Chapter 2

AN INTRODUCTION TO THE OWL WEB ONTOLOGY LANGUAGE

Jeff Heflin
Lehigh University

Abstract:

Key words:

1. INTRODUCTION

The OWL Web Ontology Language is an international standard for encoding and exchanging ontologies and is designed to support the Semantic Web. The concept of the Semantic Web is that information should be given explicit meaning, so that machines can process it more intelligently. Instead of just creating standard terms for concepts as is done in XML, the Semantic Web also allows users to provide formal definitions for the standard terms they create. Machines can then use inference algorithms to reason about the terms. For example, a semantic web search engine may conclude that a particular CD-read/write drive matches a query for "Storage Devices under $100." Furthermore, if two different sets of terms are in turn defined using a third set of common terms, then it is possible to automatically perform (partial) translations between them. It is envisioned that the Semantic Web will enable more intelligent search, electronic personal assistants, more efficient e-commerce, and coordination of heterogeneous embedded systems.

A crucial component to the Semantic Web is the definition and use of ontologies. For over a decade, artificial intelligence researchers have studied the use of ontologies for sharing and reusing knowledge (Gruber 1993, Guarino 1998, Noy and Hafner 1997). Although there is some disagreement

as to what comprises an ontology, most ontologies include a taxonomy of terms (e.g., stating that a *Car* is a *Vehicle*), and many ontology languages allow additional definitions using some type of logic. Guarino (1998) has defined an ontology as "a logical theory that accounts for the intended meaning of a formal vocabulary." A common feature in ontology languages is the ability to extend preexisting ontologies. Thus, users can customize ontologies to include domain specific information while retaining the interoperability benefits of sharing terminology where possible.

OWL is an ontology language for the Web. It became a World Wide Web Consortium (W3C) Recommendation[1] in February 2004. As such, it was designed to be compatible with the eXtensible Markup Language (XML) as well as other W3C standards. In particular, OWL extends the Resource Description Framework (RDF) and RDF Schema, two early Semantic Web standards endorsed by the W3C. Syntactically, an OWL ontology is a valid RDF document and as such also a well-formed XML document. This allows OWL to be processed by the wide range of XML and RDF tools already available.

Semantically, OWL is based on description logics (Baader et al. 2002). Generally, description logics are a family of logics that are decidable fragments of first-order predicate logic. These logics focus on describing classes and roles, and have a set-theoretic semantics. Different description logics include different subsets of logical operators. Two of OWL's sublanguages closely correspond to known description logics: OWL Lite corresponds to the description logic $\mathcal{SHIF}$(D) and OWL DL corresponds to the description logic $\mathcal{SHOIN}$(D) (Horrocks and Patel-Schneider 2003). For a brief discussion of the differences between the different OWL sublanguages, see Section 3.4.

In this chapter, I will provide an introduction to OWL. Due to limited space, this will not be a full tutorial on the use of the language. My aim is to describe OWL at a sufficient level of detail so that the reader can see the potential of the language and know enough to start using it without being dangerous. The reader is urged to look at the OWL specifications for any details not mentioned here. In particular, the OWL Guide (Smith et al. 2004) is a very good, comprehensive tutorial. The book *A Semantic Web Primer* (Antoniou and van Harmelen 2004) also provides a readable introduction to XML, RDF and OWL in one volume.

The rest of this chapter is organized as follows. The second section discusses enough RDF and RDF Schema to provide context for understanding OWL. Section 3 discusses the basics of OWL with respect to classes, properties, and instances. Section 4 introduces more advanced OWL

[1] Recommendation is the highest level of endorsement by the W3C. Other W3C Recommendations include HTML 4.0 and XML.

concepts such as boolean combinations of classes and property restrictions. Section 5 focuses on the interrelationship of OWL documents, particularly with respect to importing and versioning. Section 6 provides a gentle warning about how OWL's semantics can sometimes be unintuitive. Section 7 concludes.

2. RDF AND RDF SCHEMA

RDF is closely related to semantic networks. Like semantic networks, it is a graph-based data model with labeled nodes and directed, labeled edges. This is a very flexible model for representing data. The fundamental unit of RDF is the statement, which corresponds to an edge in the graph.

An RDF statement has three components: a subject, a predicate, and an object. The **subject** is the source of the edge and must be a resource. In RDF, a resource can be anything that is uniquely identifiable via a Uniform Resource Identifier (URI). More often than not, this identifier is a Uniform Resource Locator (URL), which is a special case of URI. However, URIs are more general than URLs. In particular, there is no requirement that a URI can be used to locate a document on the Internet. The **object** of a statement is the target of the edge. Like the subject, it can be a resource identified by a URI, but it can alternatively be a literal value like a string or a number. The **predicate** of a statement determines what kind of relationship holds between the subject and the object. It too is identified by a URI.

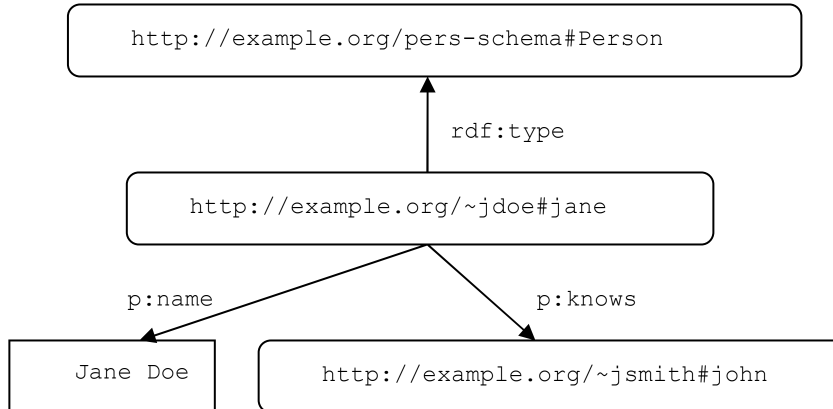

Figure 2-1. An example RDF graph.

Figure 2-1 shows an example graph with three statements. One statement has subject http://example.org/~jdoe#jane, predicate p:knows and object

http://example.org/~jsmith#john. In other words, this statement represents that "Jane knows John." The statement with predicate p:name is an example of a statement that has a literal value (i.e., "Jane Doe") as its object. This statement indicates that Jane's name is "Jane Doe." Note, p:knows and p:name are examples of qualified names, which will be explained in the next subsection. The third statement declares Jane to be a Person.

2.1 XML Serialization Syntax for RDF

In order to exchange RDF graphs, the W3C recommendation defines an XML syntax for them. Before we can discuss the syntax, it is important to introduce some basic XML terminology. XML is a markup language, and as such uses tags to provide additional information about text. Tags are indicated by angle brackets, such as the <p>, <img> and <a> tags in the HyperText Markup Language (HTML). The main syntactic unit in XML is the element, which typically consists of a start tag (such as <p>), an end tag (such as </p>) and some content enclosed between the tags. The content may be either text or more elements. If it contains other elements then these are called subelements. Every well-formed XML document has exactly one outermost element which is called the root element. Some elements may have no content; such empty elements can be written with a single tag that has a slash before the closing angle bracket (e.g. <hr />). Elements can have attributes which are name-value pairs. The attributes are listed inside of the element's start tag.

The XML shown in Figure 2-2 is a serialization of the RDF graph in Figure 2-1. The first thing to notice is that the root element is rdf:RDF; all RDF documents usually have such a root element. The use of a colon in an element or attribute name indicates that it is a qualified name. Qualified names are used with XML namespaces to provide shorthand references for URIs. The xmlns:rdf attribute on the second line of the figure specifies that the "rdf" prefix is used as an abbreviation for the namespace "http://www.w3.org/1999/02/22-rdf-syntax-ns#". The xmlns:p attribute defines p as another prefix that can be used to form qualified names. Qualified names have the form *prefix:local_name*. To construct the full URI for a qualified name, simply append the local name part to the namespace that corresponds to the prefix.

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:p="http://example.org/pers-schema#">
  <rdf:Description rdf:about="http://example.org/~jdoe#jane">
    <p:knows rdf:resource="http://example.org/~jsmith#john" />
    <p:name>Jane Doe</p:name>
    <rdf:type
         rdf:resource="http://example.org/pers-schema#Person"/>
  </rdf:Description>
</rdf:RDF>
```

Figure 2-2. RDF/XML syntax for the RDF graph in Figure 2-1

The rdf:RDF element contains an rdf:Description subelement that is used to identify a resource and to describe some of its properties. Every rdf:Description element encodes one or more RDF statements. In the figure, the subject of each of the statements is the resource given by the "rdf:about" attribute, which has the URI "http://example.org/~jdoe#jane" as its value. This rdf:Description element has three property subelements, and thus encodes three statements. The first subelement is an empty element with the qualified name p:knows; based on the namespace declaration at the beginning of the document, this refers to the resource "http://example.org/pers-schema#knows". This is the predicate of the statement. Any resource that is used as a predicate is called a property. The rdf:resource attribute is used to specify that "http://example.org/~jsmith#john" is the object of the statement. In this case the object is a full URI, but it could also be a relative URI.

As we said earlier, it is also possible for statements to have literals as objects. The second subelement of the rdf:Description encodes such a statement. Note that this element has textual content. The corresponding statement has predicate "http://example.org/pers-schema#name" and object "Jane Doe." By wrapping this text in <p:name> start and end tags, we indicate that it is a literal.

The final subelement of the rdf:Description is rdf:type. Using the namespace declaration at the beginning of the document, we can determine that this refers to the predicate http://www.w3.org/1999/02/22-rdf-syntax-ns#type. This is a property defined in RDF that allows one to categorize resources. The rdf:resource attribute is used to specify the category; in this case "http://example.org/pers-schema#Person". In RDF, types are optional and there is no limit to the number of types that a resource may have.

In Figure 2-2, we used a full URI as the value for the rdf:about attribute in the rdf:Description element. Alternatively, we could have specified a relative URI such as "#jane". Such a reference would be resolved to a full URI by prepending the base URI for the document, which by default is the

URL used to retrieve it. Thus, if this document was retrieved from "http://example.org/~jdoe", then rdf:about="http://example.org/~jdoe#jane" and rdf:about="#jane" would be equivalent. However, many web servers may allow you to retrieve the same document using different URLs: for example, "http://example.org/~jdoe" and "http://example.org/home/jdoe" may both resolve to the document in the figure. If this is the case, then the relative URI will resolve to a different full URI depending on how the document was accessed! In order to prevent this, use an xml:base attribute in the rdf:RDF tag. The value of this tag will be used as the base URI for resolving all relative references, regardless of where the document was retrieved from.

RDF also supports an alternative syntax for identifying individuals. Instead of rdf:about, you may use rdf:ID. The intention is that rdf:ID is used in the primary description of the object, while rdf:about is used for references to the object. As such, the value of rdf:ID is always a URI fragment, as opposed to a full URI. The full URI can be constructed by appending the base URI, the symbol "#" and the fragment specified by the rdf:ID attribute. From the point of view of determining what statements are encoded by RDF XML, rdf:ID="jane" and rdf:about="#jane" are equivalent. However, the rdf:ID attribute places the additional constraint that the same fragment cannot be used in another rdf:ID within the document.

Given that the type of a resource is one of the most frequently used properties, RDF provides an abbreviated syntax. The syntax shown in Figure 2-3 is equivalent to that in Figure 2-2 above. The key difference is that the rdf:Description element is replaced with a p:Person element and that the rdf:type element is missing. Generally, using any element other than rdf:Description when describing an individual implicitly states that the individual is of the type that corresponds to that element's name. Note, in this figure we also demonstrate the use of xml:base and relative rdf:about references.

```
<rdf:RDF xml:base="http://example.org/~jdoe"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:p="http://example.org/pers-schema#">
  <p:Person rdf:about="#jane">
    <p:knows rdf:resource="http://example.org/~jsmith#john" />
    <p:name>Jane Doe</p:name>
  </p:Person>
</rdf:RDF>
```

Figure 2-3. An abbreviated syntax for rdf:type statements

When literal values are used in RDF, it is possible to assign them datatypes. This can be done by simply adding an rdf:datatype attribute to the

property element that contains the literal value. In Figure 2-4, we state that Jane's age is 30. Furthermore, we indicate that the value 30 should be interpreted as an "&xsd;integer". The occurrence of "&xsd;" in this value is an XML entity reference, and is shorthand for a complete URI. In this case, it corresponds to the text "http://www.w3.org/2001/XMLSchema#". This can be determined by looking at the second line of the example. Generally, entities can be defined using "!ENTITY", a name for the entity, and a string that is the content of the entity. All entities must be defined in the internal subset of the document type declaration using the "<!DOCTYPE>" syntax. When the entity is referenced using the "&*entityname*;" syntax, the textual content is substituted. Thus the rdf:datatype attribute of the p:age element has a value of "http://www.w3.org/2001/XMLSchema#integer". This states that the value of the property is an integer as defined by XML Schema (Biron and Malhotra 2004). It is common practice for RDF and OWL documents to refer to XML Schema datatypes in this way.

```
<!DOCTYPE rdf:RDF [
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">]>
<rdf:RDF xml:base="http://example.org/~jdoe"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:p="http://example.org/pers-schema#">
  <p:Person rdf:about="#jane">
    <p:age rdf:datatype="&xsd;integer">30</p:age>
  </p:Person>
</rdf:RDF>
```

Figure 2-4. Typed literals in RDF

The alert reader may wonder what the difference between entities and namespace prefixes are. Namespace prefixes can be used to create qualified names that abbreviate the names of elements and attributes. However, qualified names cannot be used in attribute values. Thus, one has to resort to the XML trick of entities in order to abbreviate these.

2.2 RDF Schema

By itself, RDF is just a data model; it does not have any significant semantics. RDF Schema is used to define a vocabulary for use in RDF models. In particular, it allows you to define the classes used to type resources and to define the properties that resources can have. An important point is that an RDF Schema document is simply a set of RDF statements. However, RDF Schema provides a vocabulary for defining classes and properties. In particular, it includes rdfs:Class, rdf:Property (from the RDF namespace), rdfs:subClassOf, rdfs:subPropertyOf, rdfs:domain, and

rdfs:range. It also include properties for documentation, including rdfs:label and rdfs:comment. One problem with RDF Schema is that it has very weak semantic primitives. This is one of the reasons for the development of OWL. Each of the important RDF Schema terms are either included directly in OWL or are superceded by new OWL terms. As such, we will not discuss RDF Schema in detail here, but instead will discuss the relevant constructors in their appropriate context in OWL.

3. OWL BASICS

As an ontology language, OWL is primarily concerned with defining terminology that can be used in RDF documents, i.e., classes and properties. Most ontology languages have some mechanism for specifying a taxonomy of the classes. In OWL, you can specify taxonomies for both classes and properties.

```
<!DOCTYPE rdf:RDF [
   <!ENTITY owl "http://www.w3.org/2002/07/owl#">]>
<rdf:RDF xmlns:owl ="http://www.w3.org/2002/07/owl#"
      xmlns:rdf ="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
      xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
<owl:Ontology rdf:about="">
   <rdfs:label>My Ontology</rdfs:label>
   <rdfs:comment>An example ontology</rdfs:comment>
</owl:Ontology>
<owl:Class rdf:ID="Person" />
<owl:Class rdf:ID="Man" />
   <rdfs:subClassOf rdf:resource="#Person" />
</owl:Class>
<owl:ObjectProperty rdf:ID="hasChild" />
<owl:ObjectProperty rdf:ID="hasDaughter">
   <rdfs:subPropertyOf rdf:resource="#hasChild" />
</owl:ObjectProperty>
<owl:DatatypeProperty rdf:ID="age" />
<owl:ObjectProperty rdf:ID="isParentOf">
   <owl:inverseOf rdf:resource="#isChildOf" />
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="isTallerThan">
   <rdf:type rdf:resource="&owl;TransitiveProperty" />
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="isFriendOf">
   <rdf:type rdf:resource="&owl;SymmetricProperty" />
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasSSN">
   <rdf:type rdf:resource="&owl;FunctionalProperty" />
   <rdf:type rdf:resource="&owl;InverseFunctionalProperty" />
</owl:ObjectProperty>
</rdf:RDF>
```

Figure 2-5. A simple OWL ontology

As shown in Figure 2-5, the root of an OWL document is an rdf:RDF element. This is because all OWL documents are RDF documents, providing some degree of compatibility between the two standards. Typically, the start tag for the rdf:RDF element will contain attributes for each of the namespace prefixes used in the document; most ontologies will declare at least the owl, rdf, and rdfs namespaces. The owl:Ontology element serves two purposes: first it identifies the current document as an ontology and second it serves as a container for metadata about the ontology. By using the empty string as the value for the rdf:about attribute, we indicate that the base URL of the document should be used as its URI. In this way, we effectively say that the document is an ontology. In this example, the ontology has an rdfs:label and an rdfs:comment, both of which are defined in RDF Schema. The rdfs:label property provides a human-readable name for the ontology, while the

rdfs:comment property provides a textual description of the ontology. Both of these might be used to describe the ontology in an ontology library.

3.1 Classes

In RDF, an object of an rdf:type statement is implicitly a class, that is it represents a set of resources. In OWL, we can explicitly declare this resource to be a class by stating that it is of rdf:type owl:Class. Syntactically, this amounts to using an owl:Class element. In Figure 2-5, the ontology describes two classes: Person and Man. Note, it is standard RDF and OWL convention to name classes with singular nouns. It is also standard for the names to start with a capital letter and to use mixed capitals (camel case) for multi-word names.

It is essential to realize that because the example uses rdf:ID to identify classes, that their names are expanded to a full URI using the ontology's base URI. If other documents wish to refer to the same class, they must use either the full URI or a qualified name with an appropriate namespace declaration. The use of URIs to name classes (and properties) is an important aspect of the Semantic Web. Human languages frequently have polysemous terms, that is, words that have multiple meanings. For example, "bank" could mean a financial institution, the side of a river or an aerial maneuver. Assuming these different usages were defined in different ontologies, then they would each have a different URI: finance:Bank, geo:Bank, air:Bank. Thus, each of the meaning of a term would have a unique symbol. However, this solution compounds the problem of synonymy, where in general different symbols may be used with the same meaning. If the same class name is used to mean the same thing in different ontologies, then technically these classes will have different full URIs. For example, in OWL we cannot assume that your:Person and my:Person refer to the same class. However, once discovered, such problems can be easily resolved, since we can use OWL axioms to explicitly relate such classes.

If we wish to specify additional information describing the class, then we include properties from the RDFS and/or OWL vocabularies (represented by subelements in the XML syntax). The rdfs:subClassOf property can be used to relate a class to more general classes. For example, in the figure above, we state that Man is a subclass of Person. A class can also be said to have exactly the same members as another class using the owl:equivalentClass property. This is often used for synonymous classes, particularly when the classes originate in different ontologies, as discussed in the previous paragraph.

3.2 Properties

OWL can also define two types of properties: object properties and datatype properties. Object properties specify relationships between pairs of resources. Datatype properties, on the other hand, specify a relation between a resource and a data type value; they are equivalent to the notion of attributes in some formalisms. There are a number OWL and RDF terms that are used to describe properties. In Figure 2-5, we declare hasChild and hasDaughter as object properties, and we declare age as a datatype property. While classes in RDF and OWL are typically named using an initial capital letter, properties typically have an initial lower case letter. However, like class names, property names use mixed capitals in complex names.

As with classes, we describe properties by including subelements. A statement using an rdfs:subPropertyOf predicate states that every subject/object pair using the subject property is also a valid subject/object pair using the object property. In this way taxonomies of properties can be established. For example, Figure 2-5 states that the hasDaughter property is a rdfs:subPropertyOf the hasChild property. Thus we can infer that if Jack hasDaughter Sydney then Jack hasChild Sydney is also true. The owl:equivalentProperty states that the property extensions of the two properties are the same. That is, every subject/object pair for one property is a valid subject/object pair for the other. This is the property analog of owl:equivalentClass, and is frequently used to describe synonymous properties.

The rdfs:domain and rdfs:range properties are used to specify the domain and range of a property. The rdfs:domain of a property specifies that the subject of any statement using the property is a member of the class it specifies. Similarly, the rdfs:range of a property specifies that the object of any statement using the property is a member of the class or datatype it specifies. Although these properties may seem straightforward, they can lead to a number of misunderstandings and should be used carefully. See Section 6 for a discussion of some of these issues.

OWL also defines a number of constructors that specify the semantics for properties. It defines another relationship between properties using owl:inverseOf. For example, this can be used to say that the isParentOf property is the owl:inverseOf of the isChildOf property. Thus, if A is the parent of B, then B is necessarily the child of A. OWL also defines a number of property characteristics, such as owl:TransitiveProperty, owl:SymmetricProperty, owl:FunctionalProperty, and owl:InverseFunctionalProperty. Note, unlike the constructors we have seen so far, which are all properties, these are classes. That is because these four characteristic are used by making them the rdf:type of the property (note, a

property, like any resource, can have multiple types in RDF and OWL). The owl:TransitiveProperty and owl:SymmetricProperty constructors specify that the property is a transitive relation and a symmetric relation respectively. The former can be used to describe the isTallerThan property, while the latter can be used to describe the isFriendOf property. The owl:FunctionalProperty constructor states that each resource uniquely identifies its value for the property. That is, no resource can have more than one value for the property. On the other hand, the owl:InverseFunctionalProperty constructor states that each property value uniquely identifies the subject of the property. For those familiar with databases, the owl:InverseFunctionalProperty specifies a property that can be used as a primary key for the class of object that is the domain of the property. In Figure 2-5, hasSSN is both an owl:FunctionalProperty (because each person has at most one Social Security Number) and an owl:InverseFunctionalProperty (because a Social Security Number can be used to uniquely identify U.S. citizens). Note, the figure also shows the typical idiom for use of the properties described above: the property is declared as an object (or datatype) property, and then rdf:type is used to indicate additional characteristics of the property. Frequently, an XML entity reference is used, where the entity "owl" corresponds to the text "http://www.w3.org/2002/07/owl#".

3.3 Instances

In addition to expressing the semantics of classes and properties, OWL can be used to relate instances. As shown in Figure 2-6**,** the owl:sameAs property is used to state that two instances are identical. This is particularly useful in distributed settings such as the Web, where different entities may use different identifiers to refer to the same things. For example, multiple URLs may refer to the same person. A person may have different URLs for their personal and work web pages, or possibly even multiple work web pages as they change jobs over time.

```
<p:Person rdf:about="http://www.cse.lehigh.edu/~heflin/ ">
   <owl:sameAs
         rdf:resource="http://www.cs.umd.edu/~heflin/" />
</p:Person>
```

Figure 2-6. Example of owl:sameAs

It is also possible to say that two instances are different individuals. The owl:differentFrom property is used to do this. It is often the case that we

want to say that a set of individuals is pairwise distinct. Figure 2-7 shows how the owl:AllDifferent constructor can be used to identify such a set. Note that the rdf:parseType="Collection" syntax will be explained in Section 4.1.

```
<owl:AllDifferent>
   <owl:distinctMembers rdf:parseType="Collection">
      <p:Person rdf:about="#Bob" />
      <p:Person rdf:about="#Sue" />
      <p:Person rdf:about="#Mary" />
      …
   </owl:distinctMembers>
</owl:AllDifferent>
```

Figure 2-7. Example of owl:AllDifferentFrom

3.4 The Sublanguages of OWL

OWL actually consists of three languages with increasing expressivity: OWL Lite, OWL DL and OWL Full. All three of these languages allow you to describe classes, properties, and instances, but the weaker languages have restrictions on what can be stated or how it may be stated. OWL Lite is intended for users with simple modeling needs. It is missing or has weakened versions of the constructs mentioned in the next section. OWL DL has the closest correspondence to an expressive description logic and includes all of the features described in this chapter. Both OWL DL and OWL Lite require that every resource either be a class, object property, datatype property or instance[2]. An important consequence of this is that a resource cannot be treated as both a class and an instance. Furthermore, the category of each resource must be explicit in all ontologies (i.e., each resource must have an rdf:type statement). That is, in OWL Lite and OWL DL we cannot use a resource as a class without also describing it as such elsewhere in the document. OWL Full has the same features as OWL DL, but loosens the restrictions. It is possible to treat a class as an instance, and there is no need to explicitly declare the type of each resource.

Wang et al. (2006) analyzed a sample 1275 ontologies on the Web, and found that 924 of them were in OWL Full. However, most of these ontologies could be automatically converted (patched) to OWL Lite or OWL DL by adding missing type statements. After such additions, only 61 OWL Full ontologies remained. Given that most ontologies do not need the extra expressivity of OWL Full, we will focus on OWL DL in the rest of this chapter.

[2] Technically, there are other categories of resources as well, but we shall ignore them here for clarity of exposition.

4. COMPLEX OWL CLASSES

In this section, we will discuss how to further describe OWL classes using more powerful constructs. Classes can be described by means of Boolean combinations, by describing restrictions on their properties, or by enumerating their members. Additionally, classes can be disjoint with other classes.

4.1 Boolean Combinations

Using the standard set operators intersection, union, and complement, it is possible to define Boolean combinations of classes. Each of these is defined in OWL using a property. Generally, the property relates a class to another class or a collection of classes. The semantics are that the class that is the subject of the property is equivalent to the corresponding set operator applied to a set of classes specified by the property's object.

In Figure 2-8 we show an example of using intersection to define the concept Father. Father is exactly the intersection of the classes Parent and Male. In other words, anyone who is a Father is both a Parent and a Male, and anyone who is both a Parent and a Male is a Father. Intersection is equivalent to conjunction in classical logic. As with the simple classes shown in the previous section, the description of the class appears as its subelement. In this case, the owl:intersectionOf element is used. The subelements of this class are the classes which when intersected define Father. In general, owl:intersectionOf defines its subject class as being exactly equivalent to the intersection of all the classes that appear in the owl:intersectionOf element. Note, the intersected classes do not have to be named classes, but could be complex class descriptions themselves.

```
<owl:Class rdf:ID=”Father”>
   <owl:intersectionOf rdf:parseType=”Collection”>
      <owl:Class rdf:about=”#Parent” />
      <owl:Class rdf:about=”#Male” />
   </owl:intersectionOf>
</owl:Class>
```

Figure 2-8. Example of owl:intersection

At first glance, it might appear that the OWL in Figure 2-8 is equivalent to saying Father rdfs:subClassOf Parent and Father rdfs:subclassOf Male. However, these two subClassOf statements only state that all fathers must be parents and male. They cannot be used to infer that someone is a father from only their gender and parenthood, as can be done using owl:intersectionOf.

We should also point out the need for the rdf:parseType=”Collection” attribute in the owl:intersectionOf element. This is a means of specifying that the object of the property is a Collection. In RDF, a Collection is a closed list. That is, not only do we know that all of the subelements are members of the list, but we also know that there are no other members of the list. This may seem like a pedantic issue, but it is actually critical to the Semantic Web. Both RDF and OWL make the open-world assumption. That is, they assume that all RDF graphs are potentially incomplete, and can be augmented with information from other sources. However, we cannot make this assumption when defining a class. If there might be other classes participating in the intersection, then we would never be able to infer members of the defined class. This would be no better than having a set of rdfs:subClassOf statements. However, by using the Collection parse type, this problem is avoided. Further implications of OWL’s open world assumption, as opposed to closed world, can be found in Section 6.

Unions in OWL are expressed in the same manner as intersections, but we use the owl:unionOf property instead of the owl:intersectionOf property. This states that the class that is the subject of the owl:unionOf property is exactly the union of the classes contained in the Collection. This could be used for example to state that Person is the union of Male and Female. In classical logic this is equivalent to disjunction.

OWL can also define a class in terms of the complement of another class. Using owl:complementOf, we can state that the members of a class are all things that are not members of some other class. In Figure 2-9, we show that a Man is a Person who is in the complement of Woman. In other words, any Persons who are not Women are Men. Unlike owl:intersectionOf and owl:unionOf, the complement is defined over a single class. Thus, there is no need for a rdf:parseType=”Collection” attribute.

```
<owl:Class rdf:ID=”Man”>
  <owl:intersectionOf rdf:parseType=”Collection”>
    <owl:Class rdf:about=”#Person”>
    <owl:Class>
      <owl:complementOf rdf:resource=”#Woman” />
    </owl:Class>
  </owl:intersectionOf>
<owl:Class>
```

Figure 2-9. Example of owl:complementOf

It is important to consider why it would be incorrect to simply say that Man is the complement of Woman. That would say that anything which is not a woman is a man, including cars, places, animals, events, etc. By intersecting with Person, we get a sensible definition. This is an important

idiom to remember: when using owl:complementOf, always include an intersection with some class to provide context for the complement. When doing so, it is important to wrap the owl:complementOf element in an owl:Class element. Since owl:complementOf is a property that defines a class, we must provide a class for it to define. In this case, the class is anonymous.

Each Boolean operator takes one or more classes as operands. These classes may be named classes, or may be complex classes formed from descriptions. In Figure 2-9, we saw an example of combining the owl:intersectionOf and owl:complementOf Boolean operators to describe a class. In general, we can have an arbitrary nesting or ordering of Boolean operators.

4.2 Property Restrictions on Classes

In addition to describing classes using Boolean operators, we can describe classes in terms of restrictions on the property values that may occur for instances of the class. These restrictions include owl:allValuesFrom, owl:someValuesFrom, owl:hasValue, owl:cardinality, owl:minCardinality, and owl:maxCardinality.

Figure 2-10 shows an example of a description using owl:allValuesFrom, which is a form of universal quantification. In this example, we state that a Band is a subset of the objects that only have Musicians as members. The syntax relies on the use of an owl:Restriction element, and this pattern is found in all other property restrictions. The owl:Restriction contains two subelements, an owl:onProperty element and an owl:allValuesFrom element. When owl:allValuesFrom is used in an owl:Restriction, it defines a class that is the set of all objects such that every value for the property specified in the owl:onProperty element is of the type specified by the class in the owl:allValuesFrom element.

```
<owl:Class rdf:ID="Band">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasMember" />
      <owl:allValuesFrom rdf:resource="#Musician" />
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

Figure 2-10. Example using owl:allValuesFrom

In the example, the class Band is defined as a subclass of the restriction. An owl:Restriction always defines a class, which is typically anonymous.

We must then relate this class to a class that we are trying to describe or define. By saying that a Band is a subclass of the restriction, we state that every member of a Band must be a Musician, but we do not say that every group of Musicians is a band. After all, a group of musicians that gets together weekly to play poker is not necessarily a band!

One important thing to note about owl:allValuesFrom is that it has weaker semantics than users often assume. In particular, if an object has no values for the property, then it satisfies the restriction. Thus, in the example above, things that do not have any members, such as a Person Robert, can potentially be a Band. This is another reason why the subClassOf is important in the example.

Just as the owl:allValuesFrom specifies a universal constraint, owl:someValuesFrom specifies an existential one. The same syntactic pattern as above is used, with an owl:Restriction element and an owl:onProperty element, but instead of owl:allValuesFrom, owl:someValuesFrom is used for the second subelement. When owl:someValuesFrom is used in an owl:Restriction, it defines a class that is the set of all objects such that at least one value for the property specified in the owl:onProperty element is of the type specified by the class in the owl:someValuesFrom element. For example, one might create a restriction with an owl:onProperty of hasMember and an owl:someValuesFrom Singer to say that a Band is a subclass of the class of things with at least one singer.

We can also use property restrictions to give a specific value for a property, as opposed to its class. The owl:hasValue element is used for this purpose. For example we could create a restriction containing an owl:onProperty with value playsInstrument and owl:hasValue with value Guitar, in order to define a class Guitarist. Like owl:someValuesFrom, owl:hasValue is existential: it says that at least one value of the property must be the specified one. However, there could be duplicate values, or there could be other values for the property as well.

The final form of property restrictions are those based on the number of values that individuals have for specified properties. These are called cardinality restrictions. Figure 2-11 uses an owl:minCardinality restriction to define the class Parent. Once again, the owl:Restriction and owl:onProperty elements are used. The owl:minCardinality element specifies the minimum number of values that instances can have for the property. In this case, we say that a Parent has at least one value for the hasChild property. Also of note here is the rdf:datatype attribute in the owl:minCardinality element. Technically, this is a required element used to correctly interpret the value given as content for the element. However, some parsers may be more forgiving than others. In this case, the “&xsd;nonNegativeInteger” is a type defined by XML Schema. Recall from Section 2.1 that “&xsd;” is an entity

reference; we assume that there is a corresponding definition to "http://www.w3.org/2001/XMLSchema#". Also note with this example that we use owl:equivalentClass to relate the restriction to the class being described. This is because the restriction specifies necessary and sufficient conditions for being a parent: anyone who is a Parent must have at least one child, and any one who has at least one child is a Parent.

```
<owl:Class rdf:ID="Parent">
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasChild" />
      <owl:minCardinality
              rdf:datatype="&xsd;nonNegativeInteger">
        1</owl:minCardinality>
    </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>
```

Figure 2-11. Example of owl:minCardinality

The two other forms of cardinality restrictions are owl:maxCardinality and owl:cardinality. Both have similar syntax to owl:minCardinality. With owl:maxCardinality, we say that members of the class have at most the specified number of values for the property. With owl:cardinality, we say that members of the class have exactly the number of specified values for the property. Thus, owl:cardinality is equivalent to having an owl:minCardinality restriction and an owl:maxCardinality restriction set to the same value.

4.3 Disjoint and Enumerated Classes

There are two other forms of complex class descriptions. One involves describing a class in terms of a class or classes that it is disjoint with. The other involves describing the class by listing its members.

We specify disjoint classes using the owl:disjointWith property. See Figure 2-12 for a simple example that defines Male and Female as disjoint classes. This means that they have no instances in common. It is important to consider the difference between this and owl:complementOf. With the latter, knowledge that someone is not Female allows you to infer that they are a Male. With owl:disjointWith we cannot make the same inference.

```
<owl:Class rdf:ID="Male">
    <owl:disjointWith rdf:resource="#Female">
</owl:Class>
```

Figure 2-12. Example of owl:disjointWith

The members of a class can be explicitly enumerated using the owl:oneOf property. Figure 2-13 shows this construct being used to define the class of primary colors: red, blue and yellow. This construct says that the members are exactly those given: no more, no less. Due to this need to describe the complete set of members, we use the rdf:parseType="Collection" syntax that we first saw with owl:intersectionOf in Section 4.1. A subelement is given for each member. Each of these members should be explicitly typed. In this case the class owl:Thing is used, but more specific classes could be used as well.

```
<owl:Class rdf:ID="PrimaryColor">
    <owl:oneOf rdf:parseType="Collection">
        <owl:Thing rdf:about="#Red" />
        <owl:Thing rdf:about="#Blue" />
        <owl:Thing rdf:about="#Yellow" />
   </owl:oneOf>
</owl:Class>
```

Figure 2-13. Example of owl:oneOf

5. DISTRIBUTED ONTOLOGIES

So far, we have discussed features of OWL that are typically found in description logics. However, being a *Web* ontology language, OWL also provides features for relating ontologies to each other. There are two situations described below: importing an ontology and creating new versions of an ontology.

5.1 Importing Ontologies

A primary goal of the Semantic Web is to describe ontologies in a way that allows them to be reused. However, it is unlikely that an ontology developed for one purpose will exactly meet the needs of a different application. Instead, it is more likely that the old ontology will need to be extended to support additional requirements. This functionality is supported by the owl:imports statement.

Figure 2-14 shows a fragment of a fictitious news ontology that imports an equally fictitious person ontology. This is conveyed via the owl:imports property used in the owl:Ontology element that serves as the header of the document. The object of the property should be a URI that identifies another ontology. In most cases, this should be a URL that can be used to retrieve the imported ontology. However, if it is possible for systems to retrieve the ontology without an explicit URL (for example, if there is a registration service), then this is not necessary.

```
<owl:Ontology rdf:about="">
  <rdfs:label>News Ontology, v. 2.0</rdfs:label>
  <owl:imports
          rdf:resource="http://example.org/onts/person" />
  <owl:backwardCompatibleWith
          rdf:resource="http://example.org/onts/news-v10" />
</owl:Ontology>
```

Figure 2-14. Example of importing and versioning

When an ontology imports another ontology, it effectively says that all semantic conditions of the imported ontology hold in the importing ontology. As a result, the imports relationship is transitive: if an ontology A imports an ontology B, and B imports an ontology C, then A also imports C. In the OWL specifications, the set of all imported ontologies, whether directly or indirectly, is called the imports closure. It is important to note that the semantics of an ontology are not changed by ontologies that import it. Thus one can be certain of the semantics of any given ontology by simply considering its imports closure.

Finally, we should note that importing is only a semantic convention and that it has no impact on syntax. In particular, importing does not change the namespaces of the document. In order to refer to classes and properties that are defined in the imported ontology (or more generally in the imports closure), then one must define the appropriate namespace prefixes and/or entities. It is not unusual to see a namespace declaration, entity declaration and an owl:imports statement for the same URI! This is an unfortunate tradeoff that was made in order to preserve OWL's compatibility with XML and RDF.

5.2 Ontology Versioning

Since an ontology is essentially a component of a software system, it is reasonable to expect that ontologies will change over time. There are many possible reasons for change: the ontology was erroneous, the domain has evolved, or there is a desire to represent the domain in a different way. In a

centralized system, it would be simple to modify the ontology. However, in a highly decentralized system, like the Web, changes can have far reaching impacts on resources beyond the control of the original ontology author. For this reason, Semantic Web ontologies should not be changed directly. Instead, when a change needs to be made, the document should be copied and given a new URL first. In order to connect this document to the original version, OWL provides a number of versioning properties.

There are two kinds of versioning relationships: owl:priorVersion and owl:backwardCompatibleWith. The former simply states that the ontology identified by the property's object is an earlier version of the current ontology. The second states that the current ontology is not only a subsequent version of the identified ontology, but that it is also backward compatible with the prior version. By backward-compatibility we mean that the new ontology can be used as a replacement for the old without having unwanted consequences on applications. In particular, it means that all terms of the old ontology are still present in the new, and that the intended meanings of these terms are the same.

Officially, owl:priorVersion and owl:backwardCompatibleWith have no formal semantics. Instead they are intended to inform ontology authors. However, Heflin and Pan (2004) have proposed a semantics for distributed ontologies that takes into accounts the interactions between backward-compatibility and imports. It is unknown whether this proposal will have impact on future versions of OWL.

Some programming languages, such as Java, allow for deprecation of components. The point of deprecation is to preserve the component for backward-compatibility, while warning users that it may be phased out in the future. OWL can deprecate both classes and properties using the owl:DeprecatedClass and owl:DeprecatedProperty classes. Typically, there should be some axioms that provide a mapping from the deprecated classes and/or properties to new classes/properties.

Finally, OWL provides two more versioning properties. The owl:versionInfo property allows the ontology to provide a versioning string that might be used by a version management system. OWL also has an owl:incompatibleWith property that is the opposite of owl:backwardCompatibleWith. It is essentially for ontology authors that want to emphasize the point that the current ontology is not compatible with a particular prior version.

6. A WARNING ABOUT OWL'S SEMANTICS

The nature of OWL's semantics is sometimes confusing to novices. Often this results from two key principles in OWL's design: OWL does not make the closed world assumption and OWL does not make the unique names assumption. The closed world assumption presumes that anything not known to be true must be false. The unique names assumption presumes that all individual names refer to distinct objects.

As a result of not assuming a closed-world, the implications of properties like rdfs:domain and rdfs:range are often misunderstood. First and foremost, in OWL these should not be treated as database constraints. Instead, any resource that appears in the subject of a statement using the property must be inferred to be a member of the domain class. Likewise, any resource that appears in the object of statement using the property must be inferred to be a member of the range class. Thus, if we knew that Randy hasChild Fido, Fido is of type Dog and the range of hasChild was Person, then we would have to conclude that Fido was a Person as well as a Dog. We would need to state that the classes Dog and Person were disjoint in order to raise any suspicion that there might be something wrong with the data. Doing so would result in a logical contradiction, but whether the error is that Fido is not a Dog or that the hasChild property should be generalized to apply to all animals and not just people would have to be determined by a domain expert. Similarly, if a class has an owl:minCardinality restriction of 1 on some property, then that does not mean that instances of that class must include a triple for the given property. Instead, if no such triples are found, then we can infer that there is such a relationship to some yet unknown object.

OWL's open world assumption also impacts the semantics of properties that have multiple domains or multiple ranges. In each case, the domain (or range) is effectively the intersection of all such classes. This may seem counterintuitive, since if we create multiple range statements, we probably mean to say that the range is one of the classes. However, such union semantics do not work in an open world. We would never know if there is another rdfs:range statement on another Web page that will widen the property's range, and thus the statement would have no inferential power. The intersection semantics used by OWL guarantee that we can infer that the object in question is of the type specified by the range, regardless of whatever other rdfs:range statements exist.

Since OWL does not make the unique names assumption, there are some interesting issues that occur when reasoning about cardinalities. Consider the example in Figure 2-15 where we assume that p:Person has a maxCardinality restriction of 1 on the property p:hasMom. If you are new to OWL, you might think this is a violation of the property restriction. However, since

there is no assumption that Sue and Mary must be different people, this in fact leads us to infer Sue owl:sameAs Mary. If this was undesirable, we would have to also state Sue owl:differentFrom Mary, which then results in a logical contradiction.

```
<p:Person rdf:about="#Bob">
   <p:hasMom rdf:resource="#Sue" />
   <p:hasMom rdf:resource="#Mary" />
</p:Person>
```

Figure 2-15. Example demonstrating use of cardinality without the unique names assumption. If p:Person has a maxCardinality restriction of 1 on the property p:hasMom, then we infer Sue owl:sameAs Mary.

There are a number of other common mistakes made by beginning OWL ontologists. A good discussion of these issues can be found in Rector et al. (2004).

7. CONCLUSION

This chapter has provided an overview of the OWL language, with a focus on OWL DL. We have discussed the fundamentals of XML and RDF, and how they relate to OWL. We described how to create very simple OWL ontologies consisting of classes, properties and instances. We then considered how more complex OWL axioms could be stated. This was followed by a discussion of how OWL enables distributed ontologies, with a special focus on imports and versioning. Finally, we discussed some of the features of OWL's semantics that tend to lead to modeling mistakes by beginning OWL ontologists.

ACKNOWLEDGEMENTS

I would like to thank Mike Dean, Zhengxiang Pan and Abir Qasem for helpful comments on drafts of this chapter. The authorship of this chapter was supported in part by the National Science Foundation (NSF) under Grant No. IIS-0346963.

REFERENCES

G. Antoniou and F. van Harmelen. A Semantic Web Primer. MIT Press, Cambridge, MA, 2004.

F. Baader, D. Calvanese, D.L. McGuinness, D. Nardi, P.F. Patel-Schneider, eds. The Description Logic Handbook, Cambridge University Press, 2002.

P. Biron and A. Malhotra, eds.. XML Schema Part 2: Datatypes Second Edition. W3C Recommendation, 28 October 2004, http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/ .

T. Gruber. A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, vol. 5 no. 2, 1993, pp. 199-220.

N. Guarino, Formal Ontology and Information Systems. In *Proceedings of Formal Ontology and Information Systems*, Trento, Italy. IOS Press, 1998.

J. Heflin and Z. Pan. A Model Theoretic Semantics for Ontology Versioning. Third International Semantic Web Conference (ISWC 2004). LNCS 3298, Springer, 2004. pp. 62-76.

I. Horrocks and P. Patel-Schneider. Reducing OWL Entailment to Description Logic Satisfiability. In Dieter Fensel, Katia Sycara, and John Mylopoulos, editors, *Proc. of the 2003 International Semantic Web Conference (ISWC 2003)*, Lecture Notes in Computer Science 2870, pp. 17-29. Springer, 2003.

N. Noy and C. Hafner. The State of the Art in Ontology Design. *AI Magazine*, vol. 18, no. 3, 1997, pp. 53-74.

A. Rector, N. Drummond, M. Horridge, J. Rogers, H. Knublauch, R. Stevens, H. Wang, and C. Wroe. OWL Pizzas: Practical Experience of Teaching OWL-DL: Common Errors & Common Patterns. *14th International Conference on Knowledge Engineering and Knowledge Management (EKAW 2004)*, pp. 63-81. 2004.

M. Smith, C. Welty, and D. McGuinness, eds. OWL Web Ontology Language Guide, W3C Recommendation, 10 February 2004, http://www.w3.org/TR/2004/REC-owl-guide-20040210/ .

T. Wang, B. Parsia, and J. Hendler. A Survey of the Web Ontology Landscape. Fifth International Semantic Web Conference (ISWC 2006), LNCS 4273, Springer, 2006. pp. 682-694.