OWLET: An Object-Oriented Environment for OWL Ontology Management

Agostino Poggi
Dipartimento di Ingegneria dell'Informazione
Università degli Studi di Parma
Parco Area delle Scienze 181A, Parma, 43100
ITALY
http://www.ce.unipr.it/people/poggi

Abstract: - In this paper, an object-oriented model and a software environment for the management of OWL ontologies is presented. The object-oriented model allows a simple and complete representation of ontologies defined by using OWL DL profile. The software environment, called OWLET, implements this object-oriented model and provides a complete set of reasoning functions together with a graphical editor for the creation and modification of ontologies. OWLET can be very useful for realizing heterogeneous and distributed semantic systems where nodes differ for their capabilities (i.e., CPU power, memory size, …); in fact, it offers a layered reasoning API that allows to deploy a system where high power nodes take advantages of all the OWLET reasoning capabilities, medium power nodes take advantages of a limited set of OWLET reasoning capabilities (e.g., reasoning about individuals) and low power nodes delegate reasoning tasks to the other nodes of the system.

Key-Words: - OWL, ontology management, Object-Oriented model, Ontology reasoning, Heterogeneous distributed systems, Java, …

1 Introduction

The mapping of an OWL ontology [1] into an object-oriented representation can be very useful for increasing the diffusion of ontologies and semantic Web technologies. In fact, the availability of such a representation can be the basis for the development of some flexible and efficient software libraries for the management of ontologies that allow to cope with the limits of the current software libraries and tools for the realization of ontology based and semantic Web applications..

The main problem of this mapping is that there are important semantic differences between OWL and an object-oriented language and so it is difficult to provide an object-oriented mapping that both minimizes the need of writing code manually and full satisfies OWL semantics.

OWL allows the definition of classes and properties as specialization of multiple classes and properties. Therefore, the object-oriented languages that provide multiple inheritance would seem to be the most suitable for representing OWL ontologies. However, the use of multiple inheritance can cause conflict because a subclass can inherits the same variable or method from different classes. These inheritance clashes are usually resolved by the subclass either redefining the conflicting variable or method for itself or by specifying which inheritance is preferred. These inheritance clashes are possible in representing OWL ontologies (e.g., when an OWL class can inherits a restriction on the same property from different classes) and so they must be managed through the manual or automatic generation of some additional code.

OWL ontologies can be represented also by using object-oriented languages that do not provide multiple inheritance. For example, some previous approaches coped with this problem by using Java interfaces [2],[3]. This solution only partially solves the problem because interfaces allow the definition of class variables and methods, while instance variables and methods code must be provided by the classes implementing the interfaces. Therefore, the representation of OWL ontologies requires the manual or automatic generation of a large amount of additional code.

Another problem of representing OWL classes and properties with classes of an object-oriented language is the mapping of OWL class and property names into class names of the object-oriented language. In fact, the most known object-oriented languages have restrictions on the syntax of class names different from the ones imposed by the OWL language. In this case, the solution is to: i) change the OWL class and property names on the basis of the restrictions of the target language (e.g., *trading-price* may be changed to *trading_price* for defining a Java or C++ class) and ii) avoid the introduction of name conflicts (e.g., *trading-price* and *trading+price* cannot be both changed into *trading_price*).

A solution for avoiding the previous problems, is the decomposition of inheritance into the more basic mechanisms of object composition and message forwarding [4]. Therefore, for example, an OWL class contains (the references to) its super classes, does not inherit their features, but can get/modify them through the methods provided by the super classes. Moreover, as done in other approaches, the problem of representing an OWL ontology is separated from the problem of acting and reasoning on it. This solution allows the definition of a very simple OWL ontology model based on few classes, that respectively define the variables for maintaining the components of a particular kind of OWL resource and implement the methods for getting and setting their values. Therefore, an OWL ontology is described by a set of instances of the classes respectively representing an OWL class, property and individual.. Moreover, this solution avoids the problem of mapping OWL resource names in admissible identifiers of the used object-oriented language, because the name of an OWL resource become a value that is stored into the corresponding variable of the OWLET instance representing such a resource.

2 OWLET Ontology Model

The OWLET ontology model provides a complete representation of OWL DL ontologies, is based on six Java classes: *OwlOntology*, *OwlClass*, *OwlDatatype*, *OwlProperty*, *OwlRestriction* and *OwlIndividual* and an OWL ontology is represented by a set of instances of the previous classes. Moreover, OWL ontologies and ontology resources can be identified through a variable of the previous classes that maintains the ontology/resource full URI.

Id	Uri
Classes	$C = \{c_1, \dots, c_n\}$
equivalentClasses	$C_e = \{c_1, \dots, c_n\}$, $C_e \subseteq C$
disjointClasses	$C_d = \{c_1, \dots, c_{dn}\}$, $C_d \subseteq C$
Datatypes	$D = \{d_1, \dots, d_n\}$
Properties	$P = \{p_1, \dots, p_n\}$
equivalentProperties	$P_e = \{p_1, \dots, p_n\}$, $P_e \subseteq P$
individuals	$I = \{i_1, \dots, i_h\}$
equivalentIndividuals	$I_e = \{i_{e1}, \dots, i_{eh}\}$, $I_e \subseteq I$
differentIndividuals	$I_d = \{I_{d1}, \dots, I_{dn}\}$, $I_d \subseteq I$

Table 1. The *OwlOntology* class variables.

An OWL ontology is represented by an instance of the *OwlOntology* class and contains information about: i) all the classes, properties and individuals that are defined or referred in such an ontology, and ii) the equivalence and difference relationships among them (see table 1).

An OWL class is represented by an instance of the *OwlClass* class and contains information about: i) the class name, ii) the inheritance and composition relationships with some other ontology classes, iii) the composition relationships with some ontology individuals, and, finally, iv) the restrictions on ontology properties (see table 2).

Id	Uri
subClassOf	$C_{sc} = \{c_1, \dots, c_n\}$, $C_{sc} \subset C$
unionOf	$C_{uo} = \{c_1, \dots, c_n\}$, $C_{uo} \subset C$
complementOf	c_c, $c_c \in C$
one of	$I_{oo} = \{i_1, \dots, i_n\}$, $I_{oo} \subseteq I$
Restrictions	$R_i = \{r_i, \dots, r_n\}$

Table 2. The *OwlClass* class variables.

An OWL data type is represented through a subclass of the *OwlDatatype* class. In particular, while all the OWL predefined data types (i.e., the XML Schema data types and the RDF literal data type) are represented by the instance of a “singleton” class, the enumerated data types are represented by instances of the DataRange class.

Id	Uri
Type	v, $v \in$ {Object, Datatype}
Domain	$C_d = \{c_1, \dots, c_n\}$, $C_d \subseteq C$
Range	$T_r = \{t_1, \dots, t_m\}$, $t_r \subseteq D$ if type = Datatype $t_r \subseteq C$ if type = Object
subPropertyOf	$P_{sp} = \{p_1, \dots p_n\}$, $P_{sp} \subset P$
Functional	b, $b \in$ {true, false}
Transitive	b, $b \in$ {true, false}
Symmetric	b, $b \in$ {true, false}
inverseOf	p_j, $p_j \in P$
inverseFunctional	b, $b \in$ {true, false}

Table 3. The *OwlProperty* class variables.

All the types of OWL properties (i.e., *Annotation*, *Datatype*, *Object* and *Ontology* properties) are represented by instances of the *OwlProperty* class. While annotation and ontology properties only

contain the information about their type, *Datatype* and *Object* properties also contain information about the different property facets (see table 3).

classId	Uri
propertyId	Uri
maxCardinality	n, $n \geq 0$
minCardinality	n, $n \geq 0$
allValuesFrom	$T_{avf} = \{t_1, \ldots, t_n\}$, $T_{avf} \subseteq C$ if p_j.type = Object $T_{avf} \subseteq D$ if p_j.type = Datatype
someValuesFrom	$T_{svf} = \{t_1, \ldots, t_n\}$, $T_{svf} \subseteq C$ if p_j.type = Object $T_{svf} \subseteq D$ if p_j.type = Datatype
hasValue	$V_{hv} = \{v_1, \ldots, v_n\}$, $V_{hv} \subseteq I$ if p_j.type = Object $V_{hv} \subseteq D$ if p_j.type = Datatype

Table 4. The *OwlRestriction* class variables.

The set of restrictions that must be applied to the values of a specific property of the individuals belonging to a specific OWL class are grouped together and represented by an instance of the *OwlRestriction* class. This class maintains the information about the possible kinds of restriction that can be applied to a *Datatype* or *Object* property (see table 4).

An OWL individual is represented by an instance of the *OwlIndividual* class and contains information about: the classes to which the individual belongs and the property-values pairs describing the individual (see table 5).

Id	Uri
individualOf	$C_{io} = \{c_1, \ldots c_n\}$, $C_{io} \subseteq C$
Values	$V = \{V_1, \ldots V_n\}$, $V_j = \{v_1, \ldots v_n\}$, $v_{jk} \in I$ if p_j.type = Object $v_{jk} \in D$ if p_j.type = Datatype

Table 5. The *OwlIndividual* class variables.

3 OWLET Ontology Representation

Although a large part of the knowledge represented by OWL constructs can be directly mapped into equivalent entities of the OWLET classes, some of such knowledge needs more complex elaborations.

For example, the OWLET model does not provide any entity for maintaining the knowledge represented by the OWL intersection construct. It is because an OWL class defined as the intersection of some other classes is equivalent, from the semantic point of view, to a class defined as the subclass of these other classes, and because an OWL class defined as the intersection of a set of property restrictions can be represented, in the OWLET model, by a class defined as composition of such a set of restrictions.

OWLET provides a parser that maps OWL DL ontologies, represented in the OWL/RDF format, into an object-oriented representation based on the OWLET model.

For example, given the OWL/RDF fragment of figure 1, describing the *WhiteWine* class of the Wine ontology [5], then the OWLET parser creates an *OwlClass* instance for representing the *WhiteWine* class and adds a reference to the *Wine* instance to its *subClassOf* variable. Moreover, it creates an *OwlRestriction* instance for representing the restriction on the *hasColor* property and adds it to the restrictions variable of the *WhiteWine* instance.

```
<owl:Class rdf:ID="WhiteWine">
  <owl:intersectionOf
rdf:parseType="Collection">
    <owl:Class rdf:about="#Wine" />
    <owl:Restriction>
      <owl:onProperty
rdf:resource="#hasColor"/>
      <owl:hasValue rdf:resource="#White"/>
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>
```

```
<owl:Class rdf:ID="Wine">
  <rdfs:subClassOf
    rdf:resource="&food;PotableLiquid"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty
        rdf:resource="#madeFromGrape"/>
      <owl:minCardinality
        rdf:datatype="&xsd;nonNegativeInteger">
1   </owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
  ...
</owl:Class>
```

Fig. 2. The *Wine* class definition.

The restrictions defined inside an OWL subclass axiom are managed in the same way. For example, given the OWL/RDF fragment of figure 2, describing the *Wine* class of the *Wine* ontology [5], then the OWLET parser creates an *OwlRestriction* instance for representing the restriction on the *madeFromGrape* property and adds it to the restrictions variable of the *Wine* instance.

Anonymous classes can be used in an OWL ontology. For example, given the OWL/RDF fragment of figure 3, taken from the *Wine* ontology [5] and describing the *NonFrenchWine* class as the intersection between the class *Wine* and an anonymous class, then the OWLET parser creates two *OwlClass* instances for representing the two classes of the intersection and assigns to the second instance a special identifier having the following form:

<AnonymousClassID> =
<OntologyURI> ‘unnamedClass’ <counter>

where <counter> is an integer that is incremented each time a new anonymous class is found. This naming solution avoids the conflict between the names of anonymous classes of different ontologies.

```
<owl:Class rdf:ID="NonFrenchWine">
 <owl:intersectionOf rdf:parseType="Collection">
  <owl:Class rdf:about="#Wine"/>
  <owl:Class>
   <owl:complementOf>
    <owl:Restriction>
     <owl:onProperty rdf:resource="#locatedIn"/>
     <owl:hasValue rdf:resource="#FrenchRegion"/>
    </owl:Restriction>
   </owl:complementOf>
  </owl:Class>
 </owl:intersectionOf>
</owl:Class>
```

Fig. 3. The *NonFrenchWine* class definition.

```
<Measurement>
  <observedSubject rdf:resource="#JaneDoe"/>
  <observedPhenomenon rdf:resource="#Weight"/>
  <observedValue>
    <Quantity>
      <quantityValue
rdf:datatype="&xsd;float">59.5</quantityValue>
      <quantityUnit rdf:resource="#Kilogram"/>
    </Quantity>
  </observedValue>
  <timeStamp rdf:datatype="&xsd;dateTime">
    2003-01-24T09:00:08+01:00
  </timeStamp>
</Measurement>
```

Fig. 4. An anonymous *Measurement* individual.

Anonymous individuals are managed in the same way. For example, given the OWL/RDF fragment of figure 4, taken from [1] and describing an individual as composition of two anonymous individuals respectively belonging to the *Measurement* and *Quantity* classes, then the OWLET parser creates two *OwlIndividual* instances for representing the two individuals and assigns them an identifier similar to the one assigned to anonymous classes:

<AnonymousIdividualID> =
<OntologyURI> ‘unnamedIndividual’ <counter>

where <counter> is an integer that is incremented each time an anonymous individual is found. Also in this case, the naming solution avoids the conflict between the names of anonymous individuals of different ontologies.

```
<owl:Class>
  <owl:unionOf rdf:parseType="Collection">
    <owl:oneOf rdf:parseType="Collection">
      <owl:Thing rdf:about="#Tosca"/>
      <owl:Thing rdf:about="#Salome"/>
    </owl:oneOf>
    <owl:Restriction>
      <owl:onProperty
        rdf:resource=" #composed-by"/>
        <owl:hasValue rdf:resource="#Puccini"/>
    </owl:Restriction>
  </owl:unionOf>
</owl:Class>
```

Fig. 5. An anonymous class defined as union of an enumeration and a restriction.

```
…
<owl:Restriction>
  <owl:onProperty rdf:resource="#hasDrink"/>
  <owl:allValuesFrom>
    <owl:Restriction>
     <owl:onProperty rdf:resource="wine:hasSugar"/>
     <owl:hasValue rdf:resource="#Dry"/>
    </owl:Restriction>
  </owl:allValuesFrom>
</owl:Restriction>
…
```

Fig. 6. A restriction on the *hasDrink* property.

In some cases, the OWLET parser needs to introduce some additional anonymous classes to represent an OWL class defined as composition among intersections of classes, restrictions and individuals. For example, given the OWL/RDF fragment of figure 5, taken from and representing an anonymous class as union of an enumeration, that contains the *Tosca* and *Salome Opera* individuals, and of a restriction on the property *composed-by*, then the OWLET parser creates two anonymous *OwlClass* instances for respectively encapsulating the enumeration and the restriction.

Another example of the need of additional anonymous classes is presented by the OWL/RDF

fragment of figure 6 that is taken from [6]. This fragment describes an *allValuesFrom* restriction that limits the values of the *hasDrink* property to the individuals whose *hasSugar* property has *Dry* as a value. In this case, the OWLET parser creates an anonymous *OwlClass* instance for encapsulating the restriction on the *hasSugar* property and adds this *OwlClass* instance to the *allValuesFrom* variable.

4 OWLET Reasoning Tools

OWLET provides a set of tools for reasoning on ontologies. In particular, it implements algorithms for ontology consistency, class and data type satisfiability, subsumption, equivalence, individual instantiation, classification and equivalence, and property consistency and equivalence.

Some of the reasoning algorithms (i.e., ontology consistency and class satisfiability) are not realized through a direct processing on the OWLET ontology representation, but are realized through a new OWL ontology representation obtained from the OWLET representation by only transforming the OWLET class representation into a new class representation that defines an OWL class in a disjunction normal form on individuals and restrictions involved in its definition.

In fact, given an OWL class, described through the OWLET model and expressed by the form:

$$(1)\ c_x = R_x \cap SuperC_x \cap U_x \cap I_x \cap cc_x$$

we can apply a recursive expansion to: i) the super classes ($SuperC_x$), ii) the union classes (U_x), and iii) the complement class (cc_x). Finally, we can combine all the restrictions (R_x) of the expanded classes transforming (1) into the equivalent form:

$$(2)\ c_x \equiv I'_x \cup (R'_1 \cap NI'_1) \cup \ldots \cup (R'_n \cap NI'_n)$$

where: I'_x and R'_i are respectively a set of individuals and a set of restrictions participating in the definition of c_x, and NI'_i is a set of individuals participating in the definition of the complement class of c_x.

The class representation described by (2) has the advantage of checking class satisfiability of an OWL class by simply checking either if the are some individuals involved in the definition of a class (I'_x is not empty) or if there are at least a term, R'_i, that can be satisfied by at least an ontology individuals that is not member of the corresponding set of individuals, NI'_i.

5 Conclusion

In this paper, an object-oriented model and a software environment, called OWLET, for the management of OWL ontologies has been presented.

The object-oriented model allows a simple and complete representation of ontologies defined by using OWL DL profile. The software environment is realized by using the Java programming language and, besides implementing the object-oriented model and providing an API for the creation and manipulation of OWL ontologies, offers a complete set of reasoning functions for ontology consistency, class and data type satisfiability, subsumption, equivalence, individual instantiation, classification and equivalence, and property consistency and equivalence. Moreover the creation and manipulation of OWL ontology is simplified thanks to graphical editor (see figure 7) that allows: the visualization of the relationships among classes, properties and individuals, the creation, modification and deletion of both new classes, properties and individuals, and of relationships among them.

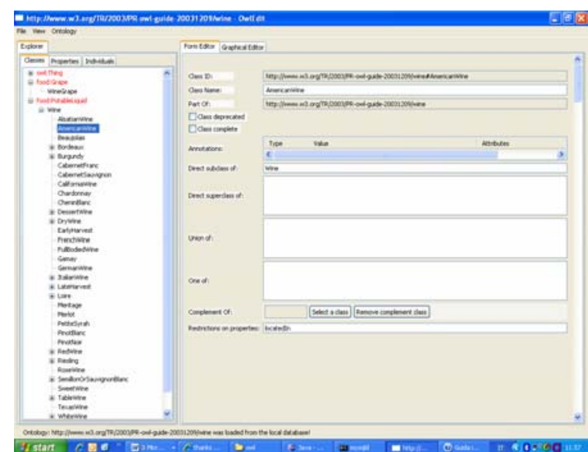

Fig. 7. A view of the OWLET ontology graphical editor.

OWLET can be considered an interesting environment for the development of ontologies, but, in particular, it can be very useful for realizing heterogeneous and distributed semantic systems where nodes differ for their capabilities (i.e., CPU power, memory size, …); in fact, it offers a layered reasoning API that allows to deploy a system where high power nodes take advantages of all the OWLET reasoning capabilities, medium power nodes take advantages of a limited set of OWLET reasoning capabilities (e.g., reasoning about individuals) and low power nodes delegate reasoning tasks to the other nodes of the system..

Future work on the OWLET system will be related to: i) the realization of an accurate performance

analysis and its comparison with the most known systems for OWL ontology management and reasoning (e.g., FACT++ [7] and RACER [8]), ii) the enhancement of the ontology development tools (e.g., the introduction of a 2D or 3D visualization of the graphs representing the relationships among the different ontology resources), and iii) the continuation of its use and experimentation for the realization of systems in both the semantic Web and in the e-business application fields.

References:

[1] Dean, M., Schreiber, G., OWL Web Ontology Language Reference, W3C Recommendation, 2004. Available, 10 February 2004. Available at http://www.w3.org/TR/owl-ref/.

[2] Bergenti, B., Poggi, A., Tomaiuolo, M., Turci. P. An Ontology Support for Semantic Aware Agents. In. Proc. Seventh International Workshop on Agent-Oriented Information Systems (AOIS-2005), Utrecht, the Netherlands, 2005.

[3] Kalyanpur, A., Pastor, D., Battle, S., Padget, J. Automatic mapping of owl ontologies into java. In Proceedings of Software Engineering. - Knowledge Engineering. (SEKE) 2004, Banff, Canada, 2004.

[4] Frohlich, P.H. Inheritance decomposed. In Proc. of the Inheritance Workshop at ECOOP 2002, Malaga, Spain, June 2002.

[5] W3C Consortium. The wine OWL ontology, 2004. Available at http://www.w3.org/TR/2004/REC-owl-guide-20040210/wine.rdf.

[6] W3C Consortium. The food OWL ontology, 2004. Available at http://www.w3.org/TR/2004/REC-owl-guide-20040210/food.rdf.

[7] Tsarkov, D., Horrocks, I. FaCT++ description logic reasoner: System description. In Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006), Lecture Notes in Artificial Intelligence, Volume 4130, pp. 292-297. Springer, 2006.

[8] Haarslev, V. Möller, R. Racer: A Core Inference Engine for the Semantic Web. In Proc. 2nd Int. Workshop on Evaluation of Ontology-based Tools (EON2003), Sanibel Island, Florida, USA, October 20, pages 27–36, 2003.