OWL that can Choose to Inherit and Hide it Too*

Shazzad Hosain
Department of Computer Science
Wayne State University, USA
shazzad@wayne.edu

Hasan Jamil
Department of Computer Science
Wayne State University, USA
jamil@cs.wayne.edu

Abstract

Although W3C (the World Wide Web Consortium) has chosen OWL Web Ontology Language as the standard for semantic knowledge representation language, OWL has inherent limitations in expressing many Object Oriented (OO) features such as default inheritance, conflict resolution in multiple inheritance, method inheritance and encapsulation. But these features are essential to model real world phenomena, a fact that is underscored by the presence of many object oriented programming languages. OWL chooses not to support these features possibly because of its two design decisions (i) decidability and (ii) low computational complexity. Usually more expressive language tends to be computationally expensive, and in some cases, the language becomes either semi-decidable or undecidable. In this paper, we extend OWL to OWL++ by supporting two inheritance modes, overriding and inflating, and three inheritance types, value, code and null. Our goal is to increase the expressive power of OWL as well as to maintain it as a computationally efficient decidable language. We demonstrate this by taking translational semantics of OWL++ to OWL. It allows us to build our OWL++ reasoner by using existing technologies such as Jena that works on top of well known OWL reasoners such as Pellet.

1 Introduction

The vision of the Semantic Web was to enable a machine to understand and process the underlying data for human consumption. As a first step, it requires a suitable knowledge representation language, often known as ontology language, to model the world for necessary reasoning. Many such languages emerged out of this necessity but W3C accepted OWL Web Ontology Language as the standard. Since then, a number of ontology supporting tools, demos or portals, reasoners have been built in addition to OWL. Despite OWL’s immense popularity it has inherent limitations in representing many Object Oriented (OO) features such as default inheritance, conflict resolution in multiple inheritance, method inheritance, encapsulation etc. that poses serious challenge representing many real world phenomenons. For example, birds fly, penguin is a bird but it can not fly. OWL cannot represent such exceptions. In fact, it cannot represent any kind of exceptions that require theory revisions, which limits OWL’s expressive power as a suitable ontology language. Usually increased expressiveness comes with the compromise of computational complexity and the decidability of the language. Since OWL chooses to be decidable and computationally efficient, it takes description logic (DL) as its logical foundation and from the very beginning, avoids exceptions or non-monotonic reasoning. For the same reason, the Semantic Web Rule Language (SWRL) [6] extends OWL with Horn rules, the decidable fragment of first order rules, to maintain the strengths of OWL.

As an ontology language, OWL is very different from programming languages such as C++, Java etc., most of which support default inheritance and other OO features. Thus, it creates impedance mismatch while developing modern applications that use OWL ontology to reason about application domain. Recently, many automatic techniques such as [9, 10], also known as object-ontology mapping, are being studied to overcome this mismatch.

1.1 Goals and Contributions

In this research our goal is to extend OWL to a new language called OWL++ so that it can handle exceptions, non-monotonic reasoning, method inheritance and encapsulation. Clearly, the goal is to enhance OWL’s expressivity as well as to reduce the impedance mismatch for efficient semantic application development. In this paper, we introduce two modes of inheritance called overriding and inflating; as well as three types of inheritance known as value, code and null (described in the next section) in OWL, in a spirit similar to OO systems. We also introduce polymor-

---

*Research supported in part by National Science Foundation grants CNS 0521454 and IIS 0612203.
phism that provide two levels of access, private and public, to OWL properties. Finally, we have implemented a Java based front-end through which one can build an OWL++ program and reason over the program by interacting its underlying reasoner. We choose to use translational semantics of OWL++. Thus we translate every OWL++ program to an equivalent OWL program and a set of Horn rules. It allows us to build OWL++ reasoner using the existing rule engine framework Jena [1] and OWL reasoner Pellet [2].

2 Motivating Example

In a hypothetical scenario from a CS department, a graduate student’s income equals the stipend he gets. Similarly a faculty’s income is only his salary and he/she must have a Masters (ms) degree. An assistant professor, who is also a faculty, has income from both his salary and consultancy. The assistant professor must have a Ph. D. (phd) degree. The graduate teaching assistant (GTA), who is considered a graduate student and faculty, has income from her stipend as well as TA-ship. Though the GTA inherits both graduate student and faculty it does not have salary property.

Figure 1 shows the ontology of our example in which classes are shown in ovals and properties are shown along the lines. The solid rectangles show default values for literal properties\(^1\) and the dashed rectangles show default rules for message properties, also known as methods. Rectangles with double bars on two sides represent private properties. For example, Faculty class has a private property salary with default value 60k and a public method income that takes the rule \(r_1\) from the following table. Finally, properties that are not inheritable by any other subclasses or instances have a line under the rectangle, e.g. total_faculty in Faculty, and the properties that are inhibited by a subclass so that the properties cannot propagate to its lower classes are shown with rectangles with a line on top of it, e.g. salary in GTA.

![Figure 1. Classes showing inheritance modes and types](image)

Now let \(joe, kelly\) be instances of GradStd \((r_8)\), \(joe\) has stipend 15k \((r_9)\), \(john, max\) are Faculty \((r_{10})\), \(max\) has a salary 75k \((r_{11})\). Sally, sue are GTA \((r_{12})\), sally has stipend 20k \((r_{13})\). Based on the database, let us ask the following queries and discuss some of its modeling features in the following sections.

<table>
<thead>
<tr>
<th>Query</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(q_1)</td>
<td>(?sally \llwineq income(x))</td>
</tr>
<tr>
<td>(q_2)</td>
<td>(?sue \llwineq income(x))</td>
</tr>
<tr>
<td>(q_3)</td>
<td>(?joe \llwineq stipend(x))</td>
</tr>
<tr>
<td>(q_4)</td>
<td>(?joe \llwineq income(x))</td>
</tr>
<tr>
<td>(q_5)</td>
<td>(?joe \llwineq avg_income(x))</td>
</tr>
<tr>
<td>(q_6)</td>
<td>(?joe \llwineq mean_dev(x))</td>
</tr>
<tr>
<td>(q_7)</td>
<td>(?joe \llwineq total_faculty(x))</td>
</tr>
<tr>
<td>(q_8)</td>
<td>(?GTA \llwineq total_faculty(x))</td>
</tr>
</tbody>
</table>

2.1. Inheritance

In the above example, we consider two modes of inheritance, overriding and inflating, and three types of inheritance value, code and null [8]. While inheriting behaviors along the class hierarchies, a significant distinction can be identified, resulting in two very different meanings of behavioral inheritance: static inheritance and dynamic inheritance. A property which is statically defined can only be statically inherited, i.e. only the value can be inherited. For example AsstProf class inherits the salary of Faculty with the value 60k. This form of inheritance is known as static inheritance. OWL already supports static inheritance and in our previous work [7], we have extended OWL with default values or overriding inheritance and conflicts in multiple inheritance in the context of static inheritance. Thus, in this paper our purpose is to introduce the above mentioned inheritance modes and types specifically for dynamic inheritance, described bellow in light of our previous work [7].

The AsstProf class overrides the income property of Faculty by defining a new rule \((r_6)\). We call it overriding

\(^1\)Literal properties take primitive data types such as int, string etc.
mode of inheritance. Now consider the attribute degrees in Faculty and AsstProf. The Faculty class has value ms and AsstProf has phd. We naturally expect that AsstProf’s degrees property should retain both phd and ms. We call it inflating mode of inheritance.

In dynamic inheritance, a method can be inherited in two different ways. First, the code computes the value at the class where the property is defined, and the “computed value” is inherited similar to the spirit of static inheritance. We call this value inheritance. For example, when subclasses of GradStd call avg_income property, the value is computed first and the subclasses inherit this value. Second, the subclasses or instances inherit the code as a whole and compute the value. For example, the instances of Faculty inherit the code and compute the value of income, because different instances have different values for salary. We call it code inheritance.

There is yet another type of inheritance, called null inheritance, which can occur in two different ways. First, the class itself declares the property as non-inheritable property, meaning that, no subclasses or instances could inherit the property. For example, the total_faculty property is not inheritable by any instances or subclasses of Faculty. Secondly, a subclass may not need to inherit a property from its super class and thus it could reject the property. For example, GTA rejects the salary property of Faculty class.

2.1.1 Inheritance Conflict

For multiple inheritance, when a property belongs to more than one parent classes, conflict occurs. For example, if both the Faculty and GradStd classes had name property, conflict would occur. In this case GTA would reject name property from one of the parent classes, probably from Faculty class, to avoid the conflict.

2.2 Encapsulation

Encapsulation provides a way to contain and hide data and codes within the class and then give different levels of access to its members. In the above model, we provide two levels of access, private and public. Both private and public properties are accessible to the class itself as well as to its subclasses and instances. The only difference between public and private is that, one can query only public properties but not private properties. For example, the query q1 will not be satisfied since stipend is a private member of GradStd. However, q1 will be satisfied because sally being an instance of GradStd can access its private property salary to compute the value of income.

3. Preliminaries

In this section, we will give an overview on OWL [11, 13] and its rule extension SWRL [6]. Since we are extending OWL with rules we intend to keep the syntax from OWL and SWRL for OWL++ to ensure maximum backward compatibility.

3.1. OWL Web Ontology Language

OWL ontology mainly defines classes, properties, instances of classes, and relationships between instances and properties. The core features of OWL are described below with appropriate examples.

3.1.1 Namespaces

A typical OWL ontology begins with a namespace declaration similar to the following. The first namespace declaration says that, elements prefixed with owl should be understood as referring to things drawn from the namespace "http://www.w3.org/.../owl#". The next three make similar statements about the RDF (rdf), RDF Schema (rdfs) and XML Schema Datatypes (xsd) namespaces.

```xml
<owl:Class rdf:ID="GTA" />
<owl:Class rdf:ID="Faculty" />
<owl:Class rdf:ID="GradStd" />
```

3.1.2 OWL Classes

A class definition has two parts: a name introduction or reference, and a list of restrictions. By default every class in OWL is a subclass of owl:Thing unless otherwise specified. For example, the following Faculty and GTA classes are subclass of owl:Thing. However, in addition to that, GTA is also a subclass of Faculty and GradStd. The standard XML serialization of classes is shown bellow:

```xml
<owl:Class rdf:ID="Faculty" />
<owl:Class rdf:ID="GTA" />
<owl:Class rdf:ID="GradStd" />
```

Since we extend OWL to OWL++, we define some of the key concepts of OWL class structure so that we can draw the differences between OWL vs OWL++ and show the rewriting process of OWL++ to OWL.

Definition 1 (OWL Class Structure) An OWL class structure $\alpha_C$ is of the form $\langle \alpha_C, \kappa \rangle$, where $\alpha_C$ is the class axiom\(^2\) and $\kappa$ is a set of OWL constructs\(^3\).

\(^2\)An URI reference to be the name of an OWL class [13]
\(^3\)constructs are statements such as subclassOf, complementOf, unionOf etc. that help in building more complex classes.
Example 1 In the above example of GTA class, \( \alpha_c \) represents the class name GTA and \( \kappa \) represents the subClassOf definitions. For Faculty class, \( \kappa = \emptyset \).

3.3.3 OWL Properties

OWL has two types of properties: (i) Datatype property that relates instances of classes to RDF literals such as XSDs, (ii) Object property that relates instances of two classes. An example of a datatype property is given below:

\[
\text{<owl:DatatypeProperty rdf:ID="salary">}
\text{<rdfs:domain rdf:resource="#Faculty"/>}
\text{<rdfs:range rdf:resource="&xsd;int"/>}
\text{</owl:DatatypeProperty>}
\]

3.3.4 OWL Individuals or Instances

Individuals or instances are the members of a class. The following examples show john and max are instances of Faculty. While john does not have any property value, max has salary value of 75k.

\[
\text{<Faculty rdf:ID="john"/>}
\text{<salary rdf:datatype="&xsd;int">75000</salary>}
\]

Definition 2 (OWL Instance Structure) An OWL instance structure \( \alpha_i \) is of the form \( <\alpha_i, pw> \), where \( \alpha_i \) is the instance axiom and \( pw \) is a set of name value pairs, possible empty set.

Example 2 In the example of max, \( \alpha_i \) represents the instance name max and \( pv \) represents the salary value of 75k.

3.3.5 OWL Syntax and Semantics

Definition 3 (OWL Vocabulary) An 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 \( V_O \). In any vocabulary, \( V_C \) and \( V_D \) are disjoint and \( V_{DP}, V_{IP}, V_{AP}, \) and \( V_{OP} \) are pairwise disjoint. \( V_C \) is the class names of a vocabulary, contains \( \text{owl:Thing} \) and \( \text{owl:Nothing} \). \( V_D \), the datatype names of a vocabulary, contains URI references for the built-in OWL datatypes and \( rdfs:\text{Literal} \). \( V_{AP} \), the annotation property names of a vocabulary, contains \( \text{owl:versionInfo}, \text{rdfs:label}, \text{rdfs:comment}, \text{rdfs:seeAlso}, \) and \( \text{rdfs:isDefinedBy} \). \( V_{IP} \), the individual-valued property names of a vocabulary, \( V_{DP} \), the data-valued property names of a vocabulary, and \( V_I \) is the individual names of a vocabulary. \( V_O \), the ontology names of a vocabulary, do not have any required members.

Definition 4 (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 mapping of URI references to OWL classes and datatypes. \( ER \) is a mapping of URI references for OWL properties. \( L \) maps typed literals to \( LV \) and \( S \) maps individual names to elements of \( EC \) (\( \text{owl:Thing} \)).

3.2. Semantic Web Rule Language (SWRL)

SWRL [6] extends OWL with simple Horn like rules of the form \( A \leftarrow B_1, \ldots, B_m \), where \( A \) is the head and \( B_i \)'s are body atoms. For detail syntax and semantics, we refer to [6]. In OWL++, we adapt the XML version of rule syntax directly from [6]. However, since the XML version is fairly verbose, we will use short hand notation of rules such as \( \text{head} \leftarrow \text{body} \) throughout the rest of the paper.

4. OWL++ Overview

We now present the syntax and semantics of the OWL++ language with an overview of its salient features with XML serialization. Similar to OWL, OWL++ also defines a set of class, instance and properties. OWL++ also supports rule similar to the way of SWRL. We define these key concepts in the following subsections.

4.1. OWL++ Classes

In contrast to OWL class definition, an OWL++ definition contains the structural aspects of a class such as different modes and types of inheritance information. We use a special construct \text{Signature} to incorporate those structural information into the class definition as shown in the following examples.

\[
\text{<owlp:Class rdf:ID="Facuty">}
\text{<owlp:Signature>}
\text{<owlp:private rdf:resource="#salary"/>}
\text{<owlp:public rdf:resource="#income"/>}
\text{<owlp:hasDefaultValue>}
\text{<salary rdf:datatype="&xsd;int">60000</salary>}
\text{<degrees rdf:datatype="&xsd;string">ms</degrees>}
\text{<income owlp:inheritanceType="code">}
\text{income(Faculty, val) <- salary(Faculty, val)}
\text{</income>}
\text{<total_faculty owlp:inheritanceType="null">}
\text{total_faculty(Faculty, val) <- iCount(Faculty, val)}
\text{</total_faculty>}
\text{</owlp:Signature>}
\text{</owlp:Class>}
\text{<owlp:Class rdf:ID="GradStd">}
\text{<owlp:Signature>}
\text{<owlp:private rdf:resource="#stipend"/>}
\text{<owlp:public rdf:resource="#income"/>}
\text{<owlp:hasDefaultValue>}
\text{<stipend rdf:datatype="&xsd;int">12000</stipend>}
\text{<income owlp:inheritanceType="code">}
\]
4.2 OWL++ Properties and Instances

We keep the same structure and syntax of OWL properties and instances for OWL++. Thus we omit these features here.

4.3 Syntax

We now formally introduce the syntax of OWL++. An OWL++ program \( P \) is a tuple of the form \( \langle \Sigma, \Omega, \Upsilon \rangle \), where \( \Sigma = \{V_C\} \) is a set of class structures, containing \texttt{owl:Thing} and \texttt{owl:Nothing}, \( \Omega = \{V_D, V_I, V_{DP}, V_{IP}, V_{AP}, V_O\} \) is the set of OWL elements as defined in Definition 3 and \( \Upsilon \) is the set of Horn like rules.

It is important to notice that there is one basic difference between OWL and OWL++. In OWL, properties are defined outside the class structure and we call these properties global property. But, in OWL++, though we define properties globally similar to OWL, we define their visibility, default values and other OO features inside the OWL++ classes locally. We exploit this important piece of locality information to capture an intuitive semantics of different kinds and types of inheritance. The idea of locality and inheritability was first introduced in [8]. We borrow those ideas to give translational semantics of OWL++ into OWL.

To better understand the concepts and the rewriting process, let us give the following definitions.

- **Context of property**: Let \( p \) be a property and \( o \) be an object, then \( o \gg p : s \) represents the fact that \( p \) is defined in the context of \( o \) for some structural information \( s \), where \( s \) could be visibility \((v)\), default values \((d)\) or rejection \((r)\) of properties. In other words, \( o \) is called the context or the descriptor of the property \( p \) with respect to \( s \), i.e. \( \text{context}(p : s) = o \). For example, the Faculty class defines the default value of salary property and thus \( \text{context}(\text{salary} : d) = \text{Faculty} \).

- **Is-a**: Let \( o_c, o_i \) and \( o_s \) be class names. Then \( o_i \in o_c \) and \( o_s \gg o_i \) 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.

4.3.1 Inheritability of Properties

Let \( p \) be a property, \( t \) be its inheritability type, where \( t \) is either \((v)\)value or \((c)\)ode, \( o \) be any instance object \((o_i)\) or class object \((o_c)\), then \[v(o, p, o_s, t)\] states that \( o \) inherits \( p \) of type \( t \) from another object \( o_s \). If \( o_a = o \) then we call it self inheritance. For example,
gets degrees from OWL and program P query time. Let we use the visibility function to encapsulate properties at every OWL V.

shows the rewriting of OWL and program P++ for OWL. Now, OWL already supports \( \ell[0, p, o, t] \), meaning that, given instance object \( (o_i) \) with property values in an OWL program, OWL reasoner returns the property values. So we have to determine the inheritability in two cases (i) \( \ell[0, p, o, t] \), where \( 0 \neq o_a \) and (ii) \( \ell[0, p, o, t] \), and probably \( o_c = o_a \). The first one determines inheritability for \( p \) of \( o_i \), which is an instance of \( o_a \). This inheritability can easily be deduced as shown in section 4.4. However, for the second case we determine the inheritability by observing the fact 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, except inverting mode), thereby guaranteeing the inheritance of most specific and conflict free class properties”. Below we define the property inheritability for the second case:

**Definition 7 (Property Inheritability)** Let \( p \) be a description, \( p \) be a property and \( t \) be its inheritability type and \( o_i \) be a class object. Then the property inheritability function \( \nabla_p(S, p, o_i) \) returns another object \( o_a \) according to the following rules. If \( \text{context}(p : d) = o_c \), then \( o_a = o_c \). In other cases, where \( \text{context}(p : d) \neq o_c ; o_a = o \) if the following happens. If \( \exists q \) such that \( o_c :: q \in S, \nabla(S, p, q) = o, \text{context}(p : d) = o \) and \( \forall_p \) such that one of the following holds: (i) \( \nabla(S, p, r) = r \) and \( \text{context}(p : d) \neq r \) or (i) \( \nabla(S, p, r) \neq r, o \) or \( p \in o(a) \).

### 4.3.2 Visibility of Properties

We use the visibility function to encapsulate properties at query time. Let \( \Lambda(o, p) \) be the visibility function, then \( \Lambda(o, p) \) returns true if \( p \) is public otherwise it returns false.

### 4.4 Semantics Based on Rewriting

We give the semantics of OWL++ by translating every OWL++ program \( P = (\Sigma, \Omega, \textbf{Y}) \) to an equivalent OWL program \( P' = (\Sigma', \Omega', \textbf{Y}') \) and a set of rules \( \textbf{Y}' \), such that \( \Sigma' = \{VC\} \), \( \Omega' = \{DV, VI, VDP, VIP, VAP, VO\} \), where \( V' = V_d \cup V_s \) and \( V_s \) is a set of system instances\(^4\) for every OWL++ class that has default values. Algorithm 1 shows the rewriting of OWL++ program.

The translation process in algorithm 1 has two main steps. First, it computes inheritability function for all

\( [\text{max, salary, max, value}] \) states that max, which is an instance of Faculty, gets its salary from max itself by value inheritance. Similarly, \( [\text{max, degrees, Faculty, value}] \) and \( [\text{AsstProf, income, Faculty, code}] \) state that max gets degrees form Faculty by value inheritance and AsstProf gets income form Faculty by code inheritance.

Algorithm 1 Rewriting OWL++ program

1. \( \text{lst} = \text{new empty list} \)
2. \( \text{for each class } c \in \Sigma \) do
3. \( \text{computeInheritableFunction}(c, \text{lst}) \)
4. \( \text{end for} \)
5. \( \text{add instance inheritability rules in } \textbf{Y}' \)
6. \( \text{for each } c_o \in \Sigma \) do
7. \( \text{rewriteClassStructure}(c_o) \)
8. \( \text{end for} \)
9. \( \text{add query translation rules in } \textbf{Y}' \)
10. \( \text{copy } \{DV, VI, VDP, VIP, VAP, VO\} \) to \( P' \)

classes (line 2-4) and then rewrites OWL++ classes (line 6-8). Algorithm 2 computes the inheritability function and puts the inheritability facts in \( \textbf{Y}' \). For example, the inheritability function will return the following facts for AsstProf.

\[
\text{inheritable(AsstProf, consulting, AsstProf, value).}
\]

\[
\text{inheritable(AsstProf, income, AsstProf, code).}
\]

\[
\text{inheritable(AsstProf, degrees, AsstProf, value).}
\]

\[
\text{inheritable(AsstProf, degrees, Faculty, value).}
\]

\[
\text{inheritable(AsstProf, salary, Faculty, value).}
\]

Once line 3 of algorithm 1 enumerates all \( \ell[0, p, o, t] \), line 5 adds the following two rules in \( \textbf{Y}' \) to enumerate all possible \( \ell[0, p, o, t] \). Line 6-8 rewrites all class structures (details in section 4.4.1). Finally, line 9 adds copy translation rules (section 4.5) and line 10 copies all other properties and instances to OWL program.

\[
\text{inheritable(source, prop, object, value) } \rightarrow \text{instanceOf (source, class)}\text{, inheritable(class, prop, object, value)}
\]

\[
\text{inheritable(source, prop, object, code) } \rightarrow \text{instanceOf (source, class)}\text{, inheritable(class, prop, object, code)}
\]

### 4.4.1 Rewriting Class Structure

Algorithm 3 rewrites a class structure. In line 1, first it changes the namespace of a class from owl to owl. The algorithm then checks if an OWL++ class structure \( c_o \) has a Signature construct \( \psi \). If \( \psi \) is empty i.e. \( oc = c_o \), then it adds the \( oc \) into the OWL program \( \Sigma' \), at line 4. But, if the \( \psi \) is not empty then the algorithm removes \( \psi \) from \( c_o \) to get \( oc \) and add the \( oc \) in \( \Sigma' \) (line 6 and 7). Now, for every signature element \( \eta, \rho \) and \( \delta \) we do the following. For visibility construct \( \eta \), we apply the visibility function \( \Lambda \) (line 8-12), and add the visibility facts in \( \textbf{Y}' \) for which the function returns true. In case of Faculty class, \( \Lambda \) will add the following facts in \( \textbf{Y}' \).

\[
\text{visible(Faculty, income).}
\]

\[
\text{visible(Faculty, degrees).}
\]

\[
\text{visible(Faculty, total_faculty).}
\]

Once the visibility facts are computed, for all classes the algorithm adds (at line 13) the following rule in \( \textbf{Y}' \) to compute the visibility of instance properties.
Algorithm 2 computeInheritableFunction(class, lcc)

Require: initially lcc = ∅
1: if class ∈ lcc then
2: continue
3: end if
4: dp = list of default properties of class from ρ(class)
5: for each dp ∈ dp do
6: if dp has type t ∈ {value, code} then
7: put inheritable(class, dp, class, t). ∈ Υ′
8: end if
9: end for
10: ldp = list of reject properties of class
11: for each pClass ∈ ldp do
12: if lcc ≠ ∅ and pClass ∉ lcc then
13: computeInheritableFunction(pClass, lcc)
14: end if
15: end for
16: lpr = list of all properties of pClass
17: for each p ∈ lpr do
18: if p ∈ ldp or p ∈ lrp or p ∉ lnp then
19: continue
20: end if
21: if p already inherited and p ∈ inflating mode then
22: throw Exception ("Inheritance Conflict")
23: end if
24: end if
25: if p has type t ∈ {value, code} then
26: put inheritable(class, p, class, t). ∈ Υ′
27: end if
28: end for
29: end for
30: add class ∈ lcc

Algorithm 3 rewriteClassStructure (c)
1: change the namespace of owlp to owl
2: if signature ψ ∈ c (αc, ψ, κ) = ∅ then
3: OWL class structure oαc(αc, κ) = cαc(αc, ψ, κ)
4: add oc(αc, κ) ∈ Σ′
5: else
6: remove ψ from cαc(αc, ψ, κ) to get ocαc(αc, κ)
7: add ocαc(αc, κ) ∈ Σ′
8: for every p ∈ ρ(cαc) do
9: if p is public then
10: add visible(αc, γ)
11: end if
12: end for
13: add rule for instance visibility
14: if ∃p ∈ δ(cαc) such that p ∈ V ′ P ∪ V ′ D then
15: create oα′, a system instance of OWL class
16: end if
17: for every p ∈ δ(cαc) do
18: if p ∈ V ′ P ∪ V ′ D then
19: add p ∈ oα′
20: else if p ∈ Cl ∈ V ′ P, Cl ⊆ H ← B and t = code then
21: add Cl′ ← H[[o/V ] ← B[o/V ], [p/V, o, value]] ∈ Υ′
22: else if p ∈ Cl ∈ V ′ C, Cl ⊆ H ← B and t = value then
23: add Cl ∈ Υ′
24: add Cl′ ← H[[o/V ] ← B, [p/V, o, value]] ∈ Υ′
25: end if
26: end for
27: if οα′ = null then
28: add οα′
29: end if
30: end if

visible(ins, prop) ← instanceOf(ins, obj), visible(obj, prop).

Now, if there is one or more static default properties defined in δ, then we create a system instance for the class and assign the property values into the class (line 14-16). For example, GradStd class has default value for stipend that is static. So, we create a system instance of GradStd, called Sys-GradStd, and assign 12k to stipend as shown below.

On the other hand, dynamic properties are defined via rules, which can be inherited value or code and thus, handled separately. (i) Let, Cl ⊆ H ← B ∈ Υ be code inheritable clause. Then for each such clause Cl, we replace it by Cl′ ← H[[o/V ] ← B[o/V ], [p/V, o, code]] ∈ Υ′, such that context of Cl is o and V is a distinct variable not occurring in Cl. For example, the rule for income property of GradStd, income(GradStd, ?val) ← stipend(GradStd, ?val) would be replaced with the rule income(?V, ?val) ← stipend(?V, ?val), inheritable(?V, income, GradStd, code)

(ii) Let, Cl ⊆ H ← B ∈ Υ be a value inheritable rule; then we add Cl′ ∈ Υ′ as well as add Cl′ ← H[[o/V ] ← B[o/V ], [p/V, o, value]] ∈ Υ′. For example, for avgIncome we add the avg_income(?V, ?val) ← avgIncome(GradStd, ?val), inheritable(?V, avg_income, GradStd, value), where avgIncome is a user defined functor [6] that computes the average income of Faculty.

We name this translation process i-completion.

Finally for null inheritance, we add the Cl ∈ Υ′. Since the inheritable function does not compute any facts, the rule is only fired by the class query.

4.5 Query Translation

We query an OWL++ program with the SPARQL query language [12]. The general syntax of SPARQL is of the form (subject predicate object), where subject, predicate and object are either variables, which start with the symbol '?', or URLs, and have usual meanings as in SPARQL. However, since we have used first order (FOL) syntax throughout the paper, we continue using FOL syntax instead of SPARQL for ease of reading.

An OWL++ query is translated in two steps. In first step, we take care of encapsulation by allowing the queries that are only public i.e. have visibility facts in the knowledge base. Thus, every query
of the form \( \text{predicate}(\text{subject}, \text{object}) \) is translated to \( \text{predicate}(\text{subject}, \text{object}), \text{visible}(\text{subject}, \text{predicate}) \). In second step we now translate the query to simulate different kinds of inheritance. Dynamic value and code inheritance have already been handled through i-completion and rewriting of necessary rules. But queries regarding static default values are translated in the context of sys-instances. For example, the query \( \text{degrees}(\text{Faculty}, \ ?\text{val}) \) is translated to \( \text{degrees}(\text{Sys}\text{-Faculty}, \ ?\text{val}) \), since the value of \( \text{degrees} \) is stored in \( \text{Sys}\text{-Faculty} \) instance. Several other examples of translated queries are given in Table 1.

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

\[
\begin{align*}
\text{property}(\text{source}, \text{val}) & \rightarrow \text{inheritable}(\text{source}, \text{prop}, \text{object}), \text{property}(\text{object}, \text{val}), \\
\text{property}(\text{source}, \text{val}) & \rightarrow \text{inheritable}(\text{source}, \text{prop}, \text{object}), \text{sys}\text{InstanceOf}(\text{object}, \text{sys}\text{-obj}), \text{property}(\text{sys}\text{-obj}, \text{val}).
\end{align*}
\]

### 5 Related Research

To the best of our knowledge, OWL++ is the first language that extends OWL with support for non-monotonic reasoning, method inheritance, and polymorphism, in a single platform. Non-monotonic reasoning, in general exception handling, has also been studied in the form of defeasible reasoning [3], where one rule is given priority over the other rules. It specifies every exception rule explicitly, thus making knowledge representation clumsy and compromising the naturalness of knowledge representation. A similar approach has also be taken in [5] called Courteous Logic Programs (CLP), to represent non-monotonic inheritance reasoning in process ontologies. However, the extension of DL with default theories proposed in [4] closely resembles to our work. It relies on specificity information of rule, similar to our context information, to make strict orders of default rules. But it requires several iterations to derive all valid consequences, which is less attractive in large scale implementations such as in semantic Web.

### 6 Summary

In this paper, we presented a new language called OWL++ that extends OWL with two modes of inheritance \textit{overriding} and \textit{inflating}; and three types of inheritance \textit{value}, \textit{code} and \textit{null}. The main purpose was to enhance expressivity of OWL and to reduce the impedance mismatch between OWL and OO languages for efficient application development. We presented detailed syntax and semantics of OWL++ and demonstrated its usefulness and salient features using several real life examples. We adapted OWL and SWRL syntax, and gave a rewriting based semantics of OWL++ for maximum backward compatibility. It allows us to use existing technologies such as Jena framework, which combines general purpose rule reasoner with the well known OWL reasoner Pellet, to build OWL++ reasoner.

### References


