World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:1, No:4, 2007

LOWL: Logic and OWL, an Extension

M. Mohsenzadeh, F. Shams, M. Teshnehlab

Abstract**—** Current research on semantic web aims at making intelligent web pages meaningful for machines. In this way, ontology plays a primary role. We believe that logic can help ontology languages (such as OWL) to be more fluent and efficient. In this paper we try to combine logic with OWL to reduce some disadvantages of this language. Therefore we extend OWL by logic and also show how logic can satisfy our future expectations of an ontology language.

Keywords**—** Logical Programming, OWL, Language Extension.

I. Introduction

Nowadays, much research has been done on the topic of information integration and knowledge combination. These researches aim at making a connection between users and heterogeneous information systems that can be used in intranet and internet environments. Within such framework, ontology plays a pivotal role, because it provides a common and shared agreement of a specific domain. In current computer science, ontology is said to be "an agreement about a shared, formal, explicit and partial account of a conceptualization" [1]. We can also say that ontology contains the vocabulary (concepts and terms) and the definition of these concepts and their relationships for a specific domain.

RDF is one of the first languages that were used for ontology representation. The expressivity of RDF is deliberately very limited [2]. OWL is the last standard of ontology language that was created by W3C. This language is base on RDF(S) and DAML+OIL, and uses XML syntax to represent ontology. More information on OWL can be found in [2].

Efficient reasoning support is one of the main requirements for an ontology language [2]. Over the last decades, reasoning about logical theories has been studied well. It seems that Description Logic (decidable fragment of FOL) can be used as an appropriate formalism for representing and reasoning about ontology.

The aim of this paper is to employ logic in OWL. Combining DL and OWL can improve several problems in using OWL. We can refer to some of them as follows: (1) The syntax of OWL is very longsome and complicated. Logic can grant a simple and fluent syntax to OWL. This is explained in section 2. (2) Logic can facilitate learning OWL. The logic syntax is more popular in comparison to OWL syntax. (3) OWL has been developed recently, and there is not many tools for it yet. But researches on logic have been continued for some decades and therefore there are many tools for logic. If we establish OWL on logic, we can use these tools. (4) In the near future, we expect much expressive ontologies for real applications. Obviously, we need database techniques to deal with such ontologies. Since we have many tools for logic that use database technology, using logic for OWL can satisfy our expectations in the future.

The rest of this paper is organized as follows. First, we give an overview on OWL. Also, we show how OWL syntax can be expressed with logic. In section 3 we present a primary mapping from OWL primitives to LP. Next (section 4), we check our method for ontology language requirements. In the next section, we try to change our mapping to satisfy the ontology language requirements. We will end with a discussion about our approach and OWL layers that we managed to support.

II. Towards Using Logic in OWL

In this section we show how OWL primitives can be expressed by logic syntax. We start with a simple example.

Example 1: A professor is an academic staff member. To express this example in OWL, we define a *professor* class and consider it as subclass of *academicStaffMember*.

```
<owl:Class rdf:ID="Professor">
  <rdfs:SubClassOf rdf:resource="#AcademicStaffMember"/>
</owl:Class>
```

If we want to express this example in FOL, we can do it with two unary predicate:

$\forall x : Professor(x) \Rightarrow AcademicStaffMember(x)$

And in the DL we can say:

$Professor \subseteq AcademicStaffMember$

Consider the following example as it combines classes and properties.

Example 2: A course is taught by an academic staff member. We define a *IsTaughtBy* property and limit its domain and range to *Course* and *AcademicStaffMember*, respectively.

```
<owl:ObjectProperty rdf:ID="IsTaughtBy">
  <rdfs:domain rdf:resource="#Course"/>
  <rdfs:range rdf:resource="#AcademicStaffMember"/>
</owl:ObjectProperty>
```

In FOL we can define a binary predicate as follows:

$\forall x,y : IsTaughtBy(x,y) \Rightarrow Course(x)$

$\forall x,y : IsTaughtBy(x,y) \Rightarrow AcademicStaffMember(y)$

Mehran Mohsenzadeh is with the Computer Engineering Department, Islamic Azad University, Sciences &Rese arch Center, Tehran, Iran (e-mail: mmohsenzadeh77@yahoo.com).

Fereidoon Shams is with the Computer Engineering Department, Islamic Azad University, Sciences &Resear ch Center, Tehran, Iran (e-mail: fsaliee@yahoo.com).

Mohammad Teshnehlab is with the Electrical Engineering Department, KNT University, Tehran, Iran (e-mail: Teshnehlab@eetd.kntu.ac.ir).

DL notation of this example is:

$T \subseteq \forall IsTaughtBy.Course$

$T \subseteq \forall IsTaughtBy.\neg AcademicStaffMember$

In this notation, T is the most general class (union of a class and its complement). The Symbol "¯ shows the inverse of a relation.

Like class hierarchies, we can also define property hierarchies. For example, we can define *IsTaughtByProfessor* as a subproperty of *IsTaughtBy* which its range is a professor (example 3)[1].

```
<rdf:Property rdf:ID="IsTaughtByProfessor">
  <rdfs:subPropertyOf rdf:resource="#IsTaughtBy"/>
</rdf:Property>
```

Logically, this can be expressed analogous to example 1 in FOL and DL.

$\forall x,y : IsTaughtByProfessor(x,y) \Rightarrow IsTaughtBy(x,y)$

$IsTaughtByProfessor \subseteq IsTaughtBy$

We can also define a class as conjunction of other classes in OWL.

Example 4: A computer professor is a professor that also is a member of computer department. In OWL, we can write:

```
<owl:Class rdf:ID="ComputerProfessor">
 <rdfs:subClassOf>
  <owl:Class>
   <owl:intersectionOf rdf:parseType="collection">
    <owl:Class
      rdf:about="#ComputerDepartmentMember"/>
    <owl:Class rdf:about="#Professor"/>
   </owl:intersectionOf>
  </owl:Class>
 </rdfs:subClassOf>
</owl:Class>
```

This corresponds to logical conjunction in FOL:

$\forall x:ComputerProfessor(x) \Rightarrow ComputerDepartmentMember(x) \wedge Professor(x)$

And intersection in DL:

$ComputerProfessor \subseteq ComputerDepartmentMember \cap Professor$

In this example if rdfs:subClassOf is omitted, the $\Rightarrow$ symbol must changes to $\Leftrightarrow$, and $\subseteq$ must change to $\equiv$.

Universal qualified quantification can be used for locally restricting the range of a property. In the following example, we define PhD courses by limiting the range of *IsTaughtBy* property.

Example 5: A PhD_Course is only taught by a professor. We define this in OWL as follows:

```
<owl:Class rdf:ID="PhD_Course">
  <rdfs:subClassOf>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="collection">
        <owl:Class rdf:about="#Course"/>
        <owl:Restriction>
          <owl:onProperty rdf:resource="#IsTaughtBy"/>
          <owl:allValuesFrom rdf:resource="#Professor"/>
        </owl:Restriction
      </owl:intersectionOf>
    </owl:Class>
  </rdfs:subClassOf>
</owl:Class>
```

The corresponding notation of this example in FOL is:

$\forall x : PhD\_Course(x) \Rightarrow Course(x) \wedge (\forall y: IsTaughtBy(x,y) \Rightarrow Professor(y))$

We can also represent this in DL as follows:

$PhD\_Course \subseteq Course \cap \forall IsTaughtBy.Professor$

In OWL, we can define symmetric, transitive and inverse properties.

Example 6: Classmate relationship is a symmetric relation. In OWL a symmetric property can be defined using owl:SymmetricProperty.

```
<owl:SymmetricProperty rdf:ID="IsClassmate"/>
```

In Logic (FOL and also DL) we define a symmetric relation as follows:

$\forall x,y : IsClassmate(x,y) \Rightarrow IsClassmate(y,x)$

Example 7: In a hierarchical structure, 'parent' relation is a transitive relation. In OWL, we use owl:TransitiveProperty to define a transitive property.

```
<owl:TransitiveProperty rdf:ID="IsParentOf"/>
```

In FOL, we define a transitive relation in the following way:

$\forall x,y,z : (IsParentOf(x,y) \wedge IsParentOf(y,z)) \Rightarrow IsParentOf(x,z)$

In DL, we use the "+symbol to express transitive property: IsParentOf +

Example 8: The inverse of 'IsTaughtBy' is 'Teaches'. We employ owl:InverseProperty to define inverse properties.

```
<owl:ObjectProperty rdf:ID="IsTaughtBy">
  <owl:inverseOf rdf:resource="#Teaches"/>
</owl:ObjectProperty>
```

The FOL corresponding notation is:

$\forall x,y : IsTaughtBy(x,y) \Rightarrow Teaches(y,x)$

$\forall x,y : Teaches(x,y) \Rightarrow IsTaughtBy(y,x)$

And in DL we use the "¯symbol:

$IsTaughtBy \equiv Teaches^{-}$

Until now, we presented the definition of ontology vocabulary and their relations. Now the definition of instances based on the corresponding vocabulary will be presented.

In the following we instantiate some instances from the *Course* and *Professor* classes.

[1] Since OWL is constructed on top of RDF(S), the rdf:Property can easily be converted to owl:ObjectProperty.

```
<Course rdf:ID="MyCourse"/>
  <IsTaughtBy>
    <Professor rdf:ID="MyProfessor"/>
  </IsTaughtBy>
</Course>
```

To define these instances, we use some facts in FOL:

```
Course("MyCourse").
Professor("MyProfessor").
IsTaughtBy("MyCourse","MyProfessor").
```

In the following section we suggest a mapping from OWL primitives to LP.

III. A Primary Solution to Employ Logic in OWL

According to the above examples, we can suggest a mapping between OWL primitives and LP statements.

- Each concept (class) in ontology can be mapped to a unary relation (predicate) which the relation name is the concept name, and the only argument is the instance of that concept. For example, if 'a' is an instance of class 'c', we write: C(a).

TABLE I
MAPPING BETWEEN OWL AND LP

OWL Statements	LP Statements
C rdfs:subClassOf B	B(X) :- C(X).
(rdfs:union C1,,Cn)rdfs:subClassOf B	B(X) :- C1(X). … B(X) :- Cn(X).
(rdfs:intersectionOf C1,…,Cn) rdfs:subClassOf B	B(X):- C1(X),,Cn(X).
P rdfs:subPropertyOf B	B(X,Y) :- P(X,Y).
P rdfs:domain C	C(X) :- P(X,Y).
P rdfs:range C	C(Y) :- P(X,Y).
C1 owl:sameClassAs C2	C2(X) :- C1(X). C1(X) :- C2(X).
P1 owl:samePropertyAs P2	P2(X) :- P1(X). P1(X) :- P2(X).
owl:SymmetricProperty P	P(Y,X) :- P(X,Y).
owl:TransitiveProperty P	P(X,Z):- P(X,Y),P(Y,Z)
R owl:InverseProperty P	R(Y,X) :- P(X,Y). P(Y,X) :- R(X,Y).
owl:allValuesFrom D is range on property P for class C	D(Y) :- P(X,Y),C(X).
owl:someValuesFrom D is range on property P for class C	In FOL we can say: $\forall X \exists Y: D(Y) \leftarrow P(X,Y) \wedge C(X)$
owl:hasValue V value of property P for class C	C(X) :- P(X,v). P(X,v) :- C(X).

- Each property (relation) can be mapped to a binary relation (predicate). The relation name is the property name. The first argument is the name of the domain instance, and the second one is the name of the range instance. For example, if 'a' and 'b' are the domain and range of property 'p' respectively, we define: P(a,b).

To define unnamed classes and properties, we need to use virtual names. Table 1 shows the corresponding mapping between OWL statements and LP statements.

Unfortunately, writing corresponding LP for owl:someValuesFrom is difficult. In the table we present a FOL corresponding, however, to write a LP corresponding we need to rewrite it based on the situation. We can use two approaches to map this statement: (1) writing all (ground) fact that can be existed. (2) Using an owl:allValuesFrom map style in Table 1 and exclude the possible (ground) fact that can't be existed. For example, if the range of property 'p' can be all instances of class 'c' except the 'a' instance, we can write:

```
D(Y) :- P(X,Y) , C(X) , except(X).
except(a).
```

IV. A Problem

Until now, we managed to find a solution to use logic for OWL, but can this solution satisfy all our expectations for an ontology language. How can we access concepts defined in the ontology? One of the important research topics in ontology is merging and integrating ontologies. In all the current approaches, access to the concepts defined in the ontology (in addition to the instances) is very necessary. When we write: C(a). , we can simply find out if 'a' is an instance of 'c' or not, but how can we see if concept (class) 'c' is defined in the ontology or, how can we access all concepts that are defined in ontology?

V. A Double Implementation

In [3] an approach called DOGMA is introduced for ontology engineering framework. In this approach ontology is divided into two parts. This can lead us to the best general and efficient ontologies. DOGMA approach decomposes ontology into an *ontology base*, which holds (multiple) intuitive conceptualization(s) of a domain, and a layer of *ontological commitments*, where each commitment holds a set of domain rules (see figure 1). More information on DOGMA approach can be found in [4, 5].

We can use this principle (DOGMA called it *the double articulation*) in our solution. If we refer to the previous mapping, we will see that our approach only supports the ontological commitment. We must support ontology base as well. When we write 'C(a).', we express two facts. First, we have a 'c' concept (class). Second, we have an instance of 'c' called 'a'. Now, we try to map OWL to LP, but this time with double implementation.

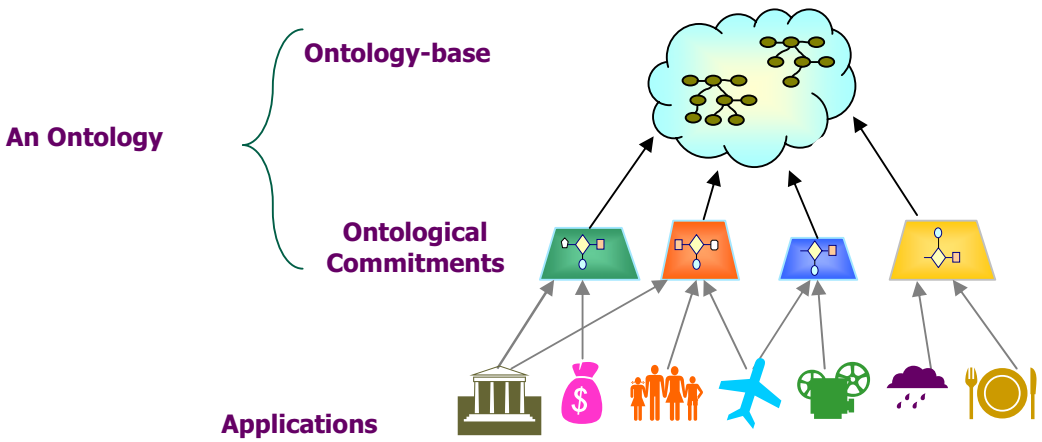

Figure 1: Knowledge organization in DOGMA approach

- Each concept (class) in ontology can be mapped to two unary relations (predicates). First relation is defined as before. The relation name is the concept name, and the only argument is the instance of that concept. Second relation is called Class' in which the only argument is the concept name. For example, if a' is an instance of class c', we write: C(a). and Class(c).
- Each property (relation) can be mapped to a binary relation (predicate) and a unary relation. The binary relation is defined as before. The relation name is the property name. The first argument is the name of the domain instance, and the second one is the name of the range instance. Second relation is called Property' which the only argument is the property name. For example, if a' and b' are the domain and range of property p' respectively, we define: P(a,b). and Property(p).
- Class instances (individuals) also must be represented in ontological layer. Instance i' of class c' must be mapped to a binary relation (predicate) as: isIndividualOf(i,c).

Now let's see the changes that must be done in Table I based on this double implementation.

Subclass: If class C' is subclass of class B', in addition to the rules mentioned in Table 1, the following rule must be added: subClassOf(C,B). We know that this relation is transitive, so we can state:

subClassOf(X,Z):- subClassOf(X,Y) , subClassOf(Y,Z)

Subproperty : If property P' is subproperty of property B', we must add the rule: subPropertyOf(P,B). This relation is also transitive, so we add the following rule:

subPropertyOf(X,Z)-
subPropertyOf(X,Y),subPropertyOf(Y,Z)

With the following rules, there is no need to write Class' and Property' relations for hierarchical classes/properties.

Class(X) :- subClassOf(X,Y).
Class(Y) :- subClassOf(X,Y).
Property(X) :- subPropertyOf(X,Y).
Property(Y) :- subPropertyOf(X,Y).

Class equivalence: following rules must be added to Table 1 for owl:sameClassAs.

sameClassAs(X,Y) :- subClassOf(X,Y), subClassOf(Y,X).
sameClassAs(X,X) :- Class(X).

Property equivalence: analogous to class equivalent, we must add the following rules:

samePropertyAs(X,Y) :-
subPropertyOf(X,Y), subPropertyOf(Y,X).
samePropertyAs(X,X) :- Property(X).

Instance equivalence: before we write the instance equivalence rules, let's see how an instance can be captured. This is done by the following rule.

Individual(I) :- isIndividualOf(I,C) , Class(C).

Now, we can write equivalence rules.

sameIndividualAs(X,Y):-isIndividualOf(X,C1),
isIndividualOf(Y,C2), sameClassAs(C1,C2).
sameIndividualAs(X,X) :- Individual(X).

we can also define unequal instances with notSameAs' predicate. If x' and y' are not equal, we can write: notSameAs(X,Y).

Equivalence: according to the above rules, we can state equivalence as follows:

sameAs(X,Y) :- sameClassAs(X,Y).
sameAs(X,Y) :- samePropertyAs(X,Y).
sameAs(X,Y) :- sameIndividualAs(X,Y).
sameAs(X,Y) :- sameAs(Y,X).
sameAs(X,Z):- sameAs(X,Y) , sameAs(Y,Z)
sameClassAs(X,Y) :- sameAs(X,Y) , Class(X) , Class(Y).
samePropertyAs(X,Y) :-
sameAs(X,Y), Property(X), Property(Y).
sameIndividualAs(X,Y) :-
sameAs(X,Y) , Individual(X) , Individual(Y).

class and property extension: consider c' is a class and p' is a property. we can extend class and property definitions with the following rules:

C(X) :- C(Y) , sameIndividualAs(X,Y).

P(X,Y) :- P(X,Z), sameIndividualAs(Y,Z)
P(X,Y) :- P(Z,Y) , sameIndividualAs(X,Z)

Functional property: before presenting rules for functional property, let's see how inconsistencies of an ontology base layer can be found.

inconsistence(X,Y) :- sameAs(X,Y) , notSameAs(X,Y).
inconsistence(X,Y) :-
sameAs(X,Y), Class(X), Property(Y).
inconsistence(X,Y) :-
sameAs(X,Y), Class(X), Individual(Y).
inconsistence(X,Y) :-
sameAs(X,Y) , Property(X) , Individual(Y).

If 'p' is a functional property, we can write:

sameIndividualAs(X,Y) :- P(D,X) , P(D,Y).

Now if 'x' and 'y' are defined as unequal instances (i.e.: notSameAs('x','y').), an inconsistency has occured. This can be catch by current logic tools in LP environments.

Inverse Functional Property: like functional property, we define the following rule for inverse functional property 'p'.

sameIndividualAs(X,Y) :- P(X,R) , P(Y,R).

Cardinality: OWL Lite (see conclusion section) restricts cardinality to (0,1). Cardinality (0,1) means functional property, hence we can use the same functional property for this cardinality.

The rest of the statements mentioned in Table I do not need any change.

VI. Conclusion

Expressiveness is one of the major goals of OWL [6]. When a language is more expressive, its use and implementation is more difficult. Current research shows that only a limited subset of OWL primitives is used (even by professional users). Therefore, three different sublanguages of OWL has been developed:1-OWL Full: provides support for maximum expressiveness and syntactic freedom of RDF with no computational guarantees. 2- OWL DL: provides support for the maximum expressiveness without losing computational completeness and decidability of the reasoning systems.3- OWL Lite: provides support for classification hierarchy and simple constraint features, such as cardinality values of 0 or 1.

We showed how OWL primitives can be expressed by logical syntax and suggested a mapping between OWL primitives and LP statements. DOGMA approach decomposes ontology into an *ontology base*, which holds (multiple) intuitive conceptualization(s) of a domain, and a layer of *ontological commitments*, where each commitment holds a set of domain rules. Therefore, in our extension, we have supported both of the layers and have presented the mapping between OWL and LP statements in both layers.

In this paper, our idea is applicable for OWL Lite and part of OWL DL. These sublanguages are expressive enough to support most of our requirements. In addition, reasoning is complete (guarantees that an answer will be found for a problem). For future work, one can try to implement more primitives of OWL to support OWL Full.

References

[1] Ushold M., Gruninger M., *Ontologies: Principles, methods and applications*, The Knowledge Engineering Review, 1996.

[2] Antoniou G., Harmelen F., *Web Ontology Language: OWL*, 2003.

[3] Ostadzadeh Sh., Mohsenzadeh M., *Ontology Engineering in comparison to Data Modeling*, Advance topics in database white paper, 2004.

[4] Jarrar M., Meersman R., *Formal Ontology Engineering in the DOGMA Approach*, International conference on Ontologies, Databases and Application, 2002.

[5] Ullman J.D., *Principles of Database and Knowledge-base Systems - volume 1*, Computer Science Press, 1988.

[6] Heflin J., *Web Ontology Language (OWL) Use Cases and Requirements*, W3C Working Draft, 2003.