April 1,2006

Extensions to OWL - Functions and Inductive Predicates
Version 0.1

ICS-16763-TR-07-005
SRI Project No. 16763
Contract No. FA8750-05-C-0230

Prepared by
Daniel Elenius
SRI International
333 Ravenswood Ave
Menlo Park, CA 94025-3493

Prepared for
Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington, VA 22203-1714

Acknowledgment and Disclaimer:
This material is based upon work supported by the Defense Advanced Projects Agency and the United State Air Force under Contract Number FA8750-05-C-0230.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the Defense Advanced Research Projects Agency and the United States Air Force.

1 Introduction

We describe how to extend OWL with functions and inductive predicates. Each extension is independent from the other, and the functional extension is composed of several parts, which are partly independent of each other. We first discuss the motivation for these extensions, and then we describe our (syntactic and semantic) approach in detail.

2 Motivation

There are two overarching motivations for both of the extensions we propose: *Expressiveness* and *Reasoning efficiency.*

We are considering here rather expressive logics. For this reason, we take SWRL FOL[1] as a starting point, as it already has most of the syntax and semantics that we need.

2.1 Inductive Predicates

Inductive predicates[2] are a way of introducing "local closed-world assumptions," which helps when reasoning about negative information. Intuitively, an inductive predicate is completely defined by the axioms in which it occurs. For example, suppose we have

$A \sqsubseteq B$
$A \sqsubseteq C$
$\mathit{inductive}(A)$

and nothing more is said about A. If only the first two axioms were present, we would never be able to prove that any individual is *not* a member of A. However, the meaning of declaring A to be inductive is that $A = B \sqcup C$. So, if we can prove that an individual is not in $B \sqcup C$, then we know that it is not in A.

Of course, OWL is already expressive enough to say $A = B \sqcup C$ directly. However, declaring a predicate to be inductive is an *extensible* way to do this. The definition of the predicate can be spread out across multiple ontologies, and the predicate is not "closed" until the knowledge base is loaded.

One application of this is in XG spectrum policies, where we usually have the "meta-rule"

```
transmit iff allow and not disallow
```

and individual policies have rules of the type

`allow if ...` or
`disallow if ...`

Now, if this was all we had to go on, we would never be able to prove `not disallow`, because the `disallow` predicate is not fully defined. Therefore, we must declare `disallow` as inductive, closing it off so that we can prove its negation.

2.2 Functions

OWL has no way to specify arbitrary computations. The so-called SWRL built-ins imply some kind of computations, but these cannot be defined in OWL (and are currently not defined anywhere else either). One could view these built-ins as "procedural attachments," and possibly add more such attachments on an as-needed basis. We have argued against such an approach elsewhere, on the grounds that the resulting system becomes

[1] `http://www.w3.org/Submission/SWRL-FOL/`
[2] By "predicates" we mean, from here on, OWL classes or nonfunctional properties.

nonverifiable and nonextensible. A better approach is to extend the language so that anything one might need can be defined *in* the language.

To do this in a practical way, we need a decent treatment of functions. OWL does have so-called functional properties, but is very limited in what one can do with them. SWRL FOL extends OWL toward First-Order Logic, but stops short of taking functions seriously. The main limitations are

- SWRL FOL does not introduce functional terms. There is no notion of function application, except in the special case of equality constraints.
- n-ary functions. SWRL FOL does suggest a way to handle these, but we found it useful to generalize slightly on their method.
- Constructors. Most functional languages[3] have a way to declare a function to be a constructor, which implies some uniqueness constraints on constructed values.

We take the minimalist approach of treating OWL's functional properties as functions, rather than adding even more syntax for functions in OWL.

The issue that makes it difficult to add functions to OWL is that in OWL, data structure is represented using *relations* (i.e., properties), whereas in functional languages, structure is represented by compound *terms*. We need to retain the OWL philosophy of using relations as the main form of structure, yet add functions in a meaningful way. Our functions need to be able to return arbitrary RDF graphs.

The main innovation to make functions-in-OWL work is the ϵ operator.[4] ϵ is a term constructor. $\epsilon x\ \phi(x)$ denotes *some* x such that $\phi(x)$ is true. Put differently, $\epsilon x\ \phi(x)$ is a *witness* to the formula $\exists x : \phi(x)$.

This lets us "push existentials into terms," so that we can define some (anonymous) individual with certain properties as the return value of a function. This is the glue we need between the relational and the functional world.

Function application can be defined using ϵ. For example, $f(g(x)) = \epsilon z\ f(\epsilon y\ g(x, y), z)$ However, we choose to add function application directly as it may be more intuitive.

Finally, *constructors* are a way to define uniqueness of individuals or values constructed using functions. This allows us to define *algebraic data types* [EM85] in OWL.

3 Approach

In [Ele07], we have described a translation from SWRL FOL, which includes OWL and SWRL, to our Universal Policy Logic (UPL). Here, we extend the syntax of SWRL FOL and OWL, and extend our translation to include the new syntactic elements.

We add the new syntactic elements to the Abstract Syntax and the XML Presentation Syntax of OWL and SWRL FOL. We do *not* define an RDF/XML syntax for the new elements, for two reasons. The first reason is that SWRL FOL does not have an RDF/XML syntax, and we do want to use SWRL FOL as a starting point, as it provides most of the syntax we need. The second reason is that layering these expressive logics on top of RDF is problematic even in principle, because RDF has its own semantics.[5] RDF can be used to express the syntax of a more expressive language, but the new syntactic elements are given a meaning by the RDF semantics, which is often different from the intended meaning, as acknowledged in the SWRL specification,[6] and also discussed in [HPSvH03].

[3]Including Functional RuleML
[4]See http://plato.stanford.edu/entries/epsilon-calculus/ for some background.
[5]http://www.w3.org/TR/rdf-mt/
[6]http://www.w3.org/Submission/SWRL/#6

Furthermore, the XML Presentation Syntax is more amenable to machine processing (e.g. XSLT transformations) than the RDF/XML syntax of OWL/SWRL. In addition, XML Schema allows one to express more constraints on the syntax, which means that an XML Schema validator can find syntactic problems that require specialized code to find if the RDF representation is used.

It should be noted that one can still refer to other ontologies in RDF/XML format from ontologies in the XML Presentation Syntax.

3.1 Abstract Syntax

We add a an optional `Constructor` attribute to the OWL abstract syntax by augmenting the `individual` and `axiom` productions, and an optional `Inductive` attribute to the `Class`, `DatatypeProperty` and `ObjectProperty` productions as shown in Figure 1.

Furthermore, we extend the SWRL abstract syntax productions for `i-object` and `d-object` to support the ϵ construct and function application as shown in Figure 2.

Besides adding the ϵ construct, we also add an abbreviated form of a common special case,

$$(R_k = i_k \; U_j = d_j)$$

which has the same meaning as

$$\texttt{eps}\ x\ R_k(x, i_k)\ \texttt{and}\ U_j(x, d_j)$$

where k, j are indices, i.e., there can be any number of the R, U atoms, in any order. Note that R_k, U_j do not have to be functional properties. Note also that this construct can be nested.

This is similar to the SWRL FOL syntactic sugar mentioned above. This abbreviated syntax is especially useful for function application, as shown in examples below.

3.2 XML Syntax

We show how the necessary elements could be added to the existing OWL, SWRL, and SWRL FOL schemas. Unfortunately, we cannot specify our extensions in an external module, because the changes take place inside the existing syntax structure.

The augmented elements of the OWL XML syntax are shown in Figures 3, 4 and 5, and the augmented SWRL XML syntax in Figure 6.

3.3 Translation

We expand the definitions of some of the translation functions defined in [Ele07]: $\mathcal{T}_\gamma$ as shown in Table 1, $\mathcal{T}_{io}$ as in Table 2, and $\mathcal{T}_{do}$ as in Table 3. This translation fully defines the semantics of the new, extended, language.

The AND function is defined as in [Ele07], i.e., it is the UPL equivalent of $\bigwedge$.

4 Notes

Some SWRL FOL `Assertions` will be of special importance. In UPL, *equations* are formulas of the type $(\forall \bar{x} : T)L = R$ and *conditional equations* formulas of the type $(\forall \bar{x} : T)L = R {\Leftarrow} C$. Equations provide useful expressiveness as they allow us to express, in the most natural way, the values of functions applied to arguments.

Equations also provide good reasoning efficiency, as they can be treated as *rewriting rules*, where occurences of the left side of an equation can simply be replaced by the right side.

SWRL FOL can express equations and conditional equations, but they are somewhat inconspicuous. Unconditional equations have one of the following SWRL atoms inside an `Assertion`:

$\texttt{sameAs}(i_1\ i_2)$
$\texttt{builtIn}(\texttt{swrlb}:\texttt{equal}(d_1 \ldots d_n))$
$p(x\ y)$

where the is are i-objects, ds are d-objects, and p is a functional property. The UPL forms of the above equations are simply $i_1 = i_2$, $d_k = d_l$ for all $k, l \in 1..n$, and $p(x) = y$.

Conditional equations in SWRL FOL have one of the forms

$\texttt{Assertion(forall(... implies(}C\ H\texttt{)))}$
$\texttt{Implies(Antecedent(}C\texttt{) Consequent(}H\texttt{))}$

where H is an atom of one of the three types above, and C is an arbitrary formula (the *condition* of the equation).

In the following, we will refer to all the SWRL forms above as equations.

One serious shortcoming in OWL is the limitation to one argument for functions (the second predicate position being the "return value"). Our solution is to name all the arguments, and introduce a new individual, which has properties with those names pointing to the values. This causes some complications, as shown in the examples below.

Another problem with OWL is that one cannot define functions that take data values as arguments. The domain of datatype properties cannot be, for example, `xsd:int`. This shows up in the syntax, where there is no function application on "d-objects". The solution is to use the multiple-arguments method with only one, data-valued argument.

A third issue is that the SWRL built-ins are not actually datatype properties. Furthermore, they often use the first argument as the result.

5 Examples

5.1 Append

Our first example is the function `append`, which takes two `rdf:Lists`, and returns the combined list. We assume that the lists only contain literals.[7]

In UPL, it can be defined as follows:

$(\forall HT, H, T, Z, Y : \texttt{Thing})$
$\quad \texttt{append}(HT, Y) = (\epsilon HZ : \texttt{Thing})\texttt{rdf}:\texttt{first}(HZ, H)\ \texttt{and}\ \texttt{rdf}:\texttt{rest}(HZ, Z) \Leftarrow$
$\qquad \texttt{rdf}:\texttt{first}(HT, H)\ \texttt{and}\ \texttt{rdf}:\texttt{rest}(HT, T)\ \texttt{and}\ \texttt{append}(T, Y) = Z$

In the abstract syntax, we could define this as a SWRL FOL `Assertion`, but in fact, we can also do it as a regular SWRL `Implies`, which saves us the trouble of writing out all the universally quantified variables. Note that all the identifiers below should strictly speaking be URIs.

```
ObjectProperty(append Functional)
ObjectProperty(list1 Functional)
```

[7]rdf:List is not really part of OWL DL – we have to define our own list classes, and we have to decide whether the `first` property is a datatype property or an object property, and therefore whether the list is an "object list" or a "data list". There is no way to define mixed lists in OWL DL.

```
ObjectProperty(list2 Functional)

Implies(
  Antecedent(
    sameAs(list1(I-variable(args)) I-variable(HT))
    sameAs(list2(I-variable(args)) I-variable(Y))
    rdf:first(I-variable(HT) I-variable(H))
    rdf:rest(I-variable(HT) I-variable(T))
    sameAs(append(list1=I-variable(T) list2=I-variable(Y))
           I-variable(Z)))
  Consequent(
    sameAs(append(I-variable(args))
           (rdf:first=I-variable(H) rdf:rest=I-variable(Z)))
```

This example makes use of two of our extensions: function application (for `append`, `list1`, and `list2`), and the "multi-argument" shorthand form (in the right side of the equation in the consequent).

Either way, to use this function one would write, for example, the term

```
append(list1=(rdf:first=a rdf:next=(rdf:first=b rdf:next=rdf:nil))
       list2=(rdf:first=c rdf:next=(rdf:first=d rdf:next=rdf:nil)))
```

which would give the result of appending the list $[a, b]$ to the list $[c, d]$, that is,

```
(rdf:first=a rdf:next=(rdf:first=b rdf:next=
  (rdf:first=c rdf:next=(rdf:first=d rdf:next=rdf:nil))))
```

Note that this is a term with four ϵs in it, one for each list cell.

An even shorter form of the rule is

```
Implies(
  Antecedent(
    sameAs(append(list1=I-variable(T)
                  list2=I-variable(Y))
           I-variable(Z)))
  Consequent(
    sameAs(append(list1=(rdf:first=I-variable(H) rdf:rest=I-variable(T))
                  list2=I-variable(Y))
           (rdf:first=I-variable(H) rdf:rest=I-variable(Z)))
```

Instead of `args` in the left side of the equation in the consequent, we now have a nested "multi-argument" term. Instead of "unwrapping" the arguments and their structure in the antecedent, we do it in the equation in the consequent. However, this version does not work. Consulting the semantics of the ϵ construct, we note that it denotes a *unique* (but arbitrary) element. But there is no guarantee that the element denoted by the argument list, or the actual arguments, in an `append` term such as the one above, is the same element as the one referred to in the left side of the equation. Lists with the same elements are not necessarily the same lists, and the same goes for the "multi-argument" individuals. That is to say, these structures are not *extensional.* We could make them extensional by adding additional axioms, and then the shorter form above would work.

It should also be noted that pattern matching with ϵ terms is very difficult.

5.2 Natural Numbers

Our second example will show our new power to define algebraic data types. We will use the paradigmatic example of the natural numbers. Of course, it would be counter-productive to actually define the natural numbers in OWL, as they are already built in, but the same technique can be used for countless other cases, and this may be a familiar one.

The basic specification, without any operations on the numbers, is as follows

```
Class(Nat Inductive)
Individual(Zero type(Nat) Constructor)
ObjectProperty(succ domain(Nat) range(Nat) Constructor)
```

Now, for example, the number 3 is represented by the term `succ(succ(succ(Zero)))`.

We can now define addition of natural numbers,

```
Class(AddNatArgs)
ObjectProperty(addNat domain(AddNatArgs) range(Nat) Functional)
ObjectProperty(n1 domain(AddNatArgs) range(Nat) Functional)
ObjectProperty(n2 domain(AddNatArgs) range(Nat) Functional)

Assertion(forall(I-variable(n Nat) addNat((n1=n n2=Zero) n)))

Assertion(forall(I-variable(args AddNatArgs) I-variable(n Nat)
                 I-variable(m Nat)
  implies(
    n1(I-variable(args) I-variable(n)) and
      n2(I-variable(args) I-variable(succ(m)))
    addNat(I-variable(args)
           succ(addNat(n1=I-variable(n) n2=I-variable(m)))))))
```

The first thing to note is that we have completely specified the type of the `addNat` function. This is somewhat awkward because of the treatment of multiple arguments. A new class `AddNatArgs` is introduced as the argument container, and `addNat`, `n1` and `n2` all have this class as their domain, and `Nat` as their range.

This example makes use of two more of our extensions: constructors and inductive predicates.

The second equation could have been written as an `Implies` instead of as an `Assertion`.[8]

This time, we have written the equations in "atom form". Because, for example, `addNat` is functional, `addNat(X Y)` is equivalent to `sameAs(addNat(X) Y)` (both will be translated to the UPL formula `addNat(X)=Y`).

Note that the only reason the second equation is conditional is because of the wrapping of multiple arguments into one individual.

References

[Ele07] Daniel Elenius. Translating OWL DL to the Universal Policy Logic, 2007.

[EM85] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1*. Springer-Verlag, 1985.

[HPSvH03] I. Horrocks, P. Patel-Schneider, and F. van Harmelen. From SHIQ and RDF to OWL: The making of a web ontology language. *Journal of Web Semantics*, 1(1):7–26, 2003.

[8]The SWRL FOL syntax is highly redundant. Everything, including all the OWL axioms, could be written as `Asertions`, since FOL contains OWL DL.

Figure 1: OWL Abstract Syntax augmented with `Constructor`, `Inductive`, and n-ary functions.

```
individual ::= 'Individual(' [ individualID ] { annotation } { 'type(' type ')' } { value }
                             ['Constructor'] ')'

axiom ::= 'Class(' classID  ['Deprecated'] modality { annotation } { description }
                  ['Inductive'] ')'

axiom ::= 'DatatypeProperty(' datavaluedPropertyID ['Deprecated'] { annotation }
              { 'super(' datavaluedPropertyID ')' }
              ['Functional' | 'Functional' 'Constructor' | 'Inductive']
              { 'domain(' classID' ')' } { 'range(' dataRange ')' } ')'
       | 'ObjectProperty(' individualvaluedPropertyID ['Deprecated'] { annotation }
              { 'super(' individualvaluedPropertyID ')' }
              [ 'inverseOf(' individualvaluedPropertyID ')' ] [ 'Symmetric' ]
              [ 'InverseFunctional' |
                'Functional' ['Constructor'] ['InverseFunctional'] |
                'Transitive' |
                'Inductive' ['Transitive' | 'InverseFunctional'] ]
              { 'domain(' classID ')' } { 'range(' classID ')' } ')'
       | 'AnnotationProperty(' annotationPropertyID { annotation } ')'
       | 'OntologyProperty(' ontologyPropertyID { annotation } ')'
```

Figure 2: SWRL abstract syntax augmented with the ϵ construct and function application.

```
named-argument ::= individualvaluedPropertyID '=' i-object
                 | datavaluedPropertyID '=' d-object

i-object ::= i-variable | individualID | 'eps(' i-variable foformula ')' |
             individualvaluedPropertyID '(' i-object ')' | '(' { named-argument } ')' |
             { named-argument }
d-object ::= d-variable | dataLiteral | 'eps(' d-variable foformula ')' |
             datavaluedPropertyID '(' i-object ')' | '(' { named-argument } ')' |
             { named-argument }
```

Table 1: Translation of declarations. A, U, R, o, and D are URI references.

Abstract Syntax, Declarations	Translation, $\mathcal{T}_\gamma$
$\texttt{Class}(A \ldots)$	$A : \texttt{Thing} \rightarrow \texttt{Prop}$
$\texttt{DatatypeProperty}(U \ldots \texttt{Functional})$	$U : \texttt{Thing} \rightarrow \texttt{Data}$
$\texttt{DatatypeProperty}(U \ldots)$	$U : \texttt{Thing} * \texttt{Data} \rightarrow \texttt{Prop}$
$\texttt{ObjectProperty}(R \ldots \texttt{Functional})$	$R : \texttt{Thing} \rightarrow \texttt{Thing}$
$\texttt{ObjectProperty}(R \ldots)$	$R : \texttt{Thing} * \texttt{Thing} \rightarrow \texttt{Prop}$
$\texttt{Individual}(o \ldots)$	$o : \texttt{Thing}$
$\texttt{Datatype}(D)$	$D : \texttt{Data} \rightarrow \texttt{Prop}$
$\texttt{DatatypeProperty}(U \ldots \texttt{Functional Constructor})$	$U : \texttt{Thing} \rightarrow \texttt{Data},\ \texttt{ctor}(U)$
$\texttt{ObjectProperty}(R \ldots \texttt{Functional Constructor})$	$U : \texttt{Thing} \rightarrow \texttt{Thing},\ \texttt{ctor}(R)$
$\texttt{Class}(A \ldots \texttt{Inductive})$	$A : \texttt{Thing} \rightarrow \texttt{Prop},\ \texttt{ind}(A)$
$\texttt{DatatypeProperty}(U \ldots \texttt{Inductive})$	$U : \texttt{Thing} * \texttt{Data} \rightarrow \texttt{Prop},\ \texttt{ind}(U)$
$\texttt{ObjectProperty}(R \ldots \texttt{Inductive})$	$R : \texttt{Thing} * \texttt{Thing} \rightarrow \texttt{Prop},\ \texttt{ind}(U)$

Table 2: Translation of SWRL I-Objects.

Abstract Syntax, I-Objects (i)	Translation, $\mathcal{T}_{io}(i)$
`I-variable`(i) $(i$ a URI reference$)$	i
o $(o$ a URI reference$)$	o
`eps`$(v\ F)$	$(\epsilon\ \mathcal{T}_{io}(v) :$ `Thing`$)\mathcal{T}_f(F)$
If `ObjectProperty`$(R \ldots$ `Functional`$) \in K$	
$R\ i$	$R\ \mathcal{T}_{io}(i)$
If `ObjectProperty`$(R_k\ \ldots) \in K$ and `DatatypeProperty`$(U_j\ \ldots) \in K$	
$(R_k = i_k\ U_j = d_j)$	$(\epsilon x :$ `Thing`$)AND_k\ \mathcal{T}_r(R_k, \mathcal{T}_{io}(i_k))$ `and` $AND_j\ \mathcal{T}_r(U_j, \mathcal{T}_{do}(d_j))$

Table 3: Translation of SWRL D-Objects.

Abstract Syntax, D-Objects (d)	Translation, $\mathcal{T}_{do}(d)$
`D-variable`(d) $(d$ a URI reference$)$	d
v	$\mathcal{T}_v(v)$
`eps`$(v\ F)$	$(\epsilon\ \mathcal{T}_{do}(v) :$ `Data`$)\mathcal{T}_f(F)$
If `DatatypeProperty`$(U \ldots$ `Functional`$) \in K$	
$U\ i$	$U\ \mathcal{T}_{io}(i)$
If all `ObjectProperty`$(R_k\ \ldots) \in K$ and all `DatatypeProperty`$(U_j\ \ldots) \in K$	
$(R_k = i_k\ U_j = d_j)$	$(\epsilon x :$ `Data`$)AND_k\ \mathcal{T}_r(R_k, \mathcal{T}_{io}(i_k))$ `and` $AND_j\ \mathcal{T}_r(U_j, \mathcal{T}_{do}(d_j))$

Figure 3: OWL XML syntax, with new constructs added (1/3).

```
<xsd:element name="Individual">
  <xsd:annotation>
    <xsd:documentation>
      Used by 'Ontology' [Lite/DL/Full]
    </xsd:documentation>
    <xsd:documentation>
      Facts about individuals have several types and
      property information, including nested facts.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:extension base="owlx:annotated">
        <xsd:sequence minOccurs="0" maxOccurs="unbounded">
          <xsd:choice>
            <xsd:element name="type">
              <xsd:complexType>
                <xsd:sequence minOccurs="0" maxOccurs="unbounded">
                  <xsd:group ref="owlx:description" />
                </xsd:sequence>
                <xsd:attribute name="name" type="owlx:ClassName" />
              </xsd:complexType>
            </xsd:element>
            <xsd:element ref="owlx:DataPropertyValue" />
            <xsd:element ref="owlx:ObjectPropertyValue" />
          </xsd:choice>
        </xsd:sequence>
        <xsd:attribute name="name" type="owlx:IndividualName" />
        <xsd:attribute name="constructor" type="xsd:boolean" />    <!-- added -->
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:element>

<xsd:element name="Class">
  <xsd:annotation>
    <xsd:documentation>
      'Class' contains a non-empty sequence of descriptions.
      Attributes provide a class name and the modality.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:extension base="owlx:annotated">
        <xsd:sequence minOccurs="0" maxOccurs="unbounded">
          <xsd:group ref="owlx:description" />
        </xsd:sequence>
        <xsd:attribute name="name" type="owlx:ClassName" use="required" />
        <xsd:attribute name="complete" type="xsd:boolean" use="required" />
        <xsd:attribute name="deprecated" type="xsd:boolean" />
        <xsd:attribute name="inductive" type="xsd:boolean" />    <!-- added -->
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:element>
```

Figure 4: OWL XML syntax, with new constructs added (2/3).

```
<xsd:element name="DatatypeProperty">
  <xsd:annotation>
    <xsd:documentation>
      'owlx:dataPropDomainRange' group is included from either
         + owl1-lite-dataDomainRangeGroup.xsd [Lite] or
         + owl1-dl-dataDomainRangeGroup.xsd [DL/Full]
    </xsd:documentation>
    <xsd:documentation>
      'owlx:dataPropInverseFuncAttr' attribute group is
      included from either
         + owl1-lite-dataPropInverseFuncAttr.xsd [Lite/DL] or
         + owl1-full-dataPropInverseFuncAttr.xsd [Full]
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:extension base="owlx:annotated">
        <xsd:sequence minOccurs="0" maxOccurs="unbounded">
          <xsd:choice>
            <xsd:element name="superProperty">
              <xsd:complexType>
                <xsd:attribute name="name" type="owlx:DataPropertyName"
                               use="required"/>
              </xsd:complexType>
            </xsd:element>
            <xsd:group ref="owlx:dataPropDomainRange" />
          </xsd:choice>
        </xsd:sequence>
        <xsd:attribute name="name" type="owlx:DataPropertyName" use="required" />
        <xsd:attributeGroup ref="owlx:dataPropInverseFuncAttr" />
        <xsd:attribute name="deprecated" type="xsd:boolean" />
        <xsd:attribute name="functional" type="xsd:boolean" />
        <xsd:attribute name="constructor" type="xsd:boolean" />   <!-- added -->
        <xsd:attribute name="inductive" type="xsd:boolean" />     <!-- added -->
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:element>
```

Figure 5: OWL XML syntax, with new constructs added (3/3).

```
<xsd:element name="ObjectProperty">
  <xsd:annotation>
    <xsd:documentation>
      'owlx:objectPropDomainRange' group is included from either
         + owl1-lite-domainRangeGroup.xsd [Lite] or
         + owl1-dl-domainRangeGroup.xsd [DL/Full]
    </xsd:documentation>
    <xsd:documentation>
      'owlx:objectPropInverseFuncAttr' attribute group is
      included from either
         + owl1-lite-objectPropInverseFuncAttr.xsd [Lite] or
         + owl1-dl-objectPropInverseFuncAttr.xsd [DL/Full]
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:extension base="owlx:annotated">
        <xsd:sequence minOccurs="0" maxOccurs="unbounded">
          <xsd:choice>
            <xsd:element name="superProperty">
              <xsd:complexType>
                <xsd:attribute name="name" type="owlx:IndividualPropertyName"
                               use="required"/>
              </xsd:complexType>
            </xsd:element>
            <xsd:group ref="owlx:objectPropDomainRange" />
          </xsd:choice>
        </xsd:sequence>
        <xsd:attribute name="name" type="owlx:IndividualPropertyName"
                       use="required" />
        <xsd:attribute name="inverseOf"  type="owlx:IndividualPropertyName" />
        <xsd:attribute name="transitive" type="xsd:boolean" />
        <xsd:attribute name="symmetric"  type="xsd:boolean" />
        <xsd:attribute name="functional" type="xsd:boolean" />
        <xsd:attributeGroup ref="owlx:objectPropInverseFuncAttr" />
        <xsd:attribute name="deprecated" type="xsd:boolean" />
        <xsd:attribute name="constructor" type="xsd:boolean" />   <!-- added -->
        <xsd:attribute name="inductive" type="xsd:boolean" />     <!-- added -->
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:element>
```

Figure 6: SWRL XML syntax, with new constructs added.

```
<xsd:group name="iObject">
  <xsd:choice>
    <xsd:element ref="owlx:Individual" />
    <xsd:element ref="ruleml:var" />
    <xsd:element ref="swrlx:epsilon" />   <!-- added -->
    <xsd:element ref="swrlx:apply" />     <!-- added -->
    <xsd:element ref="swrlx:multiarg" />  <!-- added -->
  </xsd:choice>
</xsd:group>

<xsd:group name="dObject">
  <xsd:choice>
    <xsd:element ref="owlx:DataValue" />
    <xsd:element ref="ruleml:var" />
    <xsd:element ref="swrlx:epsilon" />   <!-- added -->
    <xsd:element ref="swrlx:apply" />     <!-- added -->
    <xsd:element ref="swrlx:multiarg" />  <!-- added -->
  </xsd:choice>
</xsd:group>

<!-- added -->
<xsd:element name="epsilon">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="ruleml:var" />
      <xsd:group name="owlf:formula" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<!-- added -->
<xsd:element name="apply">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:choice>
        <xsd:element ref="owlx:IndividualPropertyName" />
        <xsd:element ref="owlx:DataPropertyName" />
      </xsd:choice>
      <xsd:group name="swrlx:iObject" minOccurs="0" maxOccurs="unbounded" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<!-- added -->
<xsd:element name="multiarg">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:choice>
        <xsd:sequence>
          <xsd:element ref="owlx:IndividualPropertyName" />
          <xsd:group name="swrlx:iObject" minOccurs="0" maxOccurs="unbounded" />
        </xsd:sequence>
        <xsd:sequence>
          <xsd:element ref="owlx:DataPropertyName" />
          <xsd:group name="swrlx:dObject" minOccurs="0" maxOccurs="unbounded" />
        </xsd:sequence>
      </xsd:choice>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```