed 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 [12], distributed under LGPL licence by the Apache Group. It is an open source project with a widespread group of users. 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.

Currently, the OWLBeans framework provides templates to generate the source file for JavaBeans and JADE ontologies. JavaBeans are generated according to the mapping between classes and concepts that we described in the previous sections. In particular, all 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.

As stated in Section 2, 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.

The generated JADE ontology file can be compiled and used to import an OWL ontology into JADE, 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.

Additional components. Additional components are provided to read and write ontologies in different formats.

For example, the *JadeReader* class allows the loading of a JADE ontology and saving it in OWL format or generating the corresponding JavaBeans.

Another component is provided to instantiate an empty JADE ontology at run time, and to populate it with classes and properties read from an OWL file or from other supported sources. This proves useful when the agent does not really need JavaBeans but can use the internal ontology model of JADE to manage the content of semantically annotated messages.

Finally, the *OwlWriter* class allows an ontology to be converted from its intermediate representation to an OWL model. This is quite straightforward, since all the information stored in the intermediate model can easily fit into an OWL ontology, in particular into a Jena *OntModel* object. 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 are instead 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 framework, 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 assigned to the property by the first class that defines it and is kept constant for the other classes in the domain. Obviously this could be incorrect in some cases. Using some class-scoped owl*:allValuesFrom* restrictions could solve most of the problems, but 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 the optional use of the class name as a prefix for the names of all its properties, hence automatically enforcing different names for properties defined in different classes. This solution is appropriate only for ontologies where property names can be decided arbitrarily. Moreover, it is appropriate when resulting OWL ontologies are used only to generate JavaBeans and JADE ontologies, since in this case the leading class name would be automatically stripped off by the *OwlFileReader* class.

Scripting Engine. The possibilities opened by embedding a scripting engine into an agent system are various. For example, 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 [13], 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.

This is achieved by embedding *Janino* [14], a Java scripting engine, into the framework. Janino can be used as a special class loader capable of loading classes directly from Java source files without first compiling them into bytecode.

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

3.2 Ontology Server

The OWLBeans framework allows agents to import taxonomies and classifications from OWL ontologies, in the form of an hierarchy of Java classes. Clearly, a more general solution must be provided for all those cases where a simplified, object-oriented view of the ontology is not enough.

For all those applications, that need a complete support of OWL ontologies, we are developing an Ontology Server. It is an agent-based application providing ontology knowledge and reasoning facilities for a community of agents. The main rationale for building on Ontology Server is to endow a community of agents with the ability to automatically process semantically annotated documents and messages. The Ontology server shares a common knowledge base about some application domains with this community of agents.

The first functionality is related to loading, importing, removing ontologies. Apart from loading ontologies at agent startup, specific actions are defined in terms of ACL requests to add ontologies to the agent knowledge base, and to remove them. Ontologies that are linked through import statements can be loaded automatically with a single request. Moreover, new relations among ontologies can be dynamically created, and existing ones can be destroyed. This import mechanism can be used to build a distributed knowledge base hierarchy; in this way, a new ontology can be plugged in easily and inherit the needed general knowledge base, instead of building it totally from scratch.

After the initial set-up, through a number of potentially related ontologies, this knowledge base can be queried from other agents. A set of predicates is defined, to check the existence of specific relations among entities. For example the Ontology Server can be asked about the equivalence of two classes or about their hierarchical relationships.

Apart from checking the existence of specific relations, the knowledge base can also be used to search for the entities satisfying certain constraints. For example, the list of all the super-classes, or of all the sub-classes, of a given class can be obtained.

Finally, client agents may be allowed to modify an ontology managed by the Ontology Server. Agents can ask to add new classes, individuals and properties to the ontology or to remove defined entities. Moreover, relations among ontology entities can also be added and removed at runtime.

Our current implementation is built as a JADE agent, using the Jena toolkit to load and manage OWL ontologies. An inference engine can be plugged into the application to reason on the knowledge base. An ontology is defined, to allow the management of the internal knowledge base. ACL requests, to access and query the Ontology Server about its knowledge base, can use this meta-ontology to represent their semantic content.

As a final point, for the Ontology Server to be really useful in an open environment, we are adding proper authorization mechanisms. In particular, we are leveraging the underlying JADE security support to implement a certificate-based access control. Only authenticated and authorized users will be granted access to managed ontologies. The delegation mechanisms of JADE allow the creation of communities of trusted users, which can share a common ontology, centrally managed by the Ontology Server.

Finally, we are developing a graphical user interface to allow the interaction with the Ontology Server through Web pages. It allows both the introspection of the existing knowledge base, as well as its modification by human users.

4 Conclusion

In this paper, we have presented a software implementation intended to provide an OWL ontology management support for multi-agent systems implemented by using JADE. The key feature that distinguishes our approach from others is the fact that lightweight agents have the possibility of directly managing ontologies that can be mapped in JavaBeans, while they can take advantage of special agents, called Ontology Servers, when they need to use more complex. Well aware of the need to clearly define the weakness of our approach in comparison to a fully-fledged OWL support, we have carried out a meticulous analysis of its expressiveness.

Our current activities are related to the experimentation of the implemented software in the realization of a multi-agent system for the remote assistance of software programmers. Furthermore, we are working on its improvement by trying alternative solutions to the use of the Jena software tool.

References

1. Word Wide Web Consortium (W3C). OWL. Web Ontology Language. http://www.w3.org/TR/owl-ref.
2. The Description Logic Handbook: Theory, Implementation and Applications. Cambridge University Press (2002)
3. Meyer, B.: Object-Oriented Software Construction. Prentice-Hall, 2nd edition (1997)
4. Baclawski, K., Kokar, M.K., Kogut, P., Hart, L., Smith, J.E., Letkowski, J., Emery, P.: Extending the Unified Modeling Language for ontology development. International Journal Software and Systems Modeling (SoSyM) 1(2) (2002) 142-156
5. Hart, L., Emery, P., Colomb, B., Raymond, K., Taraporewalla, S., Chang, D., Ye, Y., Kendall, E., Dutra, M.: OWL Full and UML 2.0 Compared (2004). http://www.omg.org/docs/ontology/04-03-01.pdf.
6. Carroll, J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., Wilkinson, K.: Jena: Implementing the Semantic Web Recommendations. In Proc 13th Int World Wide Web Conference, New York, NY (2004) 74-83
7. Jena, HP Labs Semantic Web Toolkit software and documentation. http://jena.sourceforge. net/.
8. Bechhofer, Volz, R., Lord, P.: Cooking the Semantic Web with the OWL API. In Proc. Intl Semantic Web Conference, Sanibel Island, FL, USA (2003) 659-675

9. OWL API software and documentation. http://owl.man.ac.uk/api.shtml.
10. Kalyanpur, A., Pastor, D., Battle, S., Padget, J.: Automatic Mapping of Owl Ontologies into Java. In Proceedings of Software Engineering .and Knowledge Engineering Conference. (SEKE) 2004, Banff, Canada (2004)
11. JADE software and documentation. Available at http://jade.tilab.com.
12. Velocity software and documentation. Available at http://jakarta.apache.org/velocity.
13. The Agentcities Network project home page. http://www.agentcities.net.
14. Janino software and documentation. Available at http://janino.net.