Empowering OWL with Overriding Inheritance, Conflict Resolution and Non-monotonic Reasoning[*]

Shazzad Hosain and **Hasan Jamil**
Department of Computer Science, Wayne State University, USA
shazzad@wayne.edu, jamil@cs.wayne.edu

Abstract

The popularity of OWL for knowledge representation in the Semantic Web applications makes it an attractive platform. Although OWL supports some form of object-oriented features for knowledge structuring and maintenance, it is significantly weak in capturing most essential object-oriented features such as single and multiple inheritance, default class values, methods, overriding and encapsulation in their true spirits. It is also weak in extending reasoning support for intelligent knowledge processing. Such features are becoming increasingly essential in applications such as social networks, e-commerce and knowledge rich ontology for Life Sciences. In this paper, we propose an extension of OWL toward a more powerful knowledge structuring language, called OWL^{++}, by supporting multiple different types of inheritance with overriding, and non-monotonic reasoning capabilities within OWL. We demonstrate OWL^{++}'s computability and implementability by presenting a translational semantics of OWL^{++} to OWL, for which we have robust execution engines while for the reasoning component of OWL^{++} we rely on Jena to support rules in OWL.

Introduction

Since W3C accepted OWL Web Ontology Language as a standard for knowledge representation in the Semantic Web several years ago, OWL has become the leading ontology language in industry and academia as evidenced by the significant number of applications and development tools reported in the literature. Despite OWL's immense popularity, its inherent limitation in handling exceptions and conflicts in multiple inheritance hierarchies poses significant hurdles in naturally capturing modern application semantics that demand advanced constructs to structure knowledge. This limitation may be attributed to OWL's semantic foundations rooted in Description Logic (DL). From a software engineering and knowledge representation standpoint, the importance of class and inheritance hierarchies has been well documented in the literature and in OWL's own adoption of these concepts to a limited extent. The insistence on inheritance with overriding in class hierarchies also demands attention to the concomitant issue of conflict resolution due to multiple inheritance and exceptions. The absence of a satisfactory semantic along these lines in OWL has been a major bottleneck for its wider use in modern applications requiring serious knowledge representation tools.

Conflicts and exceptions are natural in common sense knowledge. To appreciate the breadth and scope of exceptions in the context of the Semantic Web, we summarize below the various different ways it can manifest itself in such applications. For example,

- *Rules with Exception*: In e-commerce it is common to have conflicting business rules. In (Antoniou and Arief 2002), the authors present several such rules with exceptions. One of the simple ones says that 'If a customer is loyal then grant discounts', which is in contradiction with the general business rule that states 'No discount may be allowed to customers'.
- *Reasoning with Incomplete Information*: (Antoniou and Arief 2002) also describes a scenario where business rules have to cope with incomplete information, i.e. in the absence of certain information some assumptions have to be made that lead to conclusions not supported by OWL. For example, conventional wisdom dictates that the packaging for tax free food is also tax free. However, in the absence of the knowledge that any packaging that is more exotic than usual is not tax free.
- *Ontology Merging*: When ontologies from different sources are merged, contradictions arise naturally and OWL cannot cope with such inconsistencies.
- *Default Inheritance in Ontologies*: The need for default value inheritance in ontologies has been discussed in (Bernstein and Grosof 2003) in the context of modeling the MIT Process Handbook. As pointed out in this example, and numerous others (Jamil 1997), default values and overriding inheritance of such values in inheritance hierarchies are one of the most effective, efficient and elegant ways of factoring information and streamlining their maintenance.
- *Non-monotonic Reasoning*: Once exceptions are allowed, and reasoning is supported, the natural demand is the

[*]Research supported in part by National Science Foundation grants CNS 0521454 and IIS 0612203.

support for non-monotonic reasoning as advocated in RuleML.

To exemplify the need for representing default values and their inheritance in OWL class hierarchies, let us discuss a simple scenario. In OWL, all properties are defined outside of class descriptions and are thus decoupled from all classes making them global in nature, exposed and available to all classes. The mapping constructs domain and range externally tie properties to OWL classes. Since a property is global, default value assignment is not feasible or meaningful for OWL classes. Essentially, values are thus assigned at the instance level. For example, if class GradStudent has a *monthlySalary* property with a default value \$1,500, then in OWL we need to assign this value to every GradStudent instance explicitly through textual assignment. In essence, the inheritance of the default value and its possible overriding is decided statically at compile time, leaving no option for updating the default value. This leaves the issue of maintenance on shaky ground because if we are to change this default value to another value, say \$1,700, we will be required to update the knowledge base entirely and replace all occurrences of \$1,500 to \$1,700. As opposed to OWL, object-oriented (OO) systems handle default values quite elegantly. In OO systems, default class values are assigned inside a class template to an attribute or class variable (Meditskos and Bassiliades 2008) which is local to the class, and thus, shielded from outside use. The mechanism of inheritance is used to share this value at run time at the instance or subclass level, giving the opportunity to update the class value when needed without the need to update all instance objects locally. This dynamic resolution of overriding and default value inheritance facilitates maintenance and improved knowledge structuring as we will witness in the following sections. OO systems also are better equipped to handle conflicts in the case of multiple inheritance (Jamil 1997) but OWL simply does not support this feature.

Contributions of OWL^{++}

In this paper, we present an extension of OWL called OWL^{++} to support default value inheritance, overriding, conflict resolution and non-monotonic reasoning in a spirit similar to OO systems. So, the main contributions of our proposal can be summarized as follows:

- We introduce the notion of default property values in OWL classes and their inheritance with overriding in the class hierarchies.
- We address the issue of multiple inheritance and conflict resolution through the introduction of the concept of property locality and inheritability in OWL class hierarchies.
- We also introduce the concept of *null inheritance*[1] in OWL to model withdrawal of properties in subclasses to aid conflict resolution and to support complex inheritance needs.
- As a demonstration of its implementability and the soundness of its semantics, we present a translation procedure of OWL^{++} to OWL. OWL^{++} is implemented as a Java based front-end through which the users interact on top of Jena (jena.sourceforge.net). We chose to use a translational semantics to leverage existing robust OWL engines such as Jena for our implementation to avoid effort duplication, ease of implementation and efficiency, and for the reasoning support it offers for OWL.

[1]A superclass in OWL^{++} may inhibit a subclass from inheriting its properties by declaring them null, such as *AverageSalary* in employee class, which has no meaning at the instance level.

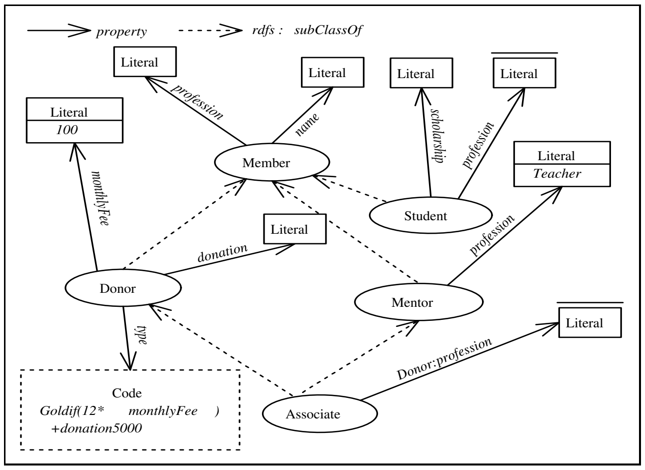

Figure 1: Classes showing inheritance modes and types

A Motivating Example

We now discuss modeling a social network type of application similar to Facebook and LinkedIn using OWL^{++} for expository purposes to highlight its salient features on intuitive grounds. In this example application, we model a non-profit humanitarian group that aims to promote education in the less fortunate parts of the world. Every member of this social network is classified as a Donor, Student, Mentor or Associate. Donors offer financial resources or funds to students for their education. Mentors are professional teachers who act as surrogates to administer the funds donated by donors and guide students, while Associates offer support in the form of coordination and program organization. The associates in this social network are usually members of the Mentor class who are teachers.

Figure 1 shows a partial ontology of this application in which classes are shown in ovals (i.e., Donor), solid arrows represent properties and the annotations on the arrows represent their names (i.e., *monthlyFee*), solid rectangles represent default class values (i.e., *\$100*), and dashed rectangles depict default class values (also known as virtual attributes) that are computed (i.e., *Gold if (monthlyFee*12 + donation)* $\geq$ *\$5,000*). Computed property values are essentially a set of rules that assigns a value to the property when executed. A bar above any rectangle indicates an inhibition of the associated property in the class for which the property is defined (i.e., property *profession* in Student class is withdrawn although it is a subclass of Member). It is important to note here that in OWL^{++}, a property can only be associated with one class giving the property a specific locality that we exploit in overriding and conflict resolution decisions during inheritance in OWL^{++}. Finally, a dashed arrow captures the notion of subclasses in OO systems (i.e., Donor inherits all the properties of Member being one of its subclasses and additionally defines three new properties *monthlyFee, donation*, and *category*). Inhibited property annotations can be further prefixed with class names to indicate the choice of inhibition (i.e., *Donor:profession* in class Associate). This feature is useful for the purpose of inheritance conflict resolution as we will discuss shortly.

Preliminaries

To stay consistent with OWL and one of its extensions called SWRL, we adapt the syntax of these two languages with slight modifications to be able to introduce the primitives of OWL^{++}. We now briefly discuss the syntax and semantics of OWL (Patel-Schneider, Hayes, and Horrocks 2004) and SWRL (Horrocks et al. 2004) as adapted in OWL^{++}.

OWL Web Ontology Language

Definition 1 (OWL Vocabulary) The OWL vocabulary V consists of a set of literals V_L and seven sets of URI references, V_C, V_D, V_I, V_{DP}, V_{IP}, V_{AP}, and V_O. The components of V, i.e., V_C, V_D, V_I, V_{DP}, V_{IP}, V_{AP}, and V_O, are pairwise disjoint. V_C consists of class names *owl:Thing* and *owl:Nothing*. V_D consists of OWL URI reference for built-in datatype name *rdfs:Literal*. V_{AP} is the set of annotations containing *owl:versionInfo*, *rdfs:label*, *rdfs:comment*, *rdfs:seeAlso*, and *rdfs:isDefinedBy*. Finally, the individual-valued property names V_{IP}, the data-valued property names V_{DP}, the individual names V_I, and the ontology names V_O of V do not have any required members.

Definition 2 (OWL Interpretation) An OWL Interpretation is a tuple of the form $I = \langle R, EC, ER, L, S, LV \rangle$, where R is a set of resources, $LV \subseteq R$ is a set of literal values, EC is a mapping of URI references to OWL classes and datatypes. Also, ER is a mapping of URI references for OWL properties, L maps typed literals to LV and S maps individual names to elements of EC (owl:Thing).

Definition 3 (OWL Classes) An OWL class oc_s is a structure of the form $< \alpha_c, \kappa >$, where α_c is the class axiom[2] and κ is a set of OWL constructs[3].

Example 1 The following example captures the Associate class of Figure 1 in OWL.

```
<owl:Class rdf:ID="Associate">
 <rdfs:subClassOf rdf:resource="#Donor" />
 <rdfs:subClassOf rdf:resource="#Mentor" />
</owl:Class>
```

Definition 4 (OWL Instances) An OWL instance oi_s is a structure of the form $< \alpha_i, pv >$, where α_i is the instance axiom and pv is a set of name value pairs.

[2]class axiom declares an URI reference to be the name of an OWL class (Bechhofer et al.)

[3]constructs are statements such as subClassOf, complementOf, unionOf etc. that help in building more complex classes

Example 2 Let D1 be an instance of Donor in which the properties *name* and *profession* have values "Sam" and "Teacher" respectively. The OWL representation for Donor is shown below.

```
<Donor rdf:ID="D1">
 <name rdf:datatype="&xsd;string">Sam</name>
 <profession rdf:datatype="&xsd;string">Engineer
 </profession>
</Donor>
```

The Semantic Web Rule Language (SWRL)

SWRL (Horrocks et al. 2004) extends OWL with simple Horn like rules. The syntax of a rule is $\mathcal{A} \leftarrow \mathcal{B}_1, \ldots, \mathcal{B}_m$, where $\mathcal{A}$ is the head and $\mathcal{B}_i s$ are body atoms. The atoms can be of the following forms: $C(x)$, $D(z)$, $P(x, y)$, $Q(x, z)$, sameAs(x, y), differentFrom (x, y) and builtIn($B, z_1, \ldots, z_n$), where C is an OWL DL description, D is an OWL DL data range, P is an OWL DL *individual-valued* property, Q is an OWL DL *data-valued* property, B is a built in predicate, x, and y are variables or OWL individuals, and $z, z_1, \ldots, z_n$ are variables or OWL data values. Usually the body atoms are binary predicates but using *built-ins* SWRL also allows some predefined *n*-ary predicates in body atoms. We adapt the XML version of rule syntax directly from (Horrocks et al. 2004). However, since the XML version is fairly verbose we will use short hand notation as follows:

$$head \leftarrow body$$

OWL^{++} Overview

We now briefly introduce the syntax and semantics of OWL^{++} with an overview of its salient features. An OWL^{++} program basically is a set of classes, instances, object and data type properties and rule definitions. In contrast to OWL class definitions, an OWL^{++} definition has properties with default value or rules through its signature construct. An instance object is similar to a class object except that instances do not define the structural aspects of class objects and can not have instances or subclasses of their own.

Syntax

Definition 5 (OWL^{++} Programs) An OWL^{++} program **P** is a tuple of the form $\langle \Sigma, \Upsilon, \Omega \rangle$, where Σ is a set of class structures, containing owl:Thing and owl:Nothing, Υ is the set of Horn like rules and $\Omega = \{V_D, V_I, V_{DP}, V_{IP}, V_{AP}, V_O\}$ is the set of OWL elements as defined in Definition 1.

Definition 6 (OWL^{++} Class Structure) An OWL^{++} class structure c_s is of the form $< \alpha_c, \psi, \kappa >$, where α_c is the class axiom, ψ is the signature of the class and κ is a set of OWL constructs.

Definition 7 (Class Signature) A class signature ψ of class c is of the form $\langle \rho, \delta \rangle$, where ρ is a set of inhibited properties from its super classes and δ is the set of properties that have either default values or a rule corresponding to a virtual property. The set of inhibited properties in c is denoted by $\rho(c)$.

Example 3 The following XML code fragment defines the structure of Donor and Associate class in OWL^{++}. Within the signature construct Donor takes a default value on *monthlyFee* and a default rule for *type* property[4].

```
<owl:Class rdf:ID="Donor">
 <rdfs:subClassOf rdf:resource="#Member" />
 <owlp:Signature>
  <owlp:hasDefaultValue>
   <monthlyFee rdf:datatype="&xsd;int">100</monthlyFee>
   type(object, "Gold") <- monthlyFee(object, fee),
    donation(object, amount), 12*fee+amount >= $5,000
  </owlp:hasDefaultValue>
 </owlp:Signature>
</owl:Class>
```

Similarly, the class Associate inhibits the property *profession* from Donor in an attempt to avoid conflict due to multiple inheritance, and thus allows the inheritance of the property along with its default value from the Mentor class, if any. In the absence of this inhibitory declaration, Associate will inherit neither based on OWL^{++}'s cautious semantics.

```
<owl:Class rdf:ID="Associate">
 <rdfs:subClassOf rdf:resource="#Donor" />
 <rdfs:subClassOf rdf:resource="#Mentor" />
 <owlp:Signature>
  <owlp:reject rdf:resource="$Donor::profession"/>
 </owlp:Signature>
</owl:Class>
```

To be able to capture an intuitive semantics of inheritance with overriding and conflict resolution, (Jamil 1997) introduced the concept of property locality, and inheritability of properties based on the locality of their definitions. The idea of inheritability states that a class can inherit a property from another class if there is a unique path of subclass relationships between these classes, and there is no intermediate class that redefines the same property (and thus overrides it), thereby guaranteeing the inheritance of most specific and conflict free class properties. The concept of locality and inheritability are captured in the following set of definitions.

- *Code or virtual property*: A property p in $\Upsilon \subset V_{CP}$ is said to be virtual if it is defined using a OWL^{++} rule.
- *Global property*: Any property $p \in P = \{V_{IP}, V_{DP}, V_{CP}\}$ is said to be global since it is defined outside of a class.
- *Locality of property*: Let p be a property and o be an object then $o : p$ represents the fact that p is locally defined at o. Intuitively, o is called the *context* or the *descriptor* of the local property p, i.e. $context(p) = o$. In example 3 the property *monthlyFee* is defined locally within the signature construct i.e. $context(monthlyFee) = Donor$.
- *Is-a*: Let o_c, o_i and o_s be class names. Then $o_i \in o_c$ and $o_s :: o_c$ are respectively instance and subclass descriptions in OWL^{++}. Intuitively, they say that o_i and o_s are instance and subclass of o_c respectively. We will use the notation $o\sharp p$ throughout when the difference between the two is unimportant.

[4]For lack of space, instead of using the customary XML based SWRL syntax, we directly use a Horn like rule notation for this virtual property *type* within the class structure below with the understanding that in a real OWL^{++} program this rule will be replaced by a SWRL like syntax.

Definition 8 (Locality) Let $\mathbf{P} = \langle \Sigma, \Upsilon, \Omega \rangle$ be a program, o_c be a class object, o be any object, φ be any signature expression for $o_c \in \Sigma$, and $p \in P$ be any property such that $context(p) = o$, or $context(\varphi) = o$. Then p and φ are local to objects o and o_c respectively in $\mathbf{P}$.

Definition 9 (Property Inheritability) Let S be a set of is-a descriptions, p be a property symbol and o be an object. Then the inheritability of the property p in the object o is defined by the *property inheritability function* ∇_p as follows:

$$\nabla_p(S,p,o) = \begin{cases} o_s & \text{if property } context(p) \neq o \text{ and } [\exists_q \text{ such that } o\sharp q \in S,\ \nabla_p(S,p,q) = o_s,\ context(p) = o_s,\ \nabla_p(S,p,o) = \nabla_p(S,p,o_s) \text{ and } (\forall_r, \text{ such that } o\sharp r \in S, \text{ one of hte following holds.} \\ & \bullet\ \nabla_p(S,p,r) = r, \text{ and } context(p) \neq r, \\ & \bullet\ \nabla_p(S,p,r) = o_s, \text{ or } p \in \rho(o).)] \\ o & \text{in all other cases} \end{cases}$$

The algorithm to compute the inheritability function $\nabla_p(S,p,o)$ is given bellow:

Algorithm 1 computeInheritableFunction(*class*, *lcc*)

```
Require: initially lcc = ∅ {lcc = list of computed classes}
  if class ∈ lcc then
    continue
  end if
  ldp = list of default properties of class
  for each dp ∈ ldp do
    put $inheritable(class, class, dp)$. into rule base
  end for
  lrp = list of inhibited properties of class
  lip = list of immediate parents of class
  for each pClass ∈ lip do
    if lcc ≠ ∅ and pClass ∉ lcc then
      computeInheritableFunction (pClass, lcc)
    end if
    lp = list of all properties of pClass
    for each p ∈ lp do
      if p ∈ ldp or p ∈ lrp then
        continue
      end if
      if p already inherited from another class then
        throw Exception ("Inheritance Conflict")
      end if
      add fact $inheritable(class, pClass, p)$. to rule base
    end for
  end for
  add class ∈ lcc
```

Definition 10 (Inherited Property) Let **P** be a program, o_c be a class object, o_i be an instance object, o be any object i.e. $o \in o_c \cup o_i$, $p \in P$ be a property, and $context(p) = o$. Then p is inherited in o if $\nabla_p(S,p,o_c) = o$. We say o *code* inherits p if $p \in V_{CP}$, otherwise it *value* inherits p if $p \in V_{IP} \cup V_{DP}$.

Semantics Based on Rewriting

The semantics of an OWL^{++} program $\mathbf{P} = \langle \mathbf{\Sigma}, \mathbf{\Upsilon}, \mathbf{\Omega} \rangle$ are obtained by translating it to an equivalent OWL program $\mathbf{P}' = \langle \Sigma', \Upsilon', \Omega' \rangle$ such that Σ' is a set of OWL classes in V_C, Υ' is a set of SWRL rules and Ω' has a set of instances $V_{I'} = V_I \cup V_{SI}$ where V_{SI} is a set of system instances[5] for every OWL^{++} class that has default values. The algorithm below describes how to translate $\mathbf{P}$ to $\mathbf{P}'$.

Algorithm 2 Rewriting class structure

```
if signature $\psi \in c_s\langle \alpha_c, \psi, \kappa \rangle = \emptyset$ then
  OWL class expression $oc\langle \alpha_c, \kappa \rangle = c_s\langle \alpha_c, \psi, \kappa \rangle$
  add $oc\langle \alpha_c, \kappa \rangle \in \Sigma'$
else
  remove $\psi$ from $c_s\langle \alpha_c, \psi, \kappa \rangle$
  add $oc\langle \alpha_c, \kappa \rangle$ in $\Sigma'$
  if $\delta \in \psi \neq \emptyset$ then
    if $\delta \in V_{IP} \cup V_{DP}$ then
      create $oi^{\top}$, a system instance of OWL class
      for every $p \in V_{IP} \cup V_{DP}$ do
        add $p \in oi^{\top}$
      end for
      add $oi^{\top} \in V_{SI}$
    end if
    if $\delta \in V_{CP}$ then
      for every $p \in V_{CP}$ do
        replace object variable with the class object
      end for
    end if
  end if
end if
```

Example 4 The Donor class in OWL^{++} shown above will be disassembled to an equivalent OWL class with the same name, a system instance named Sys-Donor with *monthlyFee* of value \$100 and a rule for the *type* property as shown below.

```
<owl:Class rdf:ID="Donor">
 <rdfs:subClassOf rdf:resource="#Member" />
</owl:Class>
<Donor rdf:ID="Sys-Donor">
 <monthlyFee rdf:datatype="&xsd;int">100</monthlyFee>
</Donor>

type(Donor, "Gold") <- monthlyFee(Donor, fee),
 donation(Donor, amount), 12*fee+amount <= $5,000
```

For the class Associate, we just remove the signature and add the following OWL class.

```
<owl:Class rdf:ID="Associate">
 <rdfs:subClassOf rdf:resource="#Donor" />
 <rdfs:subClassOf rdf:resource="#Mentor" />
</owl:Class>
```

[5]System instance is not part of OWL^{++} program individuals and users are not aware of it. It is used to simulate the default behavior of OWL^{++} class.

Table 1: Query Translation

Original Query	Translated Query
(Donor *monthlyFee* ?value)	(Sys-Donor *monthlyFee* ?value)
(Associate *profession* ?value)	(Sys-Mentor *profession* ?value)
(D1 *monthlyFee* ?value)	(Sys-Donor *monthlyFee* ?value)
(D1 *name* ?value)	(D1 *name* ?value)

Recall that we have used signatures to override values and rules locally within a class. To be able to simulate non-monotonic inheritance with overriding and conflict resolution, we create a system instance of a class if that class contains any default value for a property $p \in V_{IP} \cup V_{DP}$ and assign the value to this system instance. However, if $p \in V_{CP}$ then we replace the class object with the object returned by the inheritable function.

The translation into an OWL then completes by adding all the facts computed by ∇ function to $\mathbf{P}'$, a process we call *inheritance closure*, and by adding the following completion rules to it. The completion rules properly deduce which object o can inherit a property p from a class c and bind those inherited properties to the relevant objects. For our example, inheritance closure will add the following facts to the translated OWL program $\mathbf{P}'$.

inheritable (Donor, monthlyFee, Donor).
inheritable (Donor, type, Donor).
inheritable (Mentor, profession, Mentor).
inheritable (Associate, profession, Mentor).
inheritable (Associate, monthlyFee, Donor).
inheritable (Associate, type, Donor).

Query Translation

We query an OWL^{++} program with the SPARQL query language (Prud'hommeaux and Seaborne). The general syntax of SPARQL is of the form (subject predicate object), where subject, predicate and object are either variables[6] or URIs and have usual meanings as in SPARQL.

Any query to a class instance in OWL^{++} does not need to be translated, since the instance structure in both languages are identical. However, a query involving class structures, e.g. default values, needs to be translated to equivalent OWL query. Since the default value of a class property is tied with the object *sys-instance*, a query that asks for a class property value is translated with respect to that *sys-instance*. The *sys-instance* is actually the system instance of the class from which the property is inherited or returned by the inheritable function. For example, the query (Donor *monthlyFee* ?value) is translated to (Sys-Donor *monthlyFee* ?value), since the value of *monthlyFee* is stored in Sys-Donor instance. Several other examples of translated queries are given in Table 1.

Since *inheritance closure* computes all the facts of inheritability, the query translation now becomes a pure deduction and is implemented by the following two rules.

property(source, value) $\leftarrow$ *inheritable(source, prop, object), property (object, value).*

[6]Variables start with the symbol '?'.

property(source, value) ← inheritable(source, prop, object), sysInstanceOf(object, sys_object), property(sys_object, value).

Related Research

To the best of our knowledge, we believe that OWL^{++} is the first language that extends OWL with support for overriding inheritance, conflict resolution and non-monotonic reasoning in a single platform. However, the development proposed in this paper has been largely motivated by the work in (Jamil 1997). We now discuss other research that are similar in spirit to this development, but they do not quite address the issues raised. (Yang, Kifer, and Zhao 2003) propose a model for semantic web, and (Kifer 2005; Katz and Parsia 2005) introduce the concept of non-monotonic inheritance into non OWL based Semantic Web rule languages. On the other hand, (Bernstein and Grosof 2003) proposed a new approach called Courteous inheritance, using Courteous Logic Programs (CLP), a subset of Rule Markup Language, RuleML (Boley, Tabet, and Wagner 2001), to represent non-monotonic inheritance reasoning in process ontologies. In this approach, the PH ontology knowledge e.g. subclass relationship and property values, are represented as CLP rules. The CLP rules have labels (optional) to identify prioritized conflicts using a special predicate "overrides". For example, overrides(l1, l2) means that (any rule labeled) "l1" has strictly higher priority than "l2". Similar kind of non-monotonic extensions of Description Logic have been proposed in (Antoniou 2002). This work shows how non-monotonic rule systems in the form of defeasible reasoning can be built on top of the Description Logic. In both cases, exceptions are not modeled as part of the semantics. Instead it needs to specify every exception rule explicitly, making knowledge representation clumsy and compromising the naturalness of knowledge representation.

Summary

Non-monotonic inheritance and overriding has always been a complex subject to address satisfactorily because it involves the notion of theory or belief revision. In the context of the Semantic Web, the problem is more acute because complete enumeration of all exceptions are not possible ahead of time due to its sheer size, and the constant changes in the universe of discourse. So, there is a serious need for a language in which the semantics of the language is capable of handling such exceptions naturally without user intervention or prior knowledge. In this paper, we have proposed one such language to tackle these issues in an elegant fashion within the semantics of the language. This extension empowers OWL with the requisite set of constructs needed to support modern applications. We have presented a brief but formal syntax for our language OWL^{++} and demonstrated its usefulness and salient features using several real life examples. To comply with standards and recent developments, we have adapted to OWL and SWRL syntax, and implemented our system on top of the Jena reasoner for OWL. In our implementation, the user always uses OWL^{++} to represent an ontology, and is unaware of the use of the back-end systems. Thus, the main strength of this proposal is its tight integration with the existing OWL and SWRL languages and the rewriting based approach that we take to construct a semantics of OWL^{++}, allowing us to use existing technologies to build the OWL^{++} reasoner.

Acknowledgement: The authors would like to acknowledge the helpful discussions they have had with Artem Chebotko during the development of this article.

References

Antoniou, G., and Arief, M. 2002. Executable declarative business rules and their use in electronic commerce. In *Proceedings of ACM Symposium on Applied Computing*.

Antoniou, G. 2002. Nonmonotonic rule systems on top of ontology layers. In *ISWC '02: Proceedings of the First International Semantic Web Conference on The Semantic Web*, 394–398. London, UK: Springer-Verlag.

Bechhofer, S.; van Harmelen, F.; Hendler, J.; Horrocks, I.; McGuinness, D. L.; Patel-Schneider, P. F.; and Stein, L. A. *OWL Web Ontology Language Reference, Recommendation February 2004*. http://www.w3.org/TR/owl-ref/.

Bernstein, A., and Grosof, B. N. 2003. Beyond monotonic inheritance: Towards semantic web process ontologies. Working paper, University of Zurich, Department of Informatics, Winterthurerstrasse 190, 8057 Zurich, Switzerland.

Boley, H.; Tabet, S.; and Wagner, G. 2001. Design rationale of ruleml: A markup language for semantic web rules. In *International Semantic Web Working Symposium*.

Horrocks, I.; Patel-Schneider, P. F.; Boley, H.; Tabet, S.; Grosof, B.; and Dean, M. 2004. *SWRL: A Semantic Web Rule Language Combining OWL and RuleML*. http://www.w3.org/Submission/SWRL/.

Jamil, H. M. 1997. Implementing abstract objects with inheritance in datalog^{neg}. In *Proceedings of the 23rd International Conference on Very Large Data Bases*.

Katz, Y., and Parsia, B. 2005. Towards a nonmonotonic extension to owl. In *International Workshop on OWL Experiences and Directions*.

Kifer, M. 2005. Nonmonotonic reasoning in flora-2*. In *LPNMR '05*, 1–12.

Meditskos, G., and Bassiliades, N. 2008. A rule-based object-oriented owl reasoner. *IEEE Trans. on Knowl. and Data Eng.* 20(3):397–410.

Patel-Schneider, P. F.; Hayes, P.; and Horrocks, I. 2004. *OWL Web Ontology Language Semantics and Abstract Syntax*. http://www.w3.org/TR/owl-semantics/ direct.html.

Prud'hommeaux, E., and Seaborne, A. *SPARQL Query Language for RDF*. http://www.w3.org/TR/rdf-sparql-query/.

Yang, G.; Kifer, M.; and Zhao, C. 2003. Flora-2: A rule-based knowledge representation and inference infrastructure for the semantic web. In *2nd International Conference on Ontologies, Databases and Applications of Semantics*.