OWL 2 Web Ontology Language Primer

W3C Working Draft 11 June 2009

This version:
http://www.w3.org/TR/2009/WD-owl2-primer-20090611/
Latest version:
http://www.w3.org/TR/owl2-primer/
Previous version:
http://www.w3.org/TR/2009/WD-owl2-primer-20090421/ (color-coded diff)

Editors:
Pascal Hitzler, University of Karlsruhe
Markus Krötzsch, University of Karlsruhe
Bijan Parsia, University of Manchester
Peter F. Patel-Schneider, Bell Labs Research, Alcatel-Lucent
Sebastian Rudolph, University of Karlsruhe

This document is also available in these non-normative formats: PDF version.

Abstract

The OWL 2 Web Ontology Language, informally OWL 2, is an ontology language for the Semantic Web with formally defined meaning. OWL 2 ontologies provide classes, properties, individuals, and data values and are stored as Semantic Web documents. OWL 2 ontologies can be used along with information written in RDF, and OWL 2 ontologies themselves are primarily exchanged as RDF documents. The OWL 2 Document Overview describes the overall state of OWL 2, and should be read before other OWL 2 documents.
This primer provides an approachable introduction to OWL 2, including orientation for those coming from other disciplines, a running example showing how OWL 2 can be used to represent first simple information and then more complex information, how OWL 2 manages ontologies, and finally the distinctions between the various sublanguages of OWL 2.

Status of this Document

May Be Superseded

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at http://www.w3.org/TR/.

Summary of Changes

The rewrite and editing of this Working Draft has been completed since the previous version of 21st April, 2009.

Last Call

The Working Group believes this document is now essentially done, so this is a "Last Call" draft. The document is not expected to change significantly, going forward.

Please Comment By 30 July 2009

The OWL Working Group seeks public feedback on this Working Draft. Please send your comments to public-owl-comments@w3.org (public archive). If possible, please offer specific changes to the text that would address your concern. You may also wish to check the Wiki Version of this document and see if the relevant text has already been updated.

No Endorsement

Publication as a Working Draft does not imply endorsement by the W3C Membership. This is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite this document as other than work in progress.

Patents

This document was produced by a group operating under the 5 February 2004 W3C Patent Policy. This document is informative only. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

Table of Contents

1 Introduction

The W3C OWL 2 Web Ontology Language (OWL) is a Semantic Web language designed to represent rich and complex knowledge about things, groups of things, and relations between things. OWL is a computational logic-based language such that knowledge expressed in OWL can be reasoned with by computer programs either to verify the consistency of that knowledge or to make implicit knowledge explicit. OWL documents, known as ontologies, can be published in the World Wide Web and may refer to or be referred from other OWL ontologies. OWL is part of the W3C's Semantic Web technology stack, which includes RDF RDF/XML [*RDF*] and SPARQL [*RDF*].

The key goal of the primer is to help develop insight into OWL, its strengths, and its weaknesses. The core of the primer is an introduction to most of the language features of OWL by way of a running example. Most of the examples in the primer are taken from a sample ontology (which is presented entirely in an appendix). This sample ontology is designed to touch the key language features of OWL in an understandable way and not, in itself, to be an example of a good ontology.

1.1 Guide to this Document

This document is intended to provide an initial understanding about OWL 2. In particular it is supposed to be accessible for people yet unfamiliar with the topic. Therefore, we start with giving some high-level introduction on the nature of OWL 2 in Section 2 before Section 3 provides some very basic notions in knowledge representation and explains how they relate to terms used in OWL 2. Readers familiar with knowledge representation and reasoning might only skim through this section to get acquainted with the OWL 2 terminology.

Sections 4-8 describe most of the language features that OWL provides, starting from very basic ones and proceeding to the more sophisticated. Section 4 presents and discusses the elementary modeling features of OWL 2 before in Section 5 complex classes are introduced. Section 6 addresses advanced modeling features for properties. Section 7 focuses on advanced modeling related to datatypes. Section 8 concludes with extra-logical features used mainly for ontology management purposes.

In Section 9 we address the differences between OWL 2 DL and OWL 2 Full, the two semantic views of OWL, while in Section 10 we describe the three tractable sublanguages of OWL 2, called profiles. Tool support for OWL 2 is addressed in Section 11 and in Section 12 we give pointers on where to continue reading after our informal introduction to OWL 2.

Finally, Section 13 lists the complete example ontology used in this document.

For a comprehensive listing of the OWL 2 language features, see the [*OWL 2 Quick Guide*] which provides links into the corresponding sections of the appropriate documents concerning syntax and examples.

For readers already familiar with OWL 1, [*OWL 2 New Features and Rationale*] provides a comprehensive overview of what has changed in OWL 2.

1.2 OWL Syntaxes

OWL is a language to be used in the Semantic Web, so names in OWL are international resource identifiers (IRIs) [RFC-3987]. As IRIs are long, we will often make use of abbreviation mechanisms for writing them in OWL. The way in which such abbreviations work is specific to each syntactic format that can be used to encode OWL ontologies, but the examples in this document can generally be understood without knowing these details. Appropriate declarations for namespaces and related mechanisms are given further below in Section 8.2.

There are various syntaxes available for OWL which serve various purposes. The Functional-Style syntax [*OWL 2 Specification*] is designed to be easier for specification purposes and to provide a foundation for the implementation of OWL 2 tools such as APIs and reasoners. The RDF/XML syntax for OWL is just RDF/XML, with a particular translation for the OWL constructs [*OWL 2 RDF Mapping*]. This is the only syntax that is mandatory to be supported by all OWL 2 tools. The Manchester syntax [*OWL 2 Manchester Syntax*] is an OWL syntax that is designed to be easier for non-logicians to read. The OWL XML syntax is an XML syntax for OWL defined by an XML schema [*OWL 2 XML Serialization*]. There are tools that can translate between the different syntaxes for OWL. In many syntactic forms, OWL language constructs are also represented by IRIs, and some declarations might be needed to use the abbreviated forms as in the examples. Again, necessary details are found in Section 8.2.

The examples and the sample ontology in the appendix can be viewed as any of the four different syntaxes, and we provide both RDF/XML [*RDF/XML Syntax*] and Turtle [*RDF Turtle Syntax*] serializations for the RDF-based syntax. You can control which syntaxes are shown throughout the document by using the buttons below.

The buttons below can be used to show or hide the available syntaxes.

Hide Functional Syntax | Show Functional Syntax | Hide RDF/XML Syntax | Show RDF/XML Syntax | Hide Turtle Syntax | Show Turtle Syntax | Hide Manchester Syntax | Show Manchester Syntax | Hide OWL/XML Syntax | Show OWL/XML Syntax

2 What is OWL 2?

OWL 2 is a language for expressing *ontologies*. The term *ontology* has a complex history both in and out of computer science, but we use it to mean a certain kind of computational artifact – i.e., something akin to a program, an XML schema, or a web page – generally presented as a document. An ontology is a set of precise descriptive statements about some part of the world (usually referred to as the *domain of interest* or the *subject matter* of the ontology). Precise descriptions satisfy several purposes: most notably, they prevent misunderstandings in human communication and they ensure that software behaves in a uniform, predictable way and works well with other software.

In order to precisely describe a domain of interest, it is helpful to come up with a set of central terms – often called vocabulary – and fix their meaning. Besides a concise natural language definition, the meaning of a term can be characterized by stating how this term is interrelated to the other terms. A *terminology*, providing a vocabulary together with such interrelation information constitutes an essential part of a typical OWL 2 document. Besides this terminological knowledge, an ontology might also contain so called assertional knowledge that deals with concrete objects of the considered domain rather than general notions.

OWL 2 is not a programming language: OWL 2 is *declarative*, i.e. it describes a state of affairs in a logical way. Appropriate tools (so-called reasoners) can then be used to infer further information about that state of affairs. How these inferences are realized algorithmically is not part of the OWL document but depends on the specific implementations. Still, the correct answer to any such question is predetermined by the formal semantics (which comes in two versions: the *direct semantics* [OWL 2 Direct Semantics] and the *RDF-based semantics* [*OWL 2 RDF-Based Semantics*]). Only implementations that comply with these semantics will be regarded as OWL 2 conformant (see [*OWL 2 Conformance*]). Through its declarative nature, the activity of creating OWL 2 documents is conceptually different from programming. Still, as in both cases complex formal documents are created, certain notions from software engineering can be transferred to ontology engineering, such as methodological and collaborative aspects, modularization, patterns, etc.

OWL 2 is not a schema language for syntax conformance. Unlike XML, OWL 2 does not provide elaborate means to prescribe how a document should be structured syntactically. In particular, there is no way to enforce that a certain piece of information (like the social security number of a person) has to be syntactically present. This should be kept in mind as OWL has some features that a user might misinterpret this way.

OWL 2 is not a database framework. Admittedly, OWL 2 documents store information and so do databases. Moreover a certain analogy between assertional information and database content as well as terminological information and database schemata can be drawn. However, usually there are crucial differences in the underlying assumptions (technically: the used semantics). If some fact is not present in a database, it is usually considered false (the so-called *closed-world assumption*) whereas in the case of an OWL 2 document it may simply be missing (but possibly true), following the *open-world assumption*. Moreover, database schemata often come with the prescriptive constraint semantics mentioned above. Still, technically, databases provide a viable backbone in many ontology-oriented systems.

3 Modeling Knowledge: Basic Notions

OWL 2 is a knowledge representation language, designed to formulate, exchange and reason with knowledge about a domain of interest. Some fundamental notions should first be explained to understand how knowledge is represented in OWL 2. These basic notions are:

- **Axioms:** the basic statements that an OWL ontology expresses
- **Entities:** elements used to refer to real-world objects

- **Expressions:** combinations of entities to form complex descriptions from basic ones

While OWL 2 aims to capture knowledge, the kind of "knowledge" that can be represented by OWL does of course not reflect all aspects of human knowledge. OWL can be considered as a powerful general-purpose modeling language for certain parts of human knowledge. The results of the modeling processes are called *ontologies* – a terminology that also helps to avoid confusion since the term "model" is often used in a rather different sense in knowledge representation.

Now, in order to formulate knowledge explicitly, it is useful to assume that it consists of elementary pieces that are often referred to as *statements* or *propositions*. Statements like "it is raining" or "every man is mortal" are typical examples for such basic propositions. Indeed, every OWL 2 ontology is essentially just a collection of such basic "pieces of knowledge." Statements that are made in an ontology are called *axioms* in OWL 2, and the ontology asserts that its axioms are true. In general, OWL statements might be either true or false given a certain state of affairs. This distinguishes them from *entities* and *expressions* as described further below.

When humans think, they draw consequences from their knowledge. An important feature of OWL is that it captures this aspect of human intelligence for the forms of knowledge that it can represent. But what does it mean, generally speaking, that a statement is a consequence of other statements? Essentially it means that this statement is true whenever the other statements are. In OWL terms: we say, a set of statements *A entails* a statement *a* if in any state of affairs wherein all statements from *A* are true, also *a* is true. Moreover, a set of statements may be *consistent* (that is, there is a possible state of affairs in which all the statements in the set are jointly true) or *inconsistent* (there is no such state of affairs). The formal semantics of OWL specifies, in essence, for which possible "states of affairs" a particular set of OWL statements is true.

There are OWL tools – reasoners – that can automatically compute consequences. The way ontological axioms interact can be very subtle and difficult for people to understand. This is both a strength and a weakness of OWL 2. It is a strength because OWL 2 tools can discover information that a person would not have spotted. This allows knowledge engineers to model more directly and the system to provide useful feedback and critique of the modeling. It is a weakness because it is comparatively difficult for humans to immediately foresee the actual effect of various constructs in various combinations. Tool support ameliorates the situation but successful knowledge engineering often still requires some amount of training and experience.

Having a closer look at statements in OWL, we see that they are rarely "monolithic" but more often have some internal structure that can be explicitly represented. They normally refer to objects of the world and describe them e.g. by putting them into categories (like "Mary is female") or saying something about their relation ("John and Mary are married"). All atomic constituents of statements, be they objects (John, Mary), categories (female) or relations (married) are called *entities*. In OWL 2, we denote objects as *individuals*, categories as *classes* and relations as *properties*. Properties in OWL 2 are further subdivided. *Object properties* relate objects to objects (like a person to their spouse), while *datatype properties* assign data values to objects (like an age to a person). *Annotation properties* are used to encode information about (parts of) the ontology itself (like the author and creation date of an axiom) instead of

the domain of interest.

As a central feature of OWL, names of entities can be combined into *expressions* using so called *constructors*. As a basic example, the atomic classes “female” and “professor” could be combined conjunctively to describe the class of female professors. The latter would be described by an OWL class expression, that could be used in statements or in other expressions. In this sense, expressions can be seen as new entities which are defined by their structure. In OWL, the constructors for each sort of entity vary greatly. The expression language for classes is very rich and sophisticated, whereas the expression language for properties is much less so. These differences have historical as well as technical reasons.

4 Classes, Properties, and Individuals – And Basic Modeling With Them

After these general considerations, we now engage in the details of modeling with OWL 2. In the subsequent sections, we introduce the essential modeling features that OWL 2 offers, provide examples and give some general comments on how to use them. We proceed from basic features, which are essentially available in any modeling language, to more advanced constructs.

Thereby we will represent information about a particular family. Note that we do not intend this example to be representative of the sorts of domains OWL should be used for, or as a canonical example of good modeling with OWL, or a correct representation of the rather complex, shifting, and culturally dependent domain of families. Instead, we intend it to be a rather simple exhibition of various features of OWL.

4.1 Classes and Instances

We start by introducing the persons we are talking about. This can be done as follows:

Functional Style Syntax

```
ClassAssertion(:Person:Mary )
```

RDF/XML Syntax

```
<Person rdf:about="Mary"/>
```

Turtle Syntax

```
:Mary rdf:type:Person .
```

Manchester Syntax

```
Individual: Mary
  Types: Person
```

OWL/XML Syntax

```
<ClassAssertion>
   <Class IRI="Person"/>
   <NamedIndividual IRI="Mary"/>
</ClassAssertion>
```

This statement talks about an individual named Mary and states that this individual is a person. More technically, *being a person* is expressed by stating that Mary belongs to (or “is a member of” or, even more technically, “is an instance of”) the *class* of all persons. In general classes are used to group individuals that have something in common in order to refer to them. Hence, classes essentially represent sets of individuals. In modeling, classes are often used to denote the set of objects comprised by a concept of human thinking, like the concept *person* or the concept *woman*. Consequently, we can use the same type of statement to indicate that Mary is a woman by expressing that she is an instance of the class of women:

Functional Style Syntax

```
ClassAssertion(:Woman:Mary )
```

RDF/XML Syntax

```
<Woman rdf:about="Mary"/>
```

Turtle Syntax

```
:Mary rdf:type:Woman .
```

Manchester Syntax

```
Individual: Mary
  Types: Woman
```

OWL/XML Syntax

```
<ClassAssertion>
   <Class IRI="Woman"/>
   <NamedIndividual IRI="Mary"/>
</ClassAssertion>
```

Hereby it also becomes clear that class membership is not exclusive: as there may be diverse criteria to group individuals (like gender, age, shoe size, etc.), one individual may well belong to several classes simultaneously.

4.2 Class Hierarchies

In the previous section, we were talking about two classes: the class of all persons and that of all women. To the human reader it is clear that these two classes are in a special relationship: Person is more general than Woman, meaning that whenever we know some individual to be a woman, that individual must be a person. However, this correspondence cannot be derived from the labels “Person” and “Woman” but is part of the human background knowledge about the world and our usage of those terms.

Therefore, in order to enable a system to draw the desired conclusions, it has to be informed about this correspondence. In OWL 2, this is done by a so-called subclass axiom:

Functional Style Syntax

```
SubClassOf(:Woman:Person )
```

RDF/XML Syntax

```
<owl:Class rdf:about="Woman">
  <rdfs:subClassOf rdf:resource="Person"/>
</owl:Class>
```

Turtle Syntax

```
:Woman rdfs:subClassOf:Person .
```

Manchester Syntax

```
Class: Woman
  SubClassOf: Person
```

OWL/XML Syntax

```
<SubClassOf>
  <Class IRI="Woman"/>
  <Class IRI="Person"/>
</SubClassOf>
```

The presence of this axiom in an ontology enables reasoners to infer for every individual which is specified as an instance of the class Woman, that it is an instance of the class Person as well. As a rule of thumb, a subclass relationship between two classes A and B can be specified, if the phrase “every A is a B” makes sense and is correct.

It is common in ontological modeling to use subclass statements not only for sporadically declaring such interdependencies, but to model whole *class hierarchies* by specifying the generalization relationships of all classes in the domain of interest. Suppose we also want to state that all mothers are women:

Functional Style Syntax

```
SubClassOf(:Mother:Woman )
```

RDF/XML Syntax

```
<owl:Class rdf:about="Mother">
  <rdfs:subClassOf rdf:resource="Woman"/>
</owl:Class>
```

Turtle Syntax

```
:Mother rdfs:subClassOf:Woman .
```

Manchester Syntax

```
Class: Mother
  SubClassOf: Woman
```

OWL/XML Syntax

```
<SubClassOf>
  <Class IRI="Mother"/>
  <Class IRI="Woman"/>
</SubClassOf>
```

Then a reasoner could not only derive for every single individual that is classified as mother, that it is also a woman (and consequently a person), but also that Mother must be a subclass of Person – coinciding with our intuition. Technically, this means that the subclass relationship between classes is *transitive*. Besides this, it is also *reflexive*, meaning that every class is its own subclass – this is intuitive as well since clearly, every person is a person etc.

Classes in our vocabulary may effectively refer to the same sets, and OWL provides a mechanism by which to they are considered to be semantically equivalent. For example, we use the term Person and Human interchangeably, meaning that every instance of the class Person is also an instance of class Human, and vice versa. Two classes are considered equivalent if they contain exactly the same individuals. The following example states that the class Person is equivalent to the class Human.

Functional Style Syntax

```
EquivalentClasses(:Person:Human )
```

RDF/XML Syntax

```
<owl:Class rdf:about="Person">
  <owl:equivalentClass rdf:resource="Human"/>
</owl:Class>
```

Turtle Syntax

```
:Person owl:equivalentClass:Human .
```

Manchester Syntax

```
Class: Person
  EquivalentTo: Human
```

OWL/XML Syntax

```
<EquivalentClasses>
  <Class IRI="Person"/>
  <Class IRI="Human"/>
</EquivalentClasses>
```

Stating that Person and Human are equivalent amounts exactly to the same as stating that both Person is a subclass of Human and Human is a subclass of Person.

4.3 Class Disjointness

In Section 4.1, we stated that an individual can be an instance of several classes. However, in some cases membership in one class specifically excludes membership in another. For example, considering the classes Man and Woman, it can be excluded that there is an individual that is an instance of both classes (for the sake of the example, we disregard biological borderline cases). This “incompatibility relationship” between classes is referred to as *(class) disjointness*. Again, the information that two classes are disjoint is part of our background knowledge and has to be explicitly stated for a reasoning system to make use of it. This is done as follows:

Functional Style Syntax

```
DisjointClasses(:Woman:Man )
```

RDF/XML Syntax

```
<owl:AllDisjointClasses>
  <owl:members rdf:parseType="Collection">
    <owl:Class rdf:about="Woman"/>
    <owl:Class rdf:about="Man"/>
  </owl:members>
</owl:AllDisjointClasses>
```

Turtle Syntax

```
[]  rdf:type     owl:AllDisjointClasses;
    owl:members  (:Woman :Man ) .
```

Manchester Syntax

```
DisjointClasses: Woman Man
```

OWL/XML Syntax

```
<DisjointClasses>
    <Class IRI="Woman"/>
    <Class IRI="Man"/>
</DisjointClasses>
```

In practice, disjointness statements are often forgotten or neglected. The arguable reason for this could be that intuitively, classes are considered disjoint unless there is other evidence. By omitting disjointness statements, many potentially useful consequences can get lost. Note that in our example, the disjointness axiom is needed to deduce that Mary is not a man. Moreover, given the above axioms, a reasoner can infer the disjointness of the classes Mother and Man.

4.4 Object Properties

In the preceding sections we were concerned with describing single individuals, their class memberships, and how classes can relate to each other based on their instances. But more often than not, an ontology is also meant to specify how the individuals relate to other individuals. These relationships are central when describing a family. We start by indicating that Mary is John's wife.

Functional Style Syntax

```
ObjectPropertyAssertion(:hasWife:John:Mary )
```

RDF/XML Syntax

```
<rdf:Description rdf:about="John">
  <hasWife rdf:resource="Mary"/>
</rdf:Description>
```

Turtle Syntax

```
:John:hasWife:Mary .
```

Manchester Syntax

```
Individual: John
  Facts: hasWife Mary
```

OWL/XML Syntax

```
<ObjectPropertyAssertion>
  <ObjectProperty IRI="hasWife"/>
  <NamedIndividual IRI="John"/>
  <NamedIndividual IRI="Mary"/>
</ObjectPropertyAssertion>
```

Hereby, the entities describing in which way the individuals are related – like hasWife in our case, are called *properties.*

Note that the order in which the individuals are written is important. While “Mary is John's wife” might be true, “John is Mary's wife” certainly isn't. Indeed, this is a common source of modeling errors that can be avoided by using property names which allow only one unique intuitive reading. In case of nouns (like “wife”), such unambiguous names might be constructions with “of” or with “has” (wifeOf or hasWife). For verbs (like “to love”) an inflected form (loves) or a passive version with “by” (lovedBy) would prevent unintended readings.

We can also state that two individuals are *not* connected by a property. The following, for example, states that Mary is not Bill's wife.

Functional Style Syntax

```
NegativeObjectPropertyAssertion(:hasWife:Bill:Mary )
```

RDF/XML Syntax

```
<owl:NegativePropertyAssertion>
  <owl:sourceIndividual rdf:about="Bill"/>
  <owl:assertionProperty rdf:about="hasWife"/>
  <owl:targetIndividual rdf:about="Mary"/>
</owl:NegativePropertyAssertion>
```

Turtle Syntax

```
[]  rdf:type                owl:NegativePropertyAssertion;
    owl:sourceIndividual  :Bill;
    owl:assertionProperty :hasWife;
    owl:targetIndividual  :Mary .
```

Manchester Syntax

```
Individual: Bill
  Facts: not hasWife Mary
```

OWL/XML Syntax

```
<NegativeObjectPropertyAssertion>
  <ObjectProperty IRI="hasWife"/>
  <NamedIndividual IRI="Bill"/>
  <NamedIndividual IRI="Mary"/>
</NegativeObjectPropertyAssertion>
```

Negative property assertions provide a unique opportunity to make statements where we know something that is not true. This kind of information is particularly important in OWL where the default stance is that anything is possible until you say otherwise.

4.5 Property Hierarchies

In Section 4.2 we argued that it is useful to specify that one class membership implies another one. Essentially the same situation can occur for properties: whenever B is known to be A's wife, it is also known to be A's spouse (note, that this is not true the other way round). OWL allows to specify this statement as follows:

Functional Style Syntax

```
SubObjectPropertyOf(:hasWife:hasSpouse )
```

RDF/XML Syntax

```
<owl:ObjectProperty rdf:about="hasWife">
  <rdfs:subPropertyOf rdf:resource="hasSpouse"/>
</owl:ObjectProperty>
```

Turtle Syntax

```
:hasWife rdfs:subPropertyOf:hasSpouse .
```

Manchester Syntax

```
ObjectProperty: hasWife
  SubPropertyOf: hasSpouse
```

OWL/XML Syntax

```
<SubObjectPropertyOf>
  <ObjectProperty IRI="hasWife"/>
  <ObjectProperty IRI="hasSpouse"/>
</SubObjectPropertyOf>
```

There is also a syntactic shortcut for property equivalence, which is similar to class equivalence.

4.6 Domain and Range Restrictions

Frequently, the information that two individuals are interconnected by a certain property allows to draw further conclusions about the individuals themselves. In particular, one might infer class memberships. For instance, the statement that B is the wife of A obviously implies that B is a woman while A is a man. So in a way, the statement that two individuals are related via a certain property carries implicit additional information about these individuals. In our example, this additional information can be expressed via class memberships. OWL provides a way to state this correspondence:

Functional Style Syntax

```
ObjectPropertyDomain(:hasWife:Man )
ObjectPropertyRange(:hasWife:Woman )
```

RDF/XML Syntax

```
<owl:ObjectProperty rdf:about="hasWife">
  <rdfs:domain rdf:resource="Man"/>
  <rdfs:range rdf:resource="Woman"/>
</owl:ObjectProperty>
```

Turtle Syntax

```
:hasWife rdfs:domain:Man;
          rdfs:range :Woman .
```

Manchester Syntax

```
ObjectProperty: hasWife
  Domain: Man
  Range: Woman
```

OWL/XML Syntax

```
<ObjectPropertyDomain>
  <ObjectProperty IRI="hasWife"/>
```

```
  <Class IRI="Man"/>
</ObjectPropertyDomain>
<ObjectPropertyRange>
  <ObjectProperty IRI="hasWife"/>
  <Class IRI="Woman"/>
</ObjectPropertyRange>
```

Having these two axioms in place and given e.g. the information that Sasha is related to Hillary via the property hasWife, a reasoner would be able to infer that Sasha is a man and Hillary a woman.

4.7 Equality and Inequality of Individuals

Note that from the information given so far, it can be deduced that John and Mary are not the same individual as they are known to be instances of the disjoint classes Man and Woman, respectively. However, if we add information about another family member, say Bill, and indicate that he is a man, then there is nothing said so far that implies that John and Bill are not the same. OWL does not make the assumption that different names are names for different individuals. (This lack of a required “unique names assumption” is particularly well-suited to Semantic Web applications where names may be coined by different organizations at different times unknowingly referring to the same individual.) Hence, if we want to exclude the option of John and Bill being the same individual, this has to be explicitly specified as follows:

Functional Style Syntax

```
DifferentIndividuals(:John:Bill )
```

RDF/XML Syntax

```
<rdf:Description rdf:about="John">
  <owl:differentFrom rdf:resource="Bill"/>
</rdf:Description>
```

Turtle Syntax

```
:John owl:differentFrom:Bill .
```

Manchester Syntax

```
Individual: John
  DifferentFrom: Bill
```

OWL/XML Syntax

```
<DifferentIndividuals>
  <NamedIndividual IRI="John"/>
  <NamedIndividual IRI="Bill"/>
</DifferentIndividuals>
```

It is also possible to state that two names refer to (denote) the same individual. For example, we can say that John and Jack are the same individual.

Functional Style Syntax

```
SameIndividual(:James:Jim )
```

RDF/XML Syntax

```
<rdf:Description rdf:about="James">
  <owl:sameAs rdf:resource="Jim"/>
</rdf:Description>
```

Turtle Syntax

```
:James owl:sameAs:Jim.
```

Manchester Syntax

```
Individual: James
  SameAs: Jim
```

OWL/XML Syntax

```
<SameIndividual>
  <NamedIndividual IRI="James"/>
  <NamedIndividual IRI="Jim"/>
</SameIndividual>
```

This would enable a reasoner to infer that any information given about the individual James also holds for the individual Jim.

4.8 Datatypes

So far, we have seen how we can describe individuals via class memberships and via their relatedness to other individuals. In many cases, however, individuals are to be described by data values. Think of a person's birth date, his age, his email address etc. For this purpose, OWL provides another kind of properties, so-called *Datatype properties*. These properties relate individuals to data values (instead of to other individuals). The following is an example using a datatype property. It states that John's age is 51.

Functional Style Syntax

```
DataPropertyAssertion(:hasAge:John "51"^^xsd:integer )
```

RDF/XML Syntax

```
<Person rdf:about="John">
  <hasAge rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">51</hasAge>
</Person>
```

Turtle Syntax

```
:John :hasAge  51 .
```

Manchester Syntax

```
Individual: John
  Facts: hasAge "51"^^xsd:integer
```

OWL/XML Syntax

```
<DataPropertyAssertion>
  <DataProperty IRI="hasAge"/>
  <NamedIndividual IRI="John"/>
  <Literal datatypeIRI="http://www.w3.org/2001/XMLSchema#integer">51</Literal>
</DataPropertyAssertion>
```

Likewise, we can state that Jack's age is *not* 53.

Functional Style Syntax

```
NegativeDataPropertyAssertion(:hasAge:Jack "53"^^xsd:integer )
```

RDF/XML Syntax

```
<owl:NegativePropertyAssertion>
  <owl:sourceIndividual rdf:about="Jack"/>
  <owl:assertionProperty rdf:about="hasAge"/>
  <owl:targetValue rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">
    53
  </owl:targetValue>
</owl:NegativePropertyAssertion>
```

Turtle Syntax

```
[]  rdf:type                owl:NegativePropertyAssertion;
    owl:sourceIndividual   :Jack;
    owl:assertionProperty  :hasAge;
    owl:targetValue         53 .
```

Manchester Syntax

```
Individual: Jack
  Facts: not hasAge "53"^^xsd:integer
```

OWL/XML Syntax

```
<NegativeDataPropertyAssertion>
  <DataProperty IRI="hasAge"/>
  <NamedIndividual IRI="Jack"/>
  <Literal datatypeIRI="http://www.w3.org/2001/XMLSchema#integer">53</Literal>
</NegativeDataPropertyAssertion>
```

Domain and range can also be stated for datatype properties as it is done for object properties. In that case, however, the range will be a datatype instead of a class. The following states that the hasAge property is only used to relate persons with non-negative integers.

Functional Style Syntax

```
DataPropertyDomain(:hasAge:Person )
DataPropertyRange(:hasAge xsd:NonNegativeInteger )
```

RDF/XML Syntax

```
<owl:DatatypeProperty rdf:about="hasAge">
  <rdfs:domain rdf:resource="Person"/>
  <rdfs:range rdf:datatpye="http://www.w3.org/2001/XMLSchema#NonNegativeInteger"/>
</owl:DatatypeProperty>
```

Turtle Syntax

```
:hasAge  rdfs:domain :Person;
         rdfs:range   xsd:NonNegativeInteger .
```

Manchester Syntax

```
DatatpyeProperty: hasAge
  Domain: Person
  Range:  xsd:NonNegativeInteger
```

OWL/XML Syntax

```
<DatatypePropertyDomain>
  <DatatypeProperty IRI="hasAge"/>
  <Class IRI="Person"/>
</DatatypePropertyDomain>
<DatatypePropertyRange>
  <DatatypeProperty IRI="hasAge"/>
  <Datatype IRI="http://www.w3.org/2001/XMLSchema#NonNegativeInteger"/>
</DatatypePropertyRange>
```

We would like to point out at this stage a common mistake which easily occurs when using property domains and ranges. In the example just given, which states that the hasAge property is only used to relate persons with non-negative integers, assume that we also specify the information that Felix is in the class Cat and that Felix hasAge 9. From the combined information, it would then be possible to deduce that Felix is also in the class Person, which is probably not intended. This is a commonly modeling error: note that a domain (or range) statement is not a constraint on the knowledge, but allows a reasoner to infer further knowledge. If we state – as in our example – that an age is only given for persons, then everything we give an age for automatically becomes a person.

5 Advanced Class Relationships

In the previous sections we have dealt with classes as something “opaque” carrying a name. We used them to characterize individuals, and related them to other classes via subclass or disjointness statements.

We will now demonstrate how named classes, properties, and individuals can be used as building blocks to define new classes.

5.1 Complex Classes

By means of the language elements described so far, simple ontologies can be modeled. In order to express more complex knowledge, OWL provides logical class constructors. In particular, OWL provides language elements for logical and, or, and not. The corresponding OWL terms are borrowed from set theory: *(class) intersection*, *union* and *complement*. These constructors combine atomic classes – i.e. classes with names – to complex classes.

The *intersection* of two classes consists of exactly those individuals which are instances of both classes. The following example states that the class Mother consists of exactly those objects which are instances of both Woman and Parent:

Functional Style Syntax

```
EquivalentClasses(
 :Mother
  ObjectIntersectionOf(:Woman:Parent )
)
```

RDF/XML Syntax

```
<owl:Class rdf:about="Mother">
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="Woman"/>
        <owl:Class rdf:about="Parent"/>
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>
```

Turtle Syntax

```
:Mother  owl:equivalentClass  [
   rdf:type            owl:Class;
   owl:intersectionOf  (:Woman:Parent )
] .
```

Manchester Syntax

```
Class: Mother
  EquivalentTo: Woman and Parent
```

OWL/XML Syntax

```
<EquivalentClasses>
  <Class IRI="Mother"/>
  <ObjectIntersectionOf>
    <Class IRI="Woman"/>
    <Class IRI="Parent"/>
  </ObjectIntersectionOf>
</EquivalentClasses>
```

An example for an inference which can be drawn from this is that all instances of the class Mother are also in the class Parent.

The *union* of two classes contains every individual which is contained in at least one of these classes. Therefore we could characterize the class of all parents as the union of the classes Mother and Father:

Functional Style Syntax

```
EquivalentClasses(
 :Parent
  ObjectUnionOf(:Mother:Father )
)
```

RDF/XML Syntax

```
<owl:Class rdf:about="Parent">
  <owl:equivalentClass>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="Mother"/>
        <owl:Class rdf:about="Father"/>
      </owl:unionOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>
```

Turtle Syntax

```
:Parent  owl:equivalentClass  [
   rdf:type     owl:Class;
   owl:unionOf  (:Mother:Father )
] .
```

Manchester Syntax

```
Class: Parent
  EquivalentTo: Mother or Father
```

OWL/XML Syntax

```
<EquivalentClasses>
  <Class IRI="Parent"/>
  <ObjectUnionOf>
    <Class IRI="Mother"/>
    <Class IRI="Father"/>
  </ObjectUnionOf>
</EquivalentClasses>
```

The *complement* of a class corresponds to logical negation: it consists of exactly those objects which are not members of the class itself. The following definition of childless persons uses the class complement and also demonstrates that class constructors can be nested:

Functional Style Syntax

```
EquivalentClasses(
 :ChildlessPerson
  ObjectIntersectionOf(
   :Person
    ObjectComplementOf(:Parent )
  )
)
```

RDF/XML Syntax

```
<owl:Class rdf:about="ChildlessPerson">
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="Person"/>
        <owl:Class>
          <owl:complementOf rdf:resource="Parent"/>
        </owl:Class>
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>
```

Turtle Syntax

```
:ChildlessPerson  owl:equivalentClass  [
   rdf:type            owl:Class;
   owl:intersectionOf  (:Person  [ owl:complementOf :Parent ] )
] .
```

Manchester Syntax

```
Class: ChildlessPerson
  EquivalentTo: Person and not Parent
```

OWL/XML Syntax

```
<EquivalentClasses>
  <Class IRI="ChildlessPerson"/>
  <ObjectIntersectionOf>
    <Class IRI="Person"/>
    <ObjectComplementOf>
      <Class IRI="Parent"/>
    </ObjectComplementOf>
  </ObjectIntersectionOf>
</EquivalentClasses>
```

All the above examples demonstrate the usage of class constructors in order to *define* new classes as combination of others. But, of course, it is also possible to use class constructors together with a subclass statement in order to indicate necessary, but not sufficient, conditions for a class. The following statement indicates that every Grandfather is both a man and a parent (whereas the converse is not necessarily true):

Functional Style Syntax

```
SubClassOf(
 :Grandfather
```

```
  ObjectIntersectionOf(:Man:Parent )
)
```

RDF/XML Syntax

```
<owl:Class rdf:about="Grandfather">
  <rdfs:subClassOf>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="Man"/>
        <owl:Class rdf:about="Parent"/>
      </owl:intersectionOf>
    </owl:Class>
  </rdfs:subClassOf>
</owl:Class>
```

Turtle Syntax

```
:Grandfather  rdfs:subClassOf  [
   rdf:type            owl:Class;
   owl:intersectionOf  (:Man :Parent )
] .
```

Manchester Syntax

```
Class: Grandfather
  SubClassOf: Man and Parent
```

OWL/XML Syntax

```
<SubClassOf>
  <Class IRI="Grandfather"/>
  <ObjectIntersectionOf>
    <Class IRI="Man"/>
    <Class IRI="Parent"/>
  </ObjectIntersectionOf>
</SubClassOf>
```

In general, complex classes can be used in every place where named classes can occur, hence also in class assertions. This is demonstrated by the following example which asserts that John is a person but not a parent.

Functional Style Syntax

```
ClassAssertion(
  ObjectIntersectionOf(
   :Person
    ObjectComplementOf(:Parent )
  )
 :John
)
```

RDF/XML Syntax

```
<rdf:Description rdf:about="John">
  <rdf:type>
```

```
    <owl:Class>
      <owl:intersectionOf  rdf:parseType="Collection">
        <owl:Class rdf:about="Person"/>
        <owl:Class>
          <owl:complementOf rdf:about="Parent"/>
        </owl:Class>
      </owl:intersectionOf>
    </owl:Class>
  </rdf:type>
</rdf:Description>
```

Turtle Syntax

```
:John  rdf:type  [
   rdf:type            owl:Class;
   owl:intersectionOf  (:Person
                         [ rdf:type          owl:Class;
                           owl:complementOf :Parent     ]
                        )
 ] .
```

Manchester Syntax

```
Individual: John
  Types: Person and not Parent
```

OWL/XML Syntax

```
<ClassAssertion>
  <ObjectIntersectionOf>
   <Class IRI="Person"/>
   <ObjectComplementOf>
     <Class IRI="Parent"/>
   </ObjectComplementOf>
  </ObjectIntersectionOf>
  <NamedIndividual IRI="John"/>
</ClassAssertion>
```

5.2 Property Restrictions

Property restrictions provide another type of logic-based constructors for complex classes. As the name suggests, property restrictions use constructors involving properties.

One property restriction called *existential quantification* defines a class as the set of all individuals that are connected via a particular property to another individual which is an instance of a certain class. This is best explained by an example, like the following which defines the class of parents as the class of individuals that are linked to a person by the hasChild property.

Functional Style Syntax

```
EquivalentClasses(
 :Parent
  ObjectSomeValuesFrom(:hasChild:Person )
```

```
)
```

RDF/XML Syntax

```
<owl:Class rdf:about="Parent">
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="hasChild"/>
      <owl:someValuesFrom rdf:resource="Person"/>
    </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>
```

Turtle Syntax

```
:Parent  owl:equivalentClass  [
   rdf:type            owl:Restriction;
   owl:onProperty     :hasChild;
   owl:someValuesFrom :Person
] .
```

Manchester Syntax

```
Class: Parent
  EquivalentTo: hasChild some Person
```

OWL/XML Syntax

```
<EquivalentClasses>
  <Class IRI="Parent"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="hasChild"/>
    <Class IRI="Person"/>
  </ObjectSomeValuesFrom>
</EquivalentClasses>
```

This means that there is an expectation that for every instance of Parent, there exists at least one child, and that child is a member of the class Person. This is useful to capture *incomplete* knowledge*. For example, Sally tells us that Bob is a parent,* and therefore we can infer that he has at least one child even if we don't know their name.

Another property restriction, called *universal quantification* is used to describe a class of individuals for which all related individuals must be instances of a given class. We can use the following statement to indicate that somebody is a happy person exactly if all their children are happy persons.

Functional Style Syntax

```
EquivalentClasses(
 :HappyPerson
  ObjectAllValuesFrom(:hasChild:HappyPerson )
)
```

RDF/XML Syntax

```
<owl:Class>
  <owl:Class rdf:about="HappyPerson"/>
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty rdf:about="hasChild"/>
      <owl:allValuesFrom rdf:about="HappyPerson"/>
    </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>
```

Turtle Syntax

```
[]  rdf:type             owl:Class;
   :HappyPerson;
    owl:equivalentClass  [
      rdf:type           owl:Restriction;
      owl:onProperty    :hasChild;
      owl:allValuesFrom :Happy
    ] .
```

Manchester Syntax

```
Class: HappyPerson
  EquivalentTo: hasChild only HappyPerson
```

OWL/XML Syntax

```
<EquivalentClasses>
  <Class IRI="HappyPerson"/>
  <ObjectAllValuesFrom>
    <ObjectProperty IRI="hasChild"/>
    <Class IRI="Happy"/>
  </ObjectAllValuesFrom>
</EquivalentClasses>
```

This example also shows that OWL statements are allowed to by kind of self-referential; the class HappyPerson is used on both sides of the equivalence statement.

The usage of property restrictions may cause some conceptual confusion to “modeling beginners.” As a rule of thumb, when translating a natural language statement into a logical axiom, existential quantification occurs far more frequently. Natural language indicators for the usage of universal quantification are words like “only,” “exclusively,” or “nothing but.”

There is one particular misconception concerning the universal role restriction. As an example, consider the above happiness axiom. The intuitive reading suggests that in order to be happy, a person must have at least one happy child. Yet, this is not the case: any individual that is not a “starting point” of the property hasChild is class member of any class defined by universal quantification over hasChild. Hence, by our above statement, every childless person would be qualified as happy. In order to formalize the aforementioned intended reading, the statement would have to read as follows:

Functional Style Syntax

```
EquivalentClasses(
 :HappyPerson
  ObjectIntersectionOf(
    ObjectAllValuesFrom(:hasChild:HappyPerson )
    ObjectSomeValuesFrom(:hasChild:HappyPerson )
  )
)
```

RDF/XML Syntax

```
<owl:Class>
  <owl:Class rdf:about="HappyPerson"/>
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf parseType="Collection">
        <owl:Restriction>
          <owl:onProperty rdf:about="hasChild"/>
          <owl:allValuesFrom rdf:about="HappyPerson"/>
        </owl:Restriction>
        <owl:Restriction>
          <owl:onProperty rdf:about="hasChild"/>
          <owl:someValuesFrom rdf:about="HappyPerson"/>
        </owl:Restriction>
       </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>
```

Turtle Syntax

```
[]  rdf:type             owl:Class;
   :HappyPerson;
    owl:equivalentClass  [
      rdf:type            owl:Class;
      owl:intersectionOf  ( [ rdf:type            owl:Restriction;
                              owl:onProperty     :hasChild;
                              owl:allValuesFrom  :Happy            ]
                            [ rdf:type            owl:Restriction;
                              owl:onProperty     :hasChild;
                              owl:someValuesFrom :Happy            ]
                          )
    ] .
```

Manchester Syntax

```
Class: HappyPerson
  EquivalentTo: hasChild only HappyPerson and hasChild some HappyPerson
```

OWL/XML Syntax

```
<EquivalentClasses>
  <Class IRI="HappyPerson"/>
  <ObjectIntersectionOf>
    <ObjectAllValuesFrom>
      <ObjectProperty IRI="hasChild"/>
      <Class IRI="HappyPerson"/>
    </ObjectAllValuesFrom>
    <ObjectSomeValuesFrom>
      <ObjectProperty IRI="hasChild"/>
```

```
      <Class IRI="HappyPerson"/>
    </ObjectSomeValuesFrom>
  </ObjectIntersectionOf>
</EquivalentClasses>
```

This example also illustrates how property restrictions can be nested with complex classes.

Property restrictins can also be used to describe classes of individuals that are related to one particular individual. For instance we could define the class of John's children:

Functional Style Syntax

```
EquivalentClasses(
 :JohnsChildren
  ObjectHasValue(:hasParent:John )
)
```

RDF/XML Syntax

```
<owl:Class rdf:about="JohnsChildren">
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="hasParent"/>
      <owl:hasValue rdf:resource="John"/>
    </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>
```

Turtle Syntax

```
:JohnsChildren  owl:equivalentClass  [
   rdf:type        owl:Restriction;
   owl:onProperty :hasParent;
   owl:hasValue   :John
] .
```

Manchester Syntax

```
Class: JohnsChildren
  EquivalentTo: hasParent value John
```

OWL/XML Syntax

```
<EquivalentClasses>
  <Class IRI="JohnsChildren"/>
  <ObjectHasValue>
    <ObjectProperty IRI="hasParent"/>
    <Class IRI="John"/>
  </ObjectHasValue>
</EquivalentClasses>
```

As a special case of individuals being interlinked by properties, an individual might be linked to itself. The following example shows how to represent the idea that all narcissists love themselves.

Functional Style Syntax

```
EquivalentClasses(
 :NarcisticPerson
  ObjectHasSelf(:loves )
)
```

RDF/XML Syntax

```
<owl:Class rdf:about="NarcisticPerson">
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="loves"/>
      <owl:hasSelf rdf:datatype="&xsd;boolean">
        true
      </owl:hasSelf>
    </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>
```

Turtle Syntax

```
:NarcisticPerson owl:equivalentClass  [
   rdf:type        owl:Restriction;
   owl:onProperty :loves;
   owl:hasSelf     "true"^^xsd:boolean .
] .
```

Manchester Syntax

```
Class: NarcisticPerson
  EquivalentTo: loves Self
```

OWL/XML Syntax

```
<EquivalentClasses>
  <Class IRI="NarcisticPerson"/>
  <ObjectHasSelf>
    <ObjectProperty IRI="loves"/>
  </ObjectHasSelf>
</EquivalentClasses>
```

5.3 Property Cardinality Restrictions

Using universal and existential quantification, we can say something about all respectively at least one of somebody's children. However, we might want to specify the number of individuals involved in the restriction. Indeed, we can construct classes depending on the number of children. The following example states that John has at most four children who are themselves parents:

Functional Style Syntax

```
ClassAssertion(
  ObjectMaxCardinality( 4:hasChild:Parent )
 :John
```

```
)
```

RDF/XML Syntax

```
<rdf:Description rdf:about="John">
  <rdf:type>
    <owl:Class>
      <owl:Restriction>
        <owl:maxQualifiedCardinality rdf:datatype="&xsd;nonNegativeInteger">
          4
        </owl:maxQualifiedCardinality>
        <owl:onProperty rdf:about="hasChild"/>
        <owl:onClass rdf:about="Parent"/>
      </owl:Restriction>
    </owl:Class>
  </rdf:type>
</rdf:Description>
```

Turtle Syntax

```
:John  rdf:type  [
   rdf:type                     owl:Restriction;
   owl:maxQualifiedCardinality  "4"^^xsd:nonNegativeInteger;
   owl:onProperty              :hasChild;
   owl:onClass                 :Parent
] .
```

Manchester Syntax

```
Individual: John
  Types: hasChild max 4 Parent
```

OWL/XML Syntax

```
<ClassAssertion>
  <ObjectMaxCardinality cardinality="4">
    <ObjectProperty IRI="hasChild"/>
    <Class IRI="Parent"/>
  </ObjectMaxCardinality>
  <NamedIndividual IRI="John"/>
</ClassAssertion>
```

Note that this statement allows John to have arbitrarily many further children who are not parents.

Likewise, it is also possible to declare a minimum number by saying that John is an instance of the class of individuals having at least two children who are parents:

Functional Style Syntax

```
ClassAssertion(
  ObjectMinCardinality( 2:hasChild:Parent )
 :John
)
```

RDF/XML Syntax

```
<rdf:Description rdf:about="John">
  <rdf:type>
    <owl:Class>
      <owl:Restriction>
        <owl:minQualifiedCardinality rdf:datatype="&xsd;nonNegativeInteger">
          2
        </owl:minQualifiedCardinality>
        <owl:onProperty rdf:about="hasChild"/>
        <owl:onClass rdf:about="Parent"/>
      </owl:Restriction>
    </owl:Class>
  </rdf:type>
</rdf:Description>
```

Turtle Syntax

```
:John  rdf:type  [
   rdf:type                      owl:Restriction;
   owl:minQualifiedCardinality  "2"^^xsd:nonNegativeInteger;
   owl:onProperty               :hasChild;
   owl:onClass                  :Parent
 ] .
```

Manchester Syntax

```
Individual: John
  Types: hasChild min 2 Parent
```

OWL/XML Syntax

```
<ClassAssertion>
  <ObjectMinCardinality cardinality="2">
    <ObjectProperty IRI="hasChild"/>
    <Class IRI="Parent"/>
  </ObjectMinCardinality>
  <NamedIndividual IRI="John"/>
</ClassAssertion>
```

If we happen to know the exact number of John's children who are parents, this can be specified as follows:

Functional Style Syntax

```
ClassAssertion(
  ObjectExactCardinality( 3:hasChild:Parent )
 :John
)
```

RDF/XML Syntax

```
<rdf:Description rdf:about="John">
  <rdf:type>
    <owl:Class>
      <owl:Restriction>
        <owl:qualifiedCardinality rdf:datatype="&xsd;nonNegativeInteger">
          3
        </owl:qualifiedCardinality>
```

```
        <owl:onProperty rdf:about="hasChild"/>
        <owl:onClass rdf:about="Parent"/>
      </owl:Restriction>
    </owl:Class>
  </rdf:type>
</rdf:Description>
```

Turtle Syntax

```
:John  rdf:type  [
   rdf:type                   owl:Restriction;
   owl:qualifiedCardinality  "3"^^xsd:nonNegativeInteger;
   owl:onProperty            :hasChild;
   owl:onClass               :Parent
] .
```

Manchester Syntax

```
Individual: John
  Types: hasChild exactly 3 Parent
```

OWL/XML Syntax

```
<ClassAssertion>
  <ObjectExactCardinality cardinality="3">
    <ObjectProperty IRI="hasChild"/>
    <Class IRI="Parent"/>
  </ObjectExactCardinality>
  <NamedIndividual IRI="John"/>
</ClassAssertion>
```

In a cardinality restriction, providing the class is optional; if we just want to talk about the number of all of John's children we can write the following:

Functional Style Syntax

```
ClassAssertion(
  ObjectExactCardinality( 5:hasChild )
 :John
)
```

RDF/XML Syntax

```
<rdf:Description rdf:about="John">
  <rdf:type>
    <owl:Class>
      <owl:Restriction>
        <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">
          5
        </owl:cardinality>
        <owl:onProperty rdf:about="hasChild"/>
      </owl:Restriction>
    </owl:Class>
  </rdf:type>
</rdf:Description>
```

Turtle Syntax

```
:John  rdf:type  [
   rdf:type         owl:Restriction;
   owl:cardinality  "5"^^xsd:nonNegativeInteger;
   owl:onProperty  :hasChild
 ] .
```

Manchester Syntax

```
Individual: John
  Types: hasChild exactly 5
```

OWL/XML Syntax

```
<ClassAssertion>
  <ObjectExactCardinality cardinality="5">
    <ObjectProperty IRI="hasChild"/>
  </ObjectExactCardinality>
  <NamedIndividual IRI="John"/>
</ClassAssertion>
```

5.4 Enumeration of Individuals

A very straightforward way to describe a class is just to enumerate all its instances. OWL provides this possibility, e.g. we can create a class of birthday guests:

Functional Style Syntax

```
EquivalentClasses(
 :MyBirthdayGuests
  ObjectOneOf(:Bill:John:Mary)
)
```

RDF/XML Syntax

```
<owl:Class rdf:about="MyBirthdayGuests">
  <owl:equivalentClass>
    <owl:Class>
      <owl:oneOf rdf:parseType="Collection">
        <rdf:Description rdf:about="Bill"/>
        <rdf:Description rdf:about="John"/>
        <rdf:Description rdf:about="Mary"/>
      </owl:oneOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>
```

Turtle Syntax

```
:MyBirthdayGuests  owl:equivalentClass  [
   rdf:type   owl:Class;
   owl:oneOf  (:Bill :John :Mary )
 ] .
```

Manchester Syntax

```
Class: MyBirthdayGuests
  EquivalentTo: { Bill John Mary }
```

OWL/XML Syntax

```
<EquivalentClasses>
  <Class IRI="MyBirthdayGuests"/>
  <ObjectOneOf>
    <NamedIndividual IRI="Bill"/>
    <NamedIndividual IRI="John"/>
    <NamedIndividual IRI="Mary"/>
  </ObjectOneOf>
</EquivalentClasses>
```

Note that this axiom provides more information than simply asserting class membership of Bill, John, and Mary as described in Section 4.1. In addition to that, it also stipulates that Bill, John, and Mary are the *only* members of MyBirthdayGuest. Therefore, classes defined this way are sometimes referred to as *closed classes* or enumerated sets. If we now assert Jeff as an instance of MyBirthdayGuest, the consequence is that Jeff must be equal to one of the above three persons.

6 Advanced Use of Properties

Until now we focussed on classes and properties that were merely used as building blocks for class expressions. In the following, we will see what other modeling capabilities with properties OWL 2 offers.

6.1 Property Characteristics

Sometimes one property can be obtained by taking another property and changing its direction, i.e. inverting it. For example, the property hasParent can be defined as the inverse property of hasChild:

Functional Style Syntax

```
InverseObjectProperties(:hasParent:hasChild )
```

RDF/XML Syntax

```
<owl:ObjectProperty rdf:about="hasParent">
  <owl:inverseOf rdf:resource="hasChild"/>
</owl:ObjectProperty>
```

Turtle Syntax

```
:hasParent owl:inverseOf:hasChild .
```

Manchester Syntax

```
ObjectProperty: hasParent
  InverseOf: hasChild
```

OWL/XML Syntax

```
<InverseObjectProperties>
  <ObjectProperty IRI="hasParent"/>
  <ObjectProperty IRI="hasChild"/>
</InverseObjectProperties>
```

This would for example allow to deduce for arbitrary individuals A and B, where A is linked to B by the hasChild property, that B and A are also interlinked by the hasParent property. However, we do not need to explicitly assign a name to the inverse of a property if we just want to use it, say, inside a class expression. Instead of using the new hasParent property for the definition of the class Orphan, we can directly refer to it as the hasChild-inverse:

Functional Style Syntax

```
EquivalentClasses(
 :Orphan
  ObjectAllValuesFrom(
    ObjectInverseOf(:hasChild )
   :Dead
  )
)
```

RDF/XML Syntax

```
<owl:Class rdf:about="Orphan">
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty>
        <owl:ObjectProperty>
          <owl:inverseOf rdf:about="hasChild"/>
        </owl:ObjectProperty>
      </owl:onProperty>
      <owl:Class rdf:about="Dead"/>
    </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>
```

Turtle Syntax

```
:Orphan  owl:equivalentClass  [
   rdf:type           owl:Restriction;
   owl:onProperty     [ owl:inverseOf :hasChild ];
   owl:allValuesFrom :Dead
] .
```

Manchester Syntax

```
Class: Orphan
  EquivalentTo: inverse hasChild only Dead
```

OWL/XML Syntax

```
<EquivalentClasses>
  <Class IRI="Orphan"/>
```

```
  <ObjectAllValuesFrom>
    <InverseObjectProperty>
      <ObjectProperty IRI="hasChild"/>
    </InverseObjectProperty>
    <Class IRI="Dead"/>
  </ObjectAllValuesFrom>
</EquivalentClasses>
```

In some cases, a property and its inverse coincide, or in other words, the direction of a property doesn't matter. For instance the property hasSpouse relates A with B exactly if it relates B with A. For obvious reasons, a property with this characteristic is called *symmetric*, and it can be specified as follows

Functional Style Syntax

```
SymmetricObjectProperty(:hasSpouse )
```

RDF/XML Syntax

```
<owl:SymmetricProperty rdf:about="hasSpouse"/>
```

Turtle Syntax

```
:hasSpouse  rdf:type  owl:SymmetricProperty .
```

Manchester Syntax

```
ObjectProperty: hasSpouse
  Characteristics: Symmetric
```

OWL/XML Syntax

```
<SymmetricObjectProperty>
  <ObjectProperty IRI="hasSpouse"/>
</SymmetricObjectProperty>
```

On the other hand, a property can also be *asymmetric* meaning that if it connects A with B it never connects B with A. Clearly (excluding paradoxical scenarios resulting from time travels), this is the case for the property hasChild and is expressed like this:

Functional Style Syntax

```
AsymmetricObjectProperty(:hasChild )
```

RDF/XML Syntax

```
<owl:AsymmetricProperty rdf:about="hasChild"/>
```

Turtle Syntax

```
:hasChild  rdf:type  owl:AsymmetricProperty .
```

Manchester Syntax

```
ObjectProperty: hasChild
  Characteristics: Asymmetric
```

OWL/XML Syntax

```
<AsymmetricObjectProperty>
  <ObjectProperty IRI="hasChild"/>
</AsymmetricObjectProperty>
```

Previously, we considered subproperties in analogy to subclasses. It turns out that it also make sense to transfer the notion of class disjointness to properties: two properties are disjoint if there are no two individuals that are interlinked by both properties. Following common law, we can thus state that parent-child marriages cannot occur:

Functional Style Syntax

```
DisjointObjectProperties(:hasParent:hasSpouse )
```

RDF/XML Syntax

```
<rdf:Description rdf:about="hasParent">
  <owl:propertyDisjointWith rdf:resource="hasSpouse"/>
</rdf:Description>
```

Turtle Syntax

```
:hasParent  owl:propertyDisjointWith :hasSpouse .
```

Manchester Syntax

```
DisjointProperties: hasParent hasSpouse
```

OWL/XML Syntax

```
<DisjointObjectProperties>
  <ObjectProperty IRI="hasParent"/>
  <ObjectProperty IRI="hasSpouse"/>
</DisjointObjectProperties>
```

Properties can also be *reflexive*: such a property relates everything to itself. For the following example, note that everybody has himself as a relative.

Functional Style Syntax

```
ReflexiveObjectProperty(:hasRelative )
```

RDF/XML Syntax

```
<owl:ReflexiveProperty rdf:about="hasRelative"/>
```

Turtle Syntax

```
:hasRelative  rdf:type  owl:ReflexiveProperty .
```

Manchester Syntax

```
ObjectProperty: hasRelative
  Characteristics: Reflexive
```

OWL/XML Syntax

```
<ReflexiveObjectProperty>
  <ObjectProperty IRI="hasRelative"/>
</ReflexiveObjectProperty>
```

Note that this does not necessarily mean that every two individuals which are related by a reflexive property are identical.

Properties can furthermore be *irreflexive*, meaning that no individual can be related to itself by such a role. A typical example is the following which simply states that nobody can be his own parent.

Functional Style Syntax

```
IrreflexiveObjectProperty(:parentOf )
```

RDF/XML Syntax

```
<owl:IrreflexiveProperty rdf:about="parentOf"/>
```

Turtle Syntax

```
:parentOf  rdf:type  owl:IrreflexiveProperty .
```

Manchester Syntax

```
Objectproperty: parentOf
  Characteristics: Irreflexive
```

OWL/XML Syntax

```
<IreflexiveObjectProperty>
  <ObjectProperty IRI="parentOf"/>
</IreflexiveObjectProperty>
```

Next, consider the hasHusband property. As every person can have only one husband (which we take for granted for the sake of the example), every individual can be linked by the hasHusband property to at most one other individual. This kind of properties are called *functional* and are described as follows:

Functional Style Syntax

```
FunctionalObjectProperty(:hasHusband )
```

RDF/XML Syntax

```
<owl:FunctionalProperty rdf:about="hasHusband"/>
```

Turtle Syntax

```
:hasHusband  rdf:type  owl:FunctionalProperty .
```

Manchester Syntax

```
ObjectProperty: hasHusband
  Characteristics: Functional
```

OWL/XML Syntax

```
<FunctionalObjectProperty>
  <ObjectProperty IRI="hasHusband"/>
</FunctionalObjectProperty>
```

Note that this statement does not require every individual to have a husband, it just states that there can be no more than one. Moreover, if we additionally had a statement that Mary's husband is James and another that Mary's husband is Jim, it could be inferred that Jim and James must refer to the same individual.

It is also possible to indicate that the inverse of a given property is functional:

Functional Style Syntax

```
InverseFunctionalObjectProperty(:hasHusband )
```

RDF/XML Syntax

```
<owl:InverseFunctionalProperty rdf:about="hasHusband"/>
```

Turtle Syntax

```
:hasHusband  rdf:type  owl:InverseFunctionalProperty .
```

Manchester Syntax

```
ObjectProperty: hasHusband
  Characteristics: InverseFunctional
```

OWL/XML Syntax

```
<InverseFunctionalObjectProperty>
  <ObjectProperty IRI="hasHusband"/>
</InverseFunctionalObjectProperty>
```

This indicates that an individual can be husband of at most one other individual. The example also indicates the difference between functionality and inverse functionality, as in a polygynous situation the former axiom is valid whereas the latter isn't.

Now have a look at a property hasAncestor which is meant to link individuals A and B whenever A is a direct descendant of B. Clearly, the property hasParent is a “special case” of hasAncestor and can be defined as a subproperty thereof. Still, it would be nice to "automatically" include parents of parents (and parents of parents of parents). This can be done by defining hasAncestor as *transitive* property. A transitive property interlinks two individuals A and C whenever it interlinks A with B and B with C for some individual B.

Functional Style Syntax

```
TransitiveObjectProperty(:hasAncestor )
```

RDF/XML Syntax

```
<owl:TransitiveProperty rdf:about="hasAncestor"/>
```

Turtle Syntax

```
:hasAncestor  rdf:type  owl:TransitiveProperty .
```

Manchester Syntax

```
ObjectProperty: hasAncestor
  Characteristics: Transitive
```

OWL/XML Syntax

```
<TransitiveObjectProperty>
  <ObjectProperty IRI="hasAncestor"/>
</TransitiveObjectProperty>
```

6.2 Property Chains

While the last example from the previous section implied the presence of an hasAncestor property whenever there is a chain of hasParent properties, we might want to be a bit more specific and define, say, a hasGrandparent property instead. Technically, this means that we want hasGrandparent to connect all individuals that are linked by a chain of exactly two hasParent properties. In contrast to the previous hasAncestor example, we do not want hasParent to be a special case of hasGrandparent nor do we want hasGrandparent to refer to great-grandparents etc. We can express that every such chain has to be spanned by a hasGrandparent property as follows:

Functional Style Syntax

```
SubObjectPropertyOf(
  ObjectPropertyChain(:hasParent:hasParent )
 :hasGrandparent
)
```

RDF/XML Syntax

```
<rdf:Description rdf:about="hasGrandparent">
      <owl:propertyChainAxiom rdf:parseType="Collection">
              <owl:ObjectProperty rdf:about="hasParent"/>
              <owl:ObjectProperty rdf:about="hasParent"/>
      </owl:propertyChainAxiom>
</rdf:Description>
```

Turtle Syntax

```
:hasGrandparent  owl:propertyChainAxiom  (:hasParent :hasParent ) .
```

Manchester Syntax

```
ObjectProperty: hasGrandparent
  SubPropertyChain: hasParent o hasParent
```

OWL/XML Syntax

```
<SubObjectPropertyOf>
  <PropertyChain>
    <ObjectProperty IRI="hasParent"/>
    <ObjectProperty IRI="hasParent"/>
  </PropertyChain>
  <ObjectProperty IRI="hasGrandparent"/>
</SubObjectPropertyOf>
```

6.3 Keys

In OWL 2 a collection of (data or object) properties can be assigned as a key to a class expression. This means that each named instance of the class expression is uniquely identified by the set of values which these properties attain in relation to the instance.

A simple example of this would be the identification of a person by her social security number.

Functional Style Syntax

```
HasKey(:Person () (:hasSSN ) )
```

RDF/XML Syntax

```
<owl:Class rdf:about="Person">
  <owl:hasKey rdf:parseType="Collection">
    <owl:ObjectProperty rdf:about="hasSSN"/>
  </owl:hasKey>
</owl:Class>
```

Turtle Syntax

```
:Person owl:hasKey (:hasSSN ) .
```

Manchester Syntax

```
HasKey: Person hasSSN
```

OWL/XML Syntax

```
<HasKey>
  <Class IRI="Person"/>
  <ObjectProperty IRI="hasSSN"/>
</HasKey>
```

7 Advanced Use of Datatypes

Editor's Note: To be done: Discuss available datatypes.

In Section 4.8, we learned that individuals can be endowed with numerical information, esentially by connecting them to a data value via a datatype property - just like object properties link to other domain individuals. In fact, these parallels extend to the more advanced features of datatype usage.

First of all, data values are grouped into datatypes and we have seen in Section 4.8 how a range restriction on a datatype property can be used to indicate the kind of values this property can link to. Moreover, it is possible to express and define new datatypes by constraining or combining existing ones. Datatypes can be restricted via so-called *facets*. In the following example, we define a new datatype for a person's age by constraining the datatype integer to values between (inclusively) 0 and 150.

Functional Style Syntax

```
DatatypeDefinition(
 :personAge
  DatatypeRestriction( xsd:integer
    xsd:minInclusive "0"^^xsd:integer
    xsd:maxInclusive "150"^^xsd:integer
  )
)
```

RDF/XML Syntax

```
<rdf:Description rdf:about="personAge">
  <owl:equivalentClass>
    <owl:Datatype>
      <owl:onDatatype rdf:resource="http://www.w3.org/2001/XMLSchema#integer"/>
      <owl:withRestrictions rdf:parseType="Collection">
        <xsd:minInclusive
rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">0</xsd:minInclusive>
        <xsd:maxInclusive
rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">150</xsd:maxInclusive>
      </owl:withRestrictions>
    </owl:Datatype>
  </owl:equivalentClass>
</rdf:Description>
```

Turtle Syntax

```
:personAge  owl:equivalentClass
  [ rdf:type  rdfs:Datatype;
    owl:onDatatype  xsd:Integer;
    owl:withRestrictions (
       [ xsd:minInclusive  "0"^^xsd:integer ]
       [ xsd:maxInclusive  "150"^^xsd:integer ]
    )
  ] .
```

Manchester Syntax

```
Datatype: personAge
  EquivalentTo: integer[<= 0 , >= 150]
```

OWL/XML Syntax

```
<EquivalentClasses>
   <Datatype IRI="personAge"/>
   <DatatypeRestriction>
      <Datatype IRI="&xsd;integer"/>
      <FacetRestriction facet="&xsd;minInclusive">
        <Literal datatypeIRI="&xsd;integer">0</Literal>
      </FacetRestriction>
      <FacetRestriction facet="&xsd;maxInclusive">
        <Literal datatypeIRI="&xsd;integer">150</Literal>
      </FacetRestriction>
   </DatatypeRestriction>
 </EquivalentClasses>
```

Likewise, datatypes can be combined just like classes by complement, intersection and union. Thereby, assuming we have already defined a datatype MinorAge, we can define the datatype MajorAge by excluding all data values of MinorAge from PersonAge:

Functional Style Syntax

```
DatatypeDefinition(
 :majorAge
  DataIntersectionOf(
   :personAge
    DataComplementOf(:minorAge )
  )
)
```

RDF/XML Syntax

```
<rdf:Description rdf:about="majorAge">
  <owl:equivalentClass>
    <owl:Datatype>
      <owl:intersectionOf rdf:parseType="Collection">
        <rdf:Description rdf:about="personAge"/>
        <rdfs:Datatype>
          <owl:datatypeComplementOf about="minorAge"/>
        </rdfs:Datatype>
      </owl:intersectionOf>
    </owl:Datatype>
  </owl:equivalentClass>
</rdf:Description>
```

Turtle Syntax

```
:majorAge  owl:equivalentClass
   [ rdf:type  rdfs:Datatype;
     owl:intersectionOf (
       :personAge
        [ rdf:type  rdfs:Datatype;
          owl:datatypeComplementOf :minorAge ]
     )
   ] .
```

Manchester Syntax

```
Datatype: majorAge
   EquivalentTo: personAge and not minorAge
```

OWL/XML Syntax

```
<EquivalentClasses>
   <Datatype IRI="majorAge"/>
   <DataIntersectionOf>
      <Datatype IRI="personAge"/>
      <DataComplementOf>
        <Datatype IRI="minorAge"/>
      </DataComplementOf>
   </DataIntersectionOf>
 </EquivalentClasses>
```

Moreover, a new datatype can be generated by just enumerating the data values it contains.

Functional Style Syntax

```
DatatypeDefinition(
 :toddlerAge
  DataOneOf( "1"^^xsd:integer "2"^^xsd:integer )
)
```

RDF/XML Syntax

```
<rdf:Description rdf:about="toddlerAge">
   <owl:equivalentClass>
     <owl:Datatype>
       <owl:oneOf rdf:parseType="Collection">
          <rdf:Description rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">1</
rdf:Description>
          <rdf:Description rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">2</
rdf:Description>
       </owl:oneOf>
     </owl:Datatype>
   </owl:equivalentClass>
</rdf:Description>
```

Turtle Syntax

```
:toddlerAge  owl:equivalentClass
   [ rdf:type rdfs:Datatype;
```

```
    owl:oneOf (  "1"^^xsd:integer  "2"^^xsd:integer )
  ] .
```

Manchester Syntax

```
Datatype: toddlerAge
   EquivalentTo: { 1 2 }
```

OWL/XML Syntax

```
<EquivalentClasses>
   <Datatype IRI="toddlerAge"/>
   <DataOneOf>
     <Literal datatypeIRI="http://www.w3.org/2001/XMLSchema#integer">1</Literal>
     <Literal datatypeIRI="http://www.w3.org/2001/XMLSchema#integer">2</Literal>
   </DataOneOf>
</EquivalentClasses>
```

In Section 6.1, we saw ways of characterizing object properties. Some of those are also available for datatype properties. For example, we can express that every person has only one age by characterizing the hasAge datatype property as functional:

Functional Style Syntax

```
FunctionalDataProperty(:hasAge )
```

RDF/XML Syntax

```
<owl:FunctionalProperty rdf:about="hasAge"/>
```

Turtle Syntax

```
:hasAge  rdf:type  owl:FunctionalProperty .
```

Manchester Syntax

```
DataProperty: hasHusband
  Characteristics: Functional
```

OWL/XML Syntax

```
<FunctionalDataProperty>
 <DataProperty IRI="hasHusband"/>
</FunctionalDataProperty>
```

New classes can be defined by restrictions on datatype properties. The following example defines the class teenager as all individuals whose age is between 13 and 19 years.

Functional Style Syntax

```
SubClassOf(
 :Teenager
  ObjectSomeValuesFrom(:hasAge
```

```
    DatatypeRestriction( xsd:integer
      xsd:minInclusive "13"^^xsd:integer
      xsd:maxInclusive "19"^^xsd:integer
    )
  )
)
```

RDF/XML Syntax

```
<owl:Class rdf:about="Teenager">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="hasAge"/>
      <owl:someValuesFrom>
        <rdfs:Datatype>
          <owl:onDataType rdf:resource="&xsd;integer"/>
          <owl:withRestrictions rdf:parseType="Collection">
            <xsd:minExclusive rdf:datatype="&xsd;integer">
              12
            </xsd:minExclusive>
            <xsd:maxInclusive rdf:datatype="&xsd;integer">
              19
            </xsd:maxInclusive>
          </owl:withRestrictions>
        </rdfs:Datatype>
      </owl:someValuesFrom>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

Turtle Syntax

```
:Teenager  rdfs:subClassOf
       [ rdf:type             owl:Restriction;
         owl:onProperty      :hasAge;
         owl:someValuesFrom
          [ rdf:type             rdfs:Datatype;
            owl:onDatatype       xsd:integer;
            owl:withRestrictions (  [ xsd:minInclusive     "13"^^xsd:integer ]
                                    [ xsd:maxInclusive     "19"^^xsd:integer ]
            )
          ]
       ] .
```

Manchester Syntax

```
Class: Teenager
  SubClassOf: hasAge some integer[<= 13 , >= 19]
```

OWL/XML Syntax

```
<SubClassOf>
  <Class IRI="Teenager"/>
  <DataSomeValuesFrom>
    <DataProperty IRI="hasAge"/>
    <DatatypeRestriction>
      <Datatype IRI="&xsd;integer"/>
      <FacetRestriction facet="&xsd;minInclusive">
        <Literal datatypeIRI="&xsd;integer">13</Literal>
```

```
      </FacetRestriction>
      <FacetRestriction facet="&xsd;maxInclusive">
        <Literal datatypeIRI="&xsd;integer">19</Literal>
      </FacetRestriction>
    </DatatypeRestriction>
  </DataSomeValuesFrom>
</SubClassOf>
```

8 Document Information and Annotations

In the following, we describe features of OWL 2 which do not actually contribute to the “logical” knowledge specified in the ontology. Rather these are used to provide additional information about the ontology itself, axioms, or even single entities.

8.1 Annotating Axioms and Entities

In many cases, we want to furnish parts of our OWL ontology with information that actually does not describe the domain itself but talks about the description of the domain. OWL provides annotations for this purpose. An OWL annotation simply associates property-value pairs with parts of an ontology, or the entire ontology itself. Even annotations themselves can be annotated. Annotation information is not really part of the logical meaning of an ontology.

So, for example, we could add information to one of the classes of our ontology, giving a natural language description of its meaning.

Functional Style Syntax

```
AnnotationAssertion( rdfs:comment:Person "Represents the set of all people." )
```

RDF/XML Syntax

```
<owl:Class rdf:about="Person">
  <rdfs:comment>Represents the set of all people.</rdfs:comment>
</owl:Class>
```

Turtle Syntax

```
:Person  rdfs:comment  "Represents the set of all people."^^xsd:string .
```

Manchester Syntax

```
Class: Person
  Annotations: rdfs:comment "Represents the set of all people."
```

OWL/XML Syntax

```
<AnnotationAssertion>
  <AnnotationProperty IRI="&rdfs;comment"/>
  <IRI>Person</IRI>
  <Literal>Represents the set of all people.</Literal>
```

```
</AnnotationAssertion>
```

The following is an example of an axiom with an annotation.

Functional Style Syntax

```
SubClassOf(
  Annotation( rdfs:comment "States that every man is a person." )
 :Man
 :Person
)
```

RDF/XML Syntax

```
<owl:Class rdf:about="Man">
  <rdfs:subClassOf rdf:resource="Person"/>
</owl:Class>
<owl:Axiom>
  <owl:annotatedSource rdf:resource="Man"/>
  <owl:annotatedProperty rdf:resource="&rdfs;subClassOf"/>
  <owl:annotatedTarget rdf:resource="Person"/>
  <rdfs:comment>States that every man is a person.</rdfs:comment>
</owl:Axiom>
```

Turtle Syntax

```
:Man rdfs:subClassOf:Person .
 []  rdf:type       owl:Axiom;
     owl:annotatedSource   :Man;
     owl:annotatedProperty  rdfs:subClassOf;
     owl:annotatedTarget   :Person;
     rdfs:comment     "States that every man is a person."^^xsd:string .
```

Manchester Syntax

```
Class: Man
  SubClassOf: Annotations: rdfs:comment "States that every man is a person." Person
```

OWL/XML Syntax

```
<SubClassOf>
  <Annotation>
      <AnnotationProperty IRI="&rdfs;comment"/>
      <Literal datatypeIRI="xsd:string">"States that every man is a person."</Literal>
  </Annotation>
  <Class IRI="Man"/>
  <Class IRI="Person"/>
</SubClassOf>
```

Often such annotations are used in tools to provide access to natural language text to be displayed in help interfaces.

8.2 Ontology Management

In OWL, general information about a topic is almost always gathered into an ontology

that is then used by various applications. We can also provide a name for OWL ontologies, which is generally the place where the ontology document is located in the web. Particular information about a topic can also be placed in an ontology, if it is used by different applications.

Functional Style Syntax

```
Ontology(<http://example.com/owl/families>
  ...
)
```

RDF/XML Syntax

```
<rdf:RDF ...>
  <owl:Ontology rdf:about="http://example.com/owl/families"/>
  ...
</rdf:RDF>
```

Turtle Syntax

```
<http://example.com/owl/families> rdf:type owl:Ontology .
```

Manchester Syntax

```
Ontology: <http://example.com/owl/families>
```

OWL/XML Syntax

```
<Ontology ...
  ontologyIRI="http://example.com/owl/families">
  ...
</Ontology>
```

We place OWL ontologies into OWL documents, which are then placed into local filesystems or on the World Wide Web. Aside from containing an OWL ontology, OWL documents also contain information about transforming the short names normally used in OWL ontologies (e.g., Person) into IRIs, by providing the expansion for prefixes. The IRI is then the concatention of the prefix expansion and the reference.

In our example we have so far used a number of prefixes, including `xsd` and the empty prefix. The former prefix has been used in compact names for XML Schema datatypes, whose IRIs are fixed by the XML Schema recommendation. We thus must use the standard expansion for `xsd`, which is `http://www.w3.org/2001/XMLSchema#`. The expansion we pick for the other prefix will affect the names of the classes, properties, and individuals in our ontology, as well as the name of the ontology itself. If we are going to put the ontology on the web, we should pick an expansion that is in some part of the web that we control, so that we are not using someone else's names by accident. (Here we use a made-up place that no one controls.) The two XML-based syntaxes need namespaces for built-in names and also can use XML entities for namespaces.

Functional Style Syntax

```
Prefix(:=<http://example.com/owl/families/>)
Prefix(otherOnt:=<http://example.org/otherOntologies/families/>)
Prefix(xsd:=<http://www.w3.org/2001/XMLSchema#>)
Prefix(owl:=<http://www.w3.org/2002/07/owl#>)

Ontology(<http://example.com/owl/families>
   ...
)
```

RDF/XML Syntax

```
<!DOCTYPE rdf:RDF [
    <!ENTITY owl "http://http://www.w3.org/2002/07/owl#" >
    <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
    <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
    <!ENTITY otherOnt "http://example.org/otherOntologies/families/" >
]>

<rdf:RDF xml:base="http://example.com/owl/families/"
    xmlns="http://example.com/owl/families/"
    xmlns:otherOnt="http://example.org/otherOntologies/families/"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:xml="http://www.w3.org/2001/XMLSchema#">

    <owl:Ontology rdf:about="http://example.com/owl/families"/>
...
```

Turtle Syntax

```
@prefix: <http://example.com/owl/families/> .
@prefix otherOnt: <http://example.org/otherOntologies/families/> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
```

Manchester Syntax

```
Prefix: <http://example.com/owl/families/>
Prefix: owl <http://www.w3.org/2002/07/owl#>
Prefix: otherOnt <http://example.org/otherOntologies/families/>
```

OWL/XML Syntax

```
<!DOCTYPE Ontology [
   <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
]>
<Ontology
   xml:base="http://example.com/owl/families/"
   xmlns="http://www.w3.org/2002/07/owl#"
   xmlns:owl="http://http://www.w3.org/2002/07/owl#"
   xmlns:g="http://example.org/otherOntologies/families/"
   ontologyIRI="http://example.com/owl/families">
 ...
</Ontology>
```

It is also common in OWL to reuse general information that is stored in one ontology in other ontologies. Instead of requiring the copying of this information, OWL allows the import of the contents of entire ontologies in other ontologies, using import statements, as follows:

Functional Style Syntax

```
Import( http://example.org/otherOntologies/families.owl )
```

RDF/XML Syntax

```
<owl:Ontology rdf:about="http://example.com/owl/families">
  <owl:imports rdf:resource="http://example.org/otherOntologies/families.owl" />
</owl:Ontology>
```

Turtle Syntax

```
<http://example.com/owl/families> owl:imports
<http://example.org/otherOntologies/families.owl> .
```

Manchester Syntax

```
Import: <http://example.org/otherOntologies/families.owl>
```

OWL/XML Syntax

```
<Prefix name="otherOnt" IRI="http://example.org/otherOntologies/families/"/>
<Import>http://example.org/otherOntologies/families.owl</Import>
```

As the Semantic Web and ontology construction is distributed, it is common for ontologies to use different names for the same concept, property, or individual. As we have seen, several constructs in OWL can be used to state that different names refer to the same class, property, or individual, so, for example, we could – instead of tediously renaming entities – tie the names used in our ontology to the names used in an imported ontology as follows:

Functional Style Syntax

```
SameIndividual(:John otherOnt:JohnBrown )
SameIndividual(:Mary otherOnt:MaryBrown )
EquivalentClasses(:Adult otherOnt:Grownup )
EquivalentObjectProperties(:hasChild otherOnt:child )
EquivalentDataProperties(:hasAge otherOnt:age )
```

RDF/XML Syntax

```
<rdf:Description rdf:about="John">
  <owl:sameAs rdf:resource="&otherOnt;JohnBrown"/>
</rdf:Description>

<rdf:Description rdf:about="Mary">
  <owl:sameAs rdf:resource="&otherOnt;MaryBrown"/>
</rdf:Description>
```

```
<owl:Class rdf:about="Adult">
  <owl:equivalentClass rdf:resource="&otherOnt;Grownup"/>
</owl:Class>

<owl:DatatypeProperty rdf:about="hasAge">
  <owl:equivalentProperty rdf:resource="&otherOnt;age"/>
</owl:DatatypeProperty>

<owl:Class rdf:about="Adult">
  <owl:equivalentClass rdf:resource="&otherOnt;Grownup"/>
</owl:Class>
```

Turtle Syntax

```
:Mary      owl:sameAs              otherOnt:MaryBrown .
:John      owl:sameAs              otherOnt:JohnBrown .
:Adult     owl:equivalentClass     otherOnt:Grownup .
:hasChild  owl:equivalentProperty  otherOnt:child .
:hasAge    owl:equivalentProperty  otherOnt:age .
```

Manchester Syntax

```
SameIndividual: John otherOnt:JohnBrown
SameIndividual: Mary otherOnt:MaryBrown
EquivalentClasses: Adult otherOnt:Grownup
EquivalentProperties: hasChild otherOnt:child
EquivalentProperties: hasAge   otherOnt:age
```

OWL/XML Syntax

```
<SameIndividuals>
  <Individual IRI="John"/>
  <Individual IRI="otherOnt:JohnBrown"/>
</SameIndividuals>

<SameIndividuals>
  <Individual IRI="Mary"/>
  <Individual IRI="otherOnt:MaryBrown"/>
</SameIndividuals>

<EquivalentClasses>
  <Class IRI="Adult"/>
  <Class IRI="otherOnt:Grownup"/>
</EquivalentClasses>

<EquivalentObjectProperties>
  <ObjectProperty IRI="hasChild"/>
  <ObjectProperty IRI="otherOnt:child"/>
</EquivalentObjectProperties>

<EquivalentDataProperties>
  <DataProperty IRI="hasAge"/>
  <DataProperty IRI="otherOnt:age"/>
</EquivalentDataProperties>
```

8.3 Entity Declarations

To help with managing ontologies, OWL has the notion of declarations. The basic idea is that each class, property, or individual is supposed to be declared in an ontology, and then it can be used in that ontology and ontologies that import that ontology.

In the Manchester syntax, declarations are implicit. Constructs that provide information about a class, property, or individual implicitly declare that class, property, or individual if needed. The other syntaxes have explicit declarations.

Functional Style Syntax

```
Declaration( NamedIndividual(:John ) )
Declaration( Class(:Person ) )
Declaration( ObjectProperty(:hasWife ) )
Declaration( DataProperty(:hasAge ) )
```

RDF/XML Syntax

```
<owl:NamedIndividual rdf:about="John"/>
<owl:Class rdf:about="Person"/>
<owl:ObjectProperty rdf:about="hasWife"/>
<owl:DatatypeProperty rdf:about="hasAge"/>
```

Turtle Syntax

```
:John    rdf:type owl:NamedIndividual .
:Person  rdf:type owl:Class .
:hasWife rdf:type owl:ObjectProperty .
:hasAge  rdf:type owl:DatatypeProperty .
```

Manchester Syntax

```
Individual: John
Class: Person
ObjectProperty: hasWife
Dataproperty: hasAge
```

OWL/XML Syntax

```
<Declaration>
    <NamedIndividual IRI="John"/>
</Declaration>
<Declaration>
    <Class IRI="Person"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="hasWife"/>
</Declaration>
<Declaration>
    <DataProperty IRI="hasAge"/>
</Declaration>
```

However, an IRI may denote different entity-types (e.g. both an individual and a class) at the same time. This possibility, called “punning,” has been introduced to allow for a

certain amount of metamodeling; we give an example of this in Section 9. Still, OWL 2 does require some discipline in using and reusing names. To allow a more readable syntax, and for other technical reasons, OWL 2 DL requires that a name is not used for more than one property type (object, datatype or annotation property) nor can an IRI denote both a class and a datatype. Moreover, “built-in” names (such as those used by RDF and RDFS and various syntaxes of OWL) cannot be freely used in OWL.

9 OWL 2 DL and OWL 2 Full

There are two alternative ways of assigning meaning to ontologies in OWL 2 called the direct model-theoretic semantics [OWL 2 Direct Semantics] and the RDF-based semantics [*OWL 2 RDF-Based Semantics*]. Informally, the notion "OWL 2 DL" is used to refer to OWL 2 ontologies interpreted using the direct semantics, and the notion "OWL 2 Full" is used when considering the RDF-based semantics. The OWL 2 functional-style syntax document [*OWL 2 Specification*] additionally lists a few conditions which must be met by an OWL 2 ontology to qualify as OWL 2 DL.

The direct model-theoretic semantics [*OWL 2 Direct Semantics*] provides a meaning for OWL 2 in a [*Description Logic*] style. The RDF-based semantics [*OWL 2 RDF-Based Semantics*] is an extension of the semantics for RDFS [*RDF Semantics*] and is based on viewing OWL 2 ontologies as RDF graphs.

When thinking about ontologies the differences between these two semantics are generally quite slight. Indeed, given an OWL 2 DL ontology, inferences drawn using the direct semantics remain valid inferences under the RDF-based semantics - see the correspondence theorem in Section 7.2 of the RDF-based semantics document [*OWL 2 RDF-Based Semantics*]. The two main differences are that under the direct model-theoretic semantics annotations have no formal meaning and under the RDF-based semantics there are some extra inferences that arise from the RDF view of the universe.

Conceptually, we can think of the difference between OWL 2 DL and OWL 2 Full in two ways:

- One can see OWL 2 DL as a syntactically restricted version of OWL 2 Full where the restrictions are designed to make life easier for implementors. In fact, since OWL 2 Full (under the RDF-based semantics) is undecidable, OWL 2 DL (under the direct model-theoretic semantics) makes writing a reasoner that, in principle, can return all "yes or no" answers (subject to resource constraints) possible. As a consequence of its design, there are several production quality reasoners that cover the entire OWL 2 DL language under the direct model-theoretic semantics. There are no such reasoners for OWL 2 Full under the RDF-based semantics.

- One can see OWL 2 Full as the most straightforward extension of RDFS. As such, the RDF-based semantics for OWL 2 Full follows the RDFS semantics and general syntactic philosophy (i.e., everything is a triple and the language is fully reflective).

Of course, the two semantics have been designed together and thus have influenced each other. For example, one design goal of OWL 2 was to bring OWL 2 DL

syntactically closer to OWL 2 Full (that is, to allow more RDF Graphs/OWL 2 Full ontologies to be legal OWL 2 DL ontologies). This led to the incorporation of so-called *punning* into OWL 2, e.g., using the same IRI as a name for both a class and an individual. An example of such usage would be the following, which states that John is a father, and that father is a social role.

Functional Style Syntax

```
ClassAssertion(:Father:John )
ClassAssertion(:SocialRole:Father )
```

RDF/XML Syntax

```
<Father rdf:about="John"/>
<SocialRole rdf:about="Father"/>
```

Turtle Syntax

```
:John rdf:type:Father .
:Father rdf:type:SocialRole .
```

Manchester Syntax

```
Individual: John
  Types: Father
Individual: Father
  Types: SocialRole
```

OWL/XML Syntax

```
<ClassAssertion>
    <Class URI="Father"/>
    <NamedIndividual IRI="John"/>
</ClassAssertion>
<ClassAssertion>
    <Class URI="SocialRole"/>
    <NamedIndividual IRI="Father"/>
</ClassAssertion>
```

Note that in the first statement, Father is used as a class, while in the second it is used as an individual. In this sense, SocialRole acts as a metaclass for the class Father.

In OWL 1, a document containing these two statements would be an OWL 1 Full document, but not an OWL 1 DL document. In OWL 2 DL, however, this is allowed. It must be noted, though, that the direct model-theoretic semantics of OWL 2 DL accommodates this by understanding the class Father and the individual Father as two different views on the same IRI, i.e. they are interpreted semantically as if they were distinct.

10 OWL 2 Profiles

In addition to OWL 2 DL and OWL 2 Full, OWL 2 specifies three profiles. OWL 2, in general, is a very expressive language (both computationally and for users) and thus

can be difficult to implement well and to work with. These additional profiles are designed to be approachable subsets of OWL 2 sufficient for a variety of applications. As with OWL 2 DL, computational considerations are a major requirement of these profiles (and they are all much easier to implement with robust scalability given existing technology), but there are many subsets of OWL 2 that have good computational properties. The selected OWL 2 profiles were identified as having substantial user communities already, although there were several others not included and one should expect more to emerge. The [OWL 2 Profiles] document provides a clear template for specifying additional profiles.

In order to guarantee for scalable reasoning, the existing profiles share some limitations regarding their expressiveness. In general, they disallow negation and disjunction, as these constructs complicate reasoning and have shown to be only rarely needed for modeling. For example, in none of the profiles it is possible to specify that every person is male or female. Further specific modeling restrictions of the profiles will be dealt with in the sections on the individual profiles.

We discuss each profile and its design rationale, and provide some guidance for users in selecting which profile to work with. Please be aware that this discussion is not comprehensive, nor can it be. Part of any decision has to be based on available tooling and how that fits in with the rest of your system or workflow.

By and large, different profiles can be distinguished syntactically with there being inclusion relations between various profiles. For example, OWL 2 DL can be seen as a syntactic fragment of OWL 2 Full and OWL 2 QL is a syntactic fragment of OWL 2 DL (and thus of OWL 2 Full). Ideally, one can use a reasoner (or other tool) that is conforming for a superprofile on the subprofile with no change in the results derived. For profiles such as OWL 2 EL and OWL 2 QL in relation to OWL 2 DL this principle does hold: Every conforming OWL 2 DL reasoner is an OWL 2 EL and OWL 2 QL reasoner (but may differ in performance since the OWL 2 DL reasoner is tuned for a more general set of cases).

10.1 OWL 2 EL

Working with OWL 2 EL is fairly similar to working with OWL 2 DL: one can use class expressions on both sides of a subClassStatement and even infer such relations. For many large, class-expression oriented ontologies, by only a little simplification one can get an OWL 2 EL ontology and preserve the bulk of the meaning of the original ontology.

OWL 2 EL is designed with large biohealth ontologies in mind, such as SNOMED-CT, the NCI thesaurus, and Galen. Common characteristics of such ontologies include complex structural descriptions (e.g., defining certain body parts in terms of what parts they contain and are contained in or propagating diseases along part-subpart relations), huge numbers of classes, the heavy use of classification to manage the terminology, and the application of the resulting terminology to vast amounts of data. Thus, OWL 2 EL has a comparatively expressive class expression language and it has no restrictions on how they may be used in axioms. It also has fairly expressive property expressions, including property chains but excluding inverse.

Sensible use of OWL 2 EL is obviously not restricted to the biohealth domain: as with

the other profiles, OWL 2 EL is domain independent. However, OWL 2 EL shines when your domain and your application require recognition of structurally complex objects. Such domains include system configurations, product inventories, and many scientific domains.

Besides negation and disjunction, OWL 2 EL also disallows universal quantification on properties. Therefore propositions like “all children of a rich person are rich” cannot be stated. Moreover, as all kinds of role inverses are not available, there is no way of specifying that, say, parentOf and childOf are inverses of each other.

The following is an example which uses some of the typical modeling features available in OWL 2 EL.

Functional Style Syntax

```
SubClassOf(
 :Father
  ObjectIntersectionOf(:Man:Parent )
)

EquivalentClass(
 :Parent
  ObjectSomeValuesFrom(
   :hasChild
   :Person
  )
)

EquivalentClasses(
 :NarcisticPerson
  ObjectHasSelf(:loves )
)

DisjointClasses(
 :Mother
 :Father
 :YoungChild
)

SubObjectPropertyOf(
  ObjectPropertyChain(:hasFather:hasBrother )
 :hasUncle
)

NegativeObjectPropertyAssertion(
 :hasDaughter
 :Bill
 :Susan
)
```

RDF/XML Syntax

```
<owl:Class rdf:about="Father">
  <rdfs:subClassOf>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="Man"/>
        <owl:Class rdf:about="Parent"/>
```

```
      </owl:intersectionOf>
    </owl:Class>
  </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:about="Parent">
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="hasChild"/>
      <owl:someValuesFrom rdf:resource="Person"/>
    </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>

<owl:Class rdf:about="NarcisticPerson">
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="loves"/>
      <owl:hasSelf rdf:datatype="&xsd;boolean">
        true
      </owl:hasSelf>
    </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>

<owl:AllDisjointClasses>
  <owl:members rdf:parseType="Collection">
    <owl:Class rdf:about="Mother"/>
    <owl:Class rdf:about="Father"/>
    <owl:Class rdf:about="YoungChild"/>
  </owl:members>
</owl:AllDisjointClasses>

<rdf:Description rdf:about="hasUncle">
  <owl:propertyChainAxiom rdf:parseType="Collection">
    <owl:ObjectProperty rdf:about="hasFather"/>
    <owl:ObjectProperty rdf:about="hasBrother"/>
  </owl:propertyChainAxiom>
</rdf:Description>

<owl:NegativePropertyAssertion>
  <owl:sourceIndividual rdf:about="Bill"/>
  <owl:assertionProperty rdf:about="hasDaughter"/>
  <owl:targetIndividual rdf:about="Susan"/>
</owl:NegativePropertyAssertion>
```

Turtle Syntax

```
:Father  rdfs:subClassOf  [
   rdf:type            owl:Class;
   owl:intersectionOf  (:Man :Parent )
 ] .

:Parent  owl:equivalentClass  [
   rdf:type            owl:Restriction;
   owl:onProperty     :hasChild;
   owl:someValuesFrom :Person
 ] .

:NarcisticPerson  owl:equivalentClass  [
   rdf:type        owl:Restriction;
```

```
  owl:onProperty :loves;
  owl:hasSelf     true
] .

[] rdf:type     owl:AllDisjointClasses;
   owl:members  (:Mother :Father:YoungChild ) .

:hasUncle  owl:propertyChainAxiom  (:hasFather :hasBrother ) .

[]  rdf:type                owl:NegativePropertyAssertion;
    owl:sourceIndividual  :Bill;
    owl:assertionProperty :hasDaughter;
    owl:targetIndividual  :Susan .
```

Manchester Syntax

```
Class: Father
  SubClassOf: Man and Parent

Class: Parent
  EquivalentTo: hasChild some Person

Class: NarcisticPerson
  EquivalentTo: loves Self

DisjointClasses: Mother Father YoungChild

ObjectProperty: hasUncle
  SubPropertyChain: hasFather o hasBrother

Individual: Bill
  Facts: not hasDaughter Susan
```

OWL/XML Syntax

```
<SubClassOf>
  <Class IRI="Father"/>
  <ObjectIntersectionOf>
    <Class IRI="Man"/>
    <Class IRI="Parent"/>
  </ObjectIntersectionOf>
</SubClassOf>

<EquivalentClasses>
  <Class IRI="Parent"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="hasChild"/>
    <Class IRI="Person"/>
  </ObjectSomeValuesFrom>
</EquivalentClasses>

<EquivalentClasses>
  <Class IRI="NarcisticPerson"/>
  <ObjectHasSelf>
    <ObjectProperty IRI="loves"/>
  </ObjectHasSelf>
</EquivalentClasses>

<DisjointClasses>
    <Class IRI="Father"/>
    <Class IRI="Mother"/>
```

```
    <Class IRI="YoungChild"/>
</DisjointClasses>

<SubObjectPropertyOf>
  <PropertyChain>
    <ObjectProperty IRI="hasFather"/>
    <ObjectProperty IRI="hasBrother"/>
  </PropertyChain>
  <ObjectProperty IRI="hasUncle"/>
</SubObjectPropertyOf>

<NegativeObjectPropertyAssertion>
  <ObjectProperty IRI="hasDaughter"/>
  <NamedIndividual IRI="Bill"/>
  <NamedIndividual IRI="Susan"/>
</NegativeObjectPropertyAssertion>
```

10.2 OWL 2 QL

OWL 2 QL can be realized using standard relational database technology (e.g., SQL) simply by expanding queries in the light of class axioms. This means it can be tightly integrated with RDBMSs and benefit from their robust implementations and multi-user features. Furthermore, it can be implemented without having to “touch the data,” so really as a translational/preprocessing layer. Expressively, it can represent key features of Entity-relationship and UML diagrams (at least those with functional restrictions). Thus, it is suitable both for representing database schemas and for integrating them via query rewriting. As a result, it can also be used directly as a high level database schema language, though users may prefer a diagram based syntax.

OWL 2 QL also captures many commonly used features in RDFS and small extensions thereof, such as inverse properties and subproperty hierarchies. OWL 2 QL restricts class axioms asymmetrically, that is, you can use constructs as the subclass that you cannot use as the superclass.

Among other constructs, OWL 2 QL disallows existential quantification of roles to a class expression, i.e. it can be stated that every person has a parent but not that every person has a female parent. Moreover property chain axioms are not supported.

The following is an example which uses some of the typical modeling features available in OWL 2 QL. The first axiom states that every childless person is a person for which there does not exist anybody who has the first person as parent.

Functional Style Syntax

```
SubClassOf(
 :ChildlessPerson
  ObjectIntersectionOf(
   :Person
    ObjectComplementOf(
      ObjectSomeValuesFrom(
        ObjectInverseOf(:hasParent )
        owl:Thing
      )
    )
  )
```

```
)

DisjointClasses(
 :Mother
 :Father
 :YoungChild
)

DisjointObjectProperties(
 :hasSon
 :hasDaughter
)

SubObjectPropertyOf(
 :hasFather
 :hasParent
)
```

RDF/XML Syntax

```
<owl:Class rdf:about="ChildlessPerson">
  <rdfs:subClassOf>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="Person"/>
        <owl:Class>
          <owl:complementOf>
            <owl:Restriction>
              <owl:onProperty>
                <owl:ObjectProperty>
                  <owl:inverseOf rdf:resource="hasParent"/>
                </owl:ObjectProperty>
              </owl:onProperty>
              <owl:someValuesFrom rdf:resource="Person"/>
            </owl:Restriction>
          </owl:complementOf>
        </owl:Class>
      </owl:intersectionOf>
    </owl:Class>
  </rdfs:subClassOf>
</owl:Class>

<owl:AllDisjointClasses>
  <owl:members rdf:parseType="Collection">
    <owl:Class rdf:about="Mother"/>
    <owl:Class rdf:about="Father"/>
    <owl:Class rdf:about="YoungChild"/>
  </owl:members>
</owl:AllDisjointClasses>

<owl:ObjectProperty rdf:about="hasSon">
  <owl:propertyDisjointWith rdf:resource="hasDaughter"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="hasFather">
  <rdfs:subPropertyOf rdf:resource="hasParent"/>
</owl:ObjectProperty>
```

Turtle Syntax

```
:ChildlessPerson  owl:subClassOf  [
```

```
   rdf:type            owl:Class;
   owl:intersectionOf  (:Person
                         [ owl:complementOf  [
                             rdf:type            owl:Restriction;
                             owl:onProperty      [ owl:inverseOf :hasParent ];
                             owl:someValuesFrom  owl:Thing
                           ]
                         ]
                       )
 ] .

 []  rdf:type    owl:AllDisjointClasses;
     owl:members (:Mother :Father :YoungChild ) .

:hasSon  owl:propertyDisjointWith :hasDaughter.

:hasFather  rdfs:subPropertyOf :hasParent.
```

Manchester Syntax

```
Class: ChildlessPerson
  SubClassOf: Person and not inverse hasParent some owl:Thing

DisjointClasses: Mother Father YoungChild

DisjointProperties: hasSon hasDaughter

ObjectProperty: hasFather
  SubPropertyOf: hasParent
```

OWL/XML Syntax

```
<SubClassOf>
  <Class IRI="ChildlessPerson"/>
  <ObjectIntersectionOf>
    <Class IRI="Person"/>
    <ObjectComplementOf>
      <ObjectSomeValuesFrom>
        <InverseObjectProperty>
          <ObjectProperty IRI="hasParent"/>
        </InverseObjectProperty>
        <Class abbreviatedIRI="owl:Thing"/>
      </ObjectSomeValuesFrom>
    </ObjectComplementOf>
  </ObjectIntersectionOf>
</SubClassOf>

<DisjointClasses>
    <Class IRI="Father"/>
    <Class IRI="Mother"/>
    <Class IRI="YoungChild"/>
</DisjointClasses>

<DisjointObjectProperties>
  <ObjectProperty IRI="hasSon"/>
  <ObjectProperty IRI="hasDaughter"/>
</DisjointObjectProperties>

<SubObjectPropertyOf>
  <ObjectProperty IRI="hasFather"/>
  <ObjectProperty IRI="hasParent"/>
```

```
</SubObjectPropertyOf>
```

10.3 OWL 2 RL

The OWL 2 RL profile is aimed at applications that require scalable reasoning without sacrificing too much expressive power. It is designed to accommodate both OWL 2 applications that can trade the full expressivity of the language for efficiency, and RDF(S) applications that need some added expressivity from OWL 2. This is achieved by defining a syntactic subset of OWL 2 which is amenable to implementation using rule-based technologies, and presenting a partial axiomatization of the OWL 2 RDF-Based Semantics in the form of first-order implications that can be used as the basis for such an implementation.

Suitable rule-based implementations of OWL 2 RL can be used with arbitrary RDF graphs. As a consequence, OWL 2 RL is ideal for enriching RDF data, especially when the data must be massaged by additional rules. From a modeling perspective, however, this pushes us farther away from working with class expressions: OWL 2 RL ensures we cannot (easily) talk about unknown individuals in our superclass expressions (this restriction follows from the nature of rules). Compared with OWL 2 QL, OWL 2 RL works better when you have already massaged your data into RDF and are working with it as RDF.

Among other constructs, OWL 2 RL disallows statements where the existence of an individual enforces the existence of another individual: for instance, the statement “every person has a parent” is not expressible in OWL RL.

OWL 2 RL restricts class axioms asymmetrically, that is, you can use constructs as the subclass that you cannot use as the superclass. The following is an example which uses some of the typical modeling features available in OWL 2 RL. The first – somewhat contrived – axiom states that for each of Mary, Bill, and Meg who is female, the following holds: she is a parent with at most one child, and all her children (if she has any) are female.

Functional Style Syntax

```
SubClassOf(
  ObjectIntersectionOf(
    ObjectOneOf(:Mary:Bill:Meg )
   :Female
  )
  ObjectIntersectionOf(
   :Parent
    ObjectMaxCardinality( 1:hasChild )
    ObjectAllValuesFrom(:hasChild:Female )
  )
)

DisjointClasses(
 :Mother
 :Father
 :YoungChild
)

SubObjectPropertyOf(
```

```
  ObjectPropertyChain(:hasFather:hasBrother )
 :hasUncle
)
```

RDF/XML Syntax

```
<owl:Class>
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class>
      <owl:oneOf rdf:parseType="Collection">
        <rdf:Description rdf:about="Mary"/>
        <rdf:Description rdf:about="Bill"/>
        <rdf:Description rdf:about="Meg"/>
      </owl:oneOf>
    </owl:Class>
    <owl:Class rdf:about="Female"/>
  </owl:intersectionOf>
  <rdfs:subClassOf>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="Parent"/>
        <owl:Restriction>
          <owl:maxCardinality rdf:datatype="&xsd;nonNegativeInteger">
            2
          </owl:maxCardinality>
          <owl:onProperty rdf:resource="hasChild"/>
        </owl:Restriction>
        <owl:Restriction>
          <owl:onProperty rdf:resource="hasChild"/>
          <owl:allValuesFrom rdf:resource="Female"/>
        </owl:Restriction>
      </owl:intersectionOf>
    </owl:Class>
  </rdfs:subClassOf>
</owl:Class>

<owl:AllDisjointClasses>
  <owl:members rdf:parseType="Collection">
    <owl:Class rdf:about="Mother"/>
    <owl:Class rdf:about="Father"/>
    <owl:Class rdf:about="YoungChild"/>
  </owl:members>
</owl:AllDisjointClasses>

<rdf:Description rdf:about="hasUncle">
  <owl:propertyChainAxiom rdf:parseType="Collection">
    <owl:ObjectProperty rdf:about="hasFather"/>
    <owl:ObjectProperty rdf:about="hasBrother"/>
  </owl:propertyChainAxiom>
</rdf:Description>
```

Turtle Syntax

```
[]  rdf:type            owl:Class;
    owl:intersectionOf  ( [ rdf:type   owl:Class;
                            owl:oneOf  (:Mary :Bill :Meg ) ]
                          :Female
                        );
    rdfs:subClassOf     [
      rdf:type            owl:Class;
      owl:intersectionOf  (:Parent
```

```
                              [ rdf:type            owl:Restriction;
                                owl:maxCardinality  "1"^^xsd:nonNegativeInteger;
                                owl:onProperty     :hasChild ]
                              [ rdf:type           owl:Restriction;
                                owl:onProperty    :hasChild;
                                owl:allValuesFrom :Female ]
                            )
    ] .

[]  rdf:type     owl:AllDisjointClasses;
    owl:members  (:Mother :Father :YoungChild ) .

:hasUncle  owl:propertyChainAxiom  (:hasFather :hasBrother ) .
```

Manchester Syntax

```
Class: X
  SubClassOf: Parent and hasChild max 1 and hasChild only Female
Class: X
  EquivalentTo: {Mary Bill Meg} and Female

DisjointClasses: Mother Father YoungChild

ObjectProperty: hasUncle
  SubPropertyChain: hasFather o hasBrother
```

OWL/XML Syntax

```
<SubClassOf>
  <ObjectIntersectionOf>
    <ObjectOneOf>
      <NamedIndividual IRI="Mary"/>
      <NamedIndividual IRI="Bill"/>
      <NamedIndividual IRI="Meg"/>
    </ObjectOneOf>
    <Class IRI="Female"/>
  </ObjectIntersectionOf>
  <ObjectIntersectionOf>
    <Class IRI="Parent"/>
    <ObjectMaxCardinality cardinality="1">
      <ObjectProperty IRI="hasChild"/>
    </ObjectMaxCardinality>
    <ObjectAllValuesFrom>
      <ObjectProperty IRI="hasChild"/>
      <Class IRI="Female"/>
    </ObjectAllValuesFrom>
  </ObjectIntersectionOf>
</SubClassOf>

<DisjointClasses>
    <Class IRI="Father"/>
    <Class IRI="Mother"/>
    <Class IRI="YoungChild"/>
</DisjointClasses>

<SubObjectPropertyOf>
  <PropertyChain>
    <ObjectProperty IRI="hasFather"/>
    <ObjectProperty IRI="hasBrother"/>
  </PropertyChain>
  <ObjectProperty IRI="hasUncle"/>
```

```
</SubObjectPropertyOf>
```

11 OWL Tools

Editor's Note: Writing this part has been deferred to incorporate reference implementations. It will also contain a description of tool categories (reasoners, editors, etc.)

12 What To Read Next

This short primer can only scratch the surface of OWL. There are many longer and more involved tutorials on OWL and how to use OWL tools that can be found by searching on the Web. Reading one of these documents and using a tool to build an OWL ontology is probably the best way to learn more about OWL.

This short primer is also not a normative definition of OWL. The normative definition of the OWL syntax as well as informative descriptions of the meaning of each OWL construct can be found in the OWL 2 Structural Specification and Functional Syntax document [OWL 2 Specification].

The OWL 2 Quick Reference Guide [*OWL 2 Quick Guide*] comes handy as a reference when looking for information about a specific language feature.

For those interested in more formal documents, the formal meaning of OWL 2 can be found in the OWL 2 Semantics documents: [*OWL 2 Direct Semantics*] and [*OWL 2 RDF-Based Semantics*].

The mapping between OWL syntax and RDF triples can be found in the OWL 2 Mapping to RDF Graphs document [*OWL 2 RDF Mapping*].

13 Appendix: The Complete Sample Ontology

Here we include the complete sample OWL ontology. Ontological axioms are ordered by top-level expressive features they use. Moreover, we follow a commonly-used ordering, with ontology and declaration information first, followed by information about properties, then classes and datatypes, then individuals.

Editor's Note: The presentation/formatting of Turtle, Manchester, and OWL/XML syntax versions still needs manual improvement. Functional style and RDF/XML are complete.

Functional Style Syntax

```
Prefix(:=<http://example.com/owl/families/>)
Prefix(otherOnt:=<http://example.org/otherOntologies/families/>)
Prefix(xsd:=<http://www.w3.org/2001/XMLSchema#>)
Prefix(owl:=<http://www.w3.org/2002/07/owl#>)
```

```
Ontology(<http://example.com/owl/families>
  Import( http://example.org/otherOntologies/families.owl )

  Declaration( NamedIndividual(:John ) )
  Declaration( NamedIndividual(:Mary ) )
  Declaration( NamedIndividual(:Jim ) )
  Declaration( NamedIndividual(:James ) )
  Declaration( NamedIndividual(:Jack ) )
  Declaration( NamedIndividual(:Bill ) )
  Declaration( NamedIndividual(:Susan ) )
  Declaration( Class(:Person ) )
  AnnotationAssertion( rdfs:comment:Person "Represents the set of all people." )
  Declaration( Class(:Woman ) )
  Declaration( Class(:Parent ) )
  Declaration( Class(:Father ) )
  Declaration( Class(:Mother ) )
  Declaration( Class(:SocialRole ) )
  Declaration( Class(:Man ) )
  Declaration( Class(:Teenager ) )
  Declaration( Class(:ChildlessPerson ) )
  Declaration( Class(:Human ) )
  Declaration( Class(:Female ) )
  Declaration( Class(:HappyPerson ) )
  Declaration( Class(:JohnsChildren ) )
  Declaration( Class(:NarcisticPerson ) )
  Declaration( Class(:MyBirthdayGuests ) )
  Declaration( Class(:Dead ) )
  Declaration( Class(:Orphan ) )
  Declaration( Class(:Adult ) )
  Declaration( Class(:YoungChild ) )
  Declaration( ObjectProperty(:hasWife ) )
  Declaration( ObjectProperty(:hasChild ) )
  Declaration( ObjectProperty(:hasDaughter ) )
  Declaration( ObjectProperty(:loves ) )
  Declaration( ObjectProperty(:hasSpouse ) )
  Declaration( ObjectProperty(:hasGrandparent ) )
  Declaration( ObjectProperty(:hasParent ) )
  Declaration( ObjectProperty(:hasBrother ) )
  Declaration( ObjectProperty(:hasUncle ) )
  Declaration( ObjectProperty(:hasSon ) )
  Declaration( ObjectProperty(:hasAncestor ) )
  Declaration( ObjectProperty(:hasHusband ) )
  Declaration( DataProperty(:hasAge ) )
  Declaration( DataProperty(:hasSSN ) )
  Declaration( Datatype(:personAge ) )
  Declaration( Datatype(:majorAge ) )
  Declaration( Datatype(:toddlerAge ) )

  SubObjectPropertyOf(:hasWife:hasSpouse )
  SubObjectPropertyOf(
    ObjectPropertyChain(:hasParent:hasParent )
   :hasGrandparent
  )
  SubObjectPropertyOf(
    ObjectPropertyChain(:hasFather:hasBrother )
   :hasUncle
  )
  SubObjectPropertyOf(
   :hasFather
   :hasParent
  )
```

```
EquivalentObjectProperties(:hasChild otherOnt:child )
InverseObjectProperties(:hasParent:hasChild )
EquivalentDataProperties(:hasAge otherOnt:age )
DisjointObjectProperties(:hasSon:hasDaughter )
ObjectPropertyDomain(:hasWife:Man )
ObjectPropertyRange(:hasWife:Woman )
DataPropertyDomain(:hasAge:Person )
DataPropertyRange(:hasAge xsd:NonNegativeInteger )

SymmetricObjectProperty(:hasSpouse )
AsymmetricObjectProperty(:hasChild )
DisjointObjectProperties(:hasParent:hasSpouse )
ReflexiveObjectProperty(:hasRelative )
IrreflexiveObjectProperty(:parentOf )
FunctionalObjectProperty(:hasHusband )
InverseFunctionalObjectProperty(:hasHusband )
TransitiveObjectProperty(:hasAncestor )
FunctionalDataProperty(:hasAge )

SubClassOf(:Woman:Person )
SubClassOf(:Mother:Woman )
SubClassOf(
 :Grandfather
  ObjectIntersectionOf(:Man:Parent )
)
SubClassOf(
 :Teenager
  ObjectSomeValuesFrom(:hasAge
    DatatypeRestriction( xsd:integer
      xsd:minInclusive "13"^^xsd:integer
      xsd:maxInclusive "19"^^xsd:integer
    )
  )
)
SubClassOf(
  Annotation( rdfs:comment "States that every man is a person." )
 :Man
 :Person
)
SubClassOf(
 :Father
  ObjectIntersectionOf(:Man:Parent )
)
SubClassOf(
 :ChildlessPerson
  ObjectIntersectionOf(
   :Person
    ObjectComplementOf(
      ObjectSomeValuesFrom(
        ObjectInverseOf(:hasParent )
        owl:Thing
      )
    )
  )
)
SubClassOf(
  ObjectIntersectionOf(
    ObjectOneOf(:Mary:Bill:Meg )
   :Female
  )
  ObjectIntersectionOf(
   :Parent
```

```
      ObjectMaxCardinality( 1:hasChild )
      ObjectAllValuesFrom(:hasChild:Female )
    )
  )

  EquivalentClasses(:Person:Human )
  EquivalentClasses(
   :Mother
    ObjectIntersectionOf(:Woman:Parent )
  )
  EquivalentClasses(
   :Parent
    ObjectUnionOf(:Mother:Father )
  )
  EquivalentClasses(
   :ChildlessPerson
    ObjectIntersectionOf(
     :Person
      ObjectComplementOf(:Parent )
    )
  )
  EquivalentClasses(
   :Parent
    ObjectSomeValuesFrom(:hasChild:Person )
  )
  EquivalentClasses(
   :HappyPerson
    ObjectIntersectionOf(
       ObjectAllValuesFrom(:hasChild:HappyPerson )
       ObjectSomeValuesFrom(:hasChild:HappyPerson )
    )
  )
  EquivalentClasses(
   :JohnsChildren
    ObjectHasValue(:hasParent:John )
  )
  EquivalentClasses(
   :NarcisticPerson
    ObjectHasSelf(:loves )
  )
  EquivalentClasses(
   :MyBirthdayGuests
    ObjectOneOf(:Bill:John:Mary)
  )
  EquivalentClasses(
   :Orphan
    ObjectAllValuesFrom(
      ObjectInverseOf(:hasChild )
     :Dead
    )
  )
  EquivalentClasses(:Adult otherOnt:Grownup )
  EquivalentClass(
   :Parent
    ObjectSomeValuesFrom(
     :hasChild
     :Person
    )
  )

  DisjointClasses(:Woman:Man )
  DisjointClasses(
```

```
 :Mother
 :Father
 :YoungChild
)
HasKey(:Person () (:hasSSN ) )

DatatypeDefinition(
 :personAge
  DatatypeRestriction( xsd:integer
    xsd:minInclusive "0"^^xsd:integer
    xsd:maxInclusive "150"^^xsd:integer
  )
)
DatatypeDefinition(
 :majorAge
  DataIntersectionOf(
   :personAge
    DataComplementOf(:minorAge )
  )
)
DatatypeDefinition(
 :toddlerAge
  DataOneOf( "1"^^xsd:integer "2"^^xsd:integer )
)

ClassAssertion(:Person:Mary )
ClassAssertion(:Woman:Mary )
ClassAssertion(
  ObjectIntersectionOf(
   :Person
    ObjectComplementOf(:Parent )
  )
 :John
)
ClassAssertion(
  ObjectMaxCardinality( 4:hasChild:Parent )
 :John
)
ClassAssertion(
  ObjectMinCardinality( 2:hasChild:Parent )
 :John
)
ClassAssertion(
  ObjectExactCardinality( 3:hasChild:Parent )
 :John
)
ClassAssertion(
  ObjectExactCardinality( 5:hasChild )
 :John
)
ClassAssertion(:Father:John )
ClassAssertion(:SocialRole:Father )

ObjectPropertyAssertion(:hasWife:John:Mary )
NegativeObjectPropertyAssertion(:hasWife:Bill:Mary )
NegativeObjectPropertyAssertion(
 :hasDaughter
 :Bill
 :Susan
)
DataPropertyAssertion(:hasAge:John "51"^^xsd:integer )
NegativeDataPropertyAssertion(:hasAge:Jack "53"^^xsd:integer )
```

```
   SameIndividual(:John:Jack )
   SameIndividual(:John otherOnt:JohnBrown )
   SameIndividual(:Mary otherOnt:MaryBrown )
   DifferentIndividuals(:John:Bill )
 )
```

RDF/XML Syntax

```
<!DOCTYPE rdf:RDF [
    <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
    <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
    <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
    <!ENTITY otherOnt "http://example.org/otherOntologies/families/" >
]>

 <rdf:RDF xml:base="http://example.com/owl/families/"
   xmlns="http://example.com/owl/families/"
   xmlns:otherOnt="http://example.org/otherOntologies/families/"
   xmlns:owl="http://www.w3.org/2002/07/owl#"
   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema#">

   <owl:Ontology rdf:about="http://example.com/owl/families">
     <owl:imports rdf:resource="http://example.org/otherOntologies/families.owl" />
   </owl:Ontology>


   <owl:ObjectProperty rdf:about="hasWife">
     <rdfs:subPropertyOf rdf:resource="hasSpouse"/>
     <rdfs:domain rdf:resource="Man"/>
     <rdfs:range rdf:resource="Woman"/>
   </owl:ObjectProperty>

   <owl:ObjectProperty rdf:about="hasParent">
     <owl:inverseOf rdf:resource="hasChild"/>
     <owl:propertyDisjointWith rdf:resource="hasSpouse"/>
   </owl:ObjectProperty>

   <owl:ObjectProperty rdf:about="hasSon">
     <owl:propertyDisjointWith rdf:resource="hasDaughter"/>
   </owl:ObjectProperty>

   <owl:ObjectProperty rdf:about="hasFather">
     <rdfs:subPropertyOf rdf:resource="hasParent"/>
   </owl:ObjectProperty>

   <owl:SymmetricProperty rdf:about="hasSpouse"/>
   <owl:AsymmetricProperty rdf:about="hasChild"/>
   <owl:ReflexiveProperty rdf:about="hasRelative"/>
   <owl:IrreflexiveProperty rdf:about="parentOf"/>
   <owl:FunctionalProperty rdf:about="hasHusband"/>
   <owl:InverseFunctionalProperty rdf:about="hasHusband"/>
   <owl:TransitiveProperty rdf:about="hasAncestor"/>

   <rdf:Description rdf:about="hasGrandparent">
     <owl:propertyChainAxiom rdf:parseType="Collection">
       <owl:ObjectProperty rdf:about="hasParent"/>
       <owl:ObjectProperty rdf:about="hasParent"/>
     </owl:propertyChainAxiom>
```

```
  </rdf:Description>

  <rdf:Description rdf:about="hasUncle">
    <owl:propertyChainAxiom rdf:parseType="Collection">
      <owl:ObjectProperty rdf:about="hasFather"/>
      <owl:ObjectProperty rdf:about="hasBrother"/>
    </owl:propertyChainAxiom>
  </rdf:Description>

  <owl:DatatypeProperty rdf:about="hasAge">
    <rdfs:domain rdf:resource="Person"/>
    <rdfs:range rdf:datatpye="http://www.w3.org/2001/XMLSchema#NonNegativeInteger"/>
    <owl:equivalentProperty rdf:resource="&otherOnt;age"/>
  </owl:DatatypeProperty>
  <owl:FunctionalProperty rdf:about="hasAge"/>


  <owl:Class rdf:about="Woman">
    <rdfs:subClassOf rdf:resource="Person"/>
  </owl:Class>

  <owl:Class rdf:about="Mother">
    <rdfs:subClassOf rdf:resource="Woman"/>
    <owl:equivalentClass>
      <owl:Class>
        <owl:intersectionOf rdf:parseType="Collection">
          <owl:Class rdf:about="Woman"/>
          <owl:Class rdf:about="Parent"/>
        </owl:intersectionOf>
      </owl:Class>
    </owl:equivalentClass>
  </owl:Class>

  <owl:Class rdf:about="Person">
    <rdfs:comment>Represents the set of all people.</rdfs:comment>
    <owl:equivalentClass rdf:resource="Human"/>
    <owl:hasKey rdf:parseType="Collection">
      <owl:ObjectProperty rdf:about="hasSSN"/>
    </owl:hasKey>
  </owl:Class>

  <owl:Class rdf:about="Parent">
    <owl:equivalentClass>
      <owl:Class>
        <owl:unionOf rdf:parseType="Collection">
          <owl:Class rdf:about="Mother"/>
          <owl:Class rdf:about="Father"/>
        </owl:unionOf>
      </owl:Class>
    </owl:equivalentClass>
    <owl:equivalentClass>
      <owl:Restriction>
        <owl:onProperty rdf:resource="hasChild"/>
        <owl:someValuesFrom rdf:resource="Person"/>
      </owl:Restriction>
    </owl:equivalentClass>
  </owl:Class>

  <owl:Class rdf:about="ChildlessPerson">
    <owl:equivalentClass>
      <owl:Class>
        <owl:intersectionOf rdf:parseType="Collection">
```

```
        <owl:Class rdf:about="Person"/>
        <owl:Class>
          <owl:complementOf rdf:resource="Parent"/>
        </owl:Class>
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>

<owl:Class rdf:about="Grandfather">
  <rdfs:subClassOf>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="Man"/>
        <owl:Class rdf:about="Parent"/>
      </owl:intersectionOf>
    </owl:Class>
  </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:about="HappyPerson">
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf parseType="Collection">
        <owl:Restriction>
          <owl:onProperty rdf:about="hasChild"/>
          <owl:allValuesFrom rdf:about="HappyPerson"/>
        </owl:Restriction>
        <owl:Restriction>
          <owl:onProperty rdf:about="hasChild"/>
          <owl:someValuesFrom rdf:about="HappyPerson"/>
        </owl:Restriction>
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>

<owl:Class rdf:about="JohnsChildren">
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="hasParent"/>
      <owl:hasValue rdf:resource="John"/>
    </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>

<owl:Class rdf:about="NarcisticPerson">
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="loves"/>
      <owl:hasSelf rdf:datatype="&xsd;boolean">
        true
      </owl:hasSelf>
    </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>

<owl:Class rdf:about="MyBirthdayGuests">
  <owl:equivalentClass>
    <owl:Class>
      <owl:oneOf rdf:parseType="Collection">
        <rdf:Description rdf:about="Bill"/>
```

```
        <rdf:Description rdf:about="John"/>
        <rdf:Description rdf:about="Mary"/>
      </owl:oneOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>

<owl:Class rdf:about="Orphan">
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty>
        <owl:ObjectProperty>
          <owl:inverseOf rdf:about="hasChild"/>
        </owl:ObjectProperty>
      </owl:onProperty>
      <owl:Class rdf:about="Dead"/>
    </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>

<owl:Class rdf:about="Teenager">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="hasAge"/>
      <owl:someValuesFrom>
        <rdfs:Datatype>
          <owl:onDataType rdf:resource="&xsd;integer"/>
          <owl:withRestrictions rdf:parseType="Collection">
            <xsd:minExclusive rdf:datatype="&xsd;integer">
              12
            </xsd:minExclusive>
            <xsd:maxInclusive rdf:datatype="&xsd;integer">
              19
            </xsd:maxInclusive>
          </owl:withRestrictions>
        </rdfs:Datatype>
      </owl:someValuesFrom>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:about="Man">
  <rdfs:subClassOf rdf:resource="Person"/>
</owl:Class>
<owl:Axiom>
  <owl:annotatedSource rdf:resource="Man"/>
  <owl:annotatedProperty rdf:resource="&rdfs;subClassOf"/>
  <owl:annotatedTarget rdf:resource="Person"/>
  <rdfs:comment>States that every man is a person.</rdfs:comment>
</owl:Axiom>

<owl:Class rdf:about="Adult">
  <owl:equivalentClass rdf:resource="&otherOnt;Grownup"/>
</owl:Class>

<owl:Class rdf:about="Father">
  <rdfs:subClassOf>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="Man"/>
        <owl:Class rdf:about="Parent"/>
      </owl:intersectionOf>
```

```
      </owl:Class>
    </rdfs:subClassOf>
  </owl:Class>

  <owl:Class rdf:about="ChildlessPerson">
    <rdfs:subClassOf>
      <owl:Class>
        <owl:intersectionOf rdf:parseType="Collection">
          <owl:Class rdf:about="Person"/>
          <owl:Class>
            <owl:complementOf>
              <owl:Restriction>
                <owl:onProperty>
                  <owl:ObjectProperty>
                    <owl:inverseOf rdf:resource="hasParent"/>
                  </owl:ObjectProperty>
                </owl:onProperty>
                <owl:someValuesFrom rdf:resource="&owl;Thing"/>
              </owl:Restriction>
            </owl:complementOf>
          </owl:Class>
        </owl:intersectionOf>
      </owl:Class>
    </rdfs:subClassOf>
  </owl:Class>

  <owl:Class>
    <owl:intersectionOf rdf:parseType="Collection">
      <owl:Class>
        <owl:oneOf rdf:parseType="Collection">
          <rdf:Description rdf:about="Mary"/>
          <rdf:Description rdf:about="Bill"/>
          <rdf:Description rdf:about="Meg"/>
        </owl:oneOf>
      </owl:Class>
      <owl:Class rdf:about="Female"/>
    </owl:intersectionOf>
    <rdfs:subClassOf>
      <owl:Class>
        <owl:intersectionOf rdf:parseType="Collection">
          <owl:Class rdf:about="Parent"/>
          <owl:Restriction>
            <owl:maxCardinality rdf:datatype="&xsd;nonNegativeInteger">
              2
            </owl:maxCardinality>
            <owl:onProperty rdf:resource="hasChild"/>
          </owl:Restriction>
          <owl:Restriction>
            <owl:onProperty rdf:resource="hasChild"/>
            <owl:allValuesFrom rdf:resource="Female"/>
          </owl:Restriction>
        </owl:intersectionOf>
      </owl:Class>
    </rdfs:subClassOf>
  </owl:Class>

  <owl:AllDisjointClasses>
    <owl:members rdf:parseType="Collection">
      <owl:Class rdf:about="Woman"/>
      <owl:Class rdf:about="Man"/>
    </owl:members>
  </owl:AllDisjointClasses>
```

```
   <owl:AllDisjointClasses>
     <owl:members rdf:parseType="Collection">
       <owl:Class rdf:about="Mother"/>
       <owl:Class rdf:about="Father"/>
       <owl:Class rdf:about="YoungChild"/>
     </owl:members>
   </owl:AllDisjointClasses>


   <rdf:Description rdf:about="personAge">
     <owl:equivalentClass>
       <owl:Datatype>
         <owl:onDatatype rdf:resource="http://www.w3.org/2001/XMLSchema#integer"/>
         <owl:withRestrictions rdf:parseType="Collection">
           <xsd:minInclusive
rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">0</xsd:minInclusive>
           <xsd:maxInclusive
rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">150</xsd:maxInclusive>
         </owl:withRestrictions>
       </owl:Datatype>
     </owl:equivalentClass>
   </rdf:Description>

   <rdf:Description rdf:about="majorAge">
     <owl:equivalentClass>
       <owl:Datatype>
         <owl:intersectionOf rdf:parseType="Collection">
           <rdf:Description rdf:about="personAge"/>
           <rdfs:Datatype>
             <owl:datatypeComplementOf about="minorAge"/>
           </rdfs:Datatype>
         </owl:intersectionOf>
       </owl:Datatype>
     </owl:equivalentClass>
   </rdf:Description>

   <rdf:Description rdf:about="toddlerAge">
     <owl:equivalentClass>
       <owl:Datatype>
         <owl:oneOf rdf:parseType="Collection">
           <rdf:Description rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">1</
rdf:Description>
           <rdf:Description rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">2</
rdf:Description>
         </owl:oneOf>
       </owl:Datatype>
     </owl:equivalentClass>
   </rdf:Description>


   <Person rdf:about="Mary">
     <rdf:type rdf:resource="Woman"/>
     <owl:sameAs rdf:resource="&otherOnt;MaryBrown"/>
   </Person>

   <owl:NamedIndividual rdf:about="James">
     <owl:sameAs rdf:resource="Jim"/>
   </owl:NamedIndividual>

   <owl:NamedIndividual rdf:about="John">
     <hasWife rdf:resource="Mary"/>
```

```
    <hasAge rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">51</hasAge>
    <owl:differentFrom rdf:resource="Bill"/>
    <owl:sameAs rdf:resource="&otherOnt;JohnBrown"/>
    <rdf:type rdf:resource="Person"/>
    <rdf:type rdf:resource="Father"/>
    <rdf:type>
      <owl:Class>
        <owl:intersectionOf  rdf:parseType="Collection">
          <owl:Class rdf:about="Person"/>
          <owl:Class>
            <owl:complementOf rdf:about="Parent"/>
          </owl:Class>
        </owl:intersectionOf>
      </owl:Class>
    </rdf:type>
    <rdf:type>
      <owl:Class>
        <owl:Restriction>
          <owl:maxQualifiedCardinality rdf:datatype="&xsd;nonNegativeInteger">
            4
          </owl:maxQualifiedCardinality>
          <owl:onProperty rdf:about="hasChild"/>
          <owl:onClass rdf:about="Parent"/>
        </owl:Restriction>
      </owl:Class>
    </rdf:type>
    <rdf:type>
      <owl:Class>
        <owl:Restriction>
          <owl:minQualifiedCardinality rdf:datatype="&xsd;nonNegativeInteger">
            2
          </owl:minQualifiedCardinality>
          <owl:onProperty rdf:about="hasChild"/>
          <owl:onClass rdf:about="Parent"/>
        </owl:Restriction>
      </owl:Class>
    </rdf:type>
    <rdf:type>
      <owl:Class>
        <owl:Restriction>
          <owl:qualifiedCardinality rdf:datatype="&xsd;nonNegativeInteger">
            3
          </owl:qualifiedCardinality>
          <owl:onProperty rdf:about="hasChild"/>
          <owl:onClass rdf:about="Parent"/>
        </owl:Restriction>
      </owl:Class>
    </rdf:type>
    <rdf:type>
      <owl:Class>
        <owl:Restriction>
          <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">
            5
          </owl:cardinality>
          <owl:onProperty rdf:about="hasChild"/>
        </owl:Restriction>
      </owl:Class>
    </rdf:type>
  </owl:NamedIndividual>

  <SocialRole rdf:about="Father"/>
```

```
   <owl:NegativePropertyAssertion>
     <owl:sourceIndividual rdf:about="Bill"/>
     <owl:assertionProperty rdf:about="hasWife"/>
     <owl:targetIndividual rdf:about="Mary"/>
   </owl:NegativePropertyAssertion>

   <owl:NegativePropertyAssertion>
     <owl:sourceIndividual rdf:about="Jack"/>
     <owl:assertionProperty rdf:about="hasAge"/>
     <owl:targetValue
rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">53</owl:targetValue>
   </owl:NegativePropertyAssertion>

   <owl:NegativePropertyAssertion>
     <owl:sourceIndividual rdf:about="Bill"/>
     <owl:assertionProperty rdf:about="hasDaughter"/>
     <owl:targetIndividual rdf:about="Susan"/>
   </owl:NegativePropertyAssertion>
 </rdf:RDF>
```

Turtle Syntax

```
@prefix: <http://example.com/owl/families/> .
@prefix otherOnt: <http://example.org/otherOntologies/families/> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://example.com/owl/families>
     rdf:type owl:Ontology;
     owl:imports <http://example.org/otherOntologies/families.owl> .


:Mary rdf:type:Person .
:Mary rdf:type:Woman .
:Woman rdfs:subClassOf:Person .
:Mother rdfs:subClassOf:Woman .
:Person owl:equivalentClass:Human .
[]  rdf:type     owl:AllDisjointClasses;
    owl:members  (:Woman :Man ) .
:John:hasWife:Mary .
[]  rdf:type               owl:NegativePropertyAssertion;
    owl:sourceIndividual  :Bill;
    owl:assertionProperty :hasWife;
    owl:targetIndividual  :Mary .
:hasWife rdfs:subPropertyOf:hasSpouse .
:hasWife rdfs:domain:Man;
         rdfs:range :Woman .
:John  owl:differentFrom:Bill .
:James owl:sameAs:Jim .
:John :hasAge  51 .
[]  rdf:type               owl:NegativePropertyAssertion;
    owl:sourceIndividual  :Jack;
    owl:assertionProperty :hasAge;
    owl:targetValue        53 .
:hasAge  rdfs:domain :Person;
         rdfs:range   xsd:NonNegativeInteger .
:Mother  owl:equivalentClass  [
  rdf:type            owl:Class;
  owl:intersectionOf  (:Woman:Parent )
```

```
] .
:Parent  owl:equivalentClass  [
  rdf:type     owl:Class;
  owl:unionOf  (:Mother:Father )
] .
:ChildlessPerson  owl:equivalentClass  [
  rdf:type            owl:Class;
  owl:intersectionOf  (:Person  [ owl:complementOf :Parent ] )
] .
:Grandfather  rdfs:subClassOf  [
  rdf:type            owl:Class;
  owl:intersectionOf  (:Man :Parent )
] .
:John  rdf:type  [
  rdf:type            owl:Class;
  owl:intersectionOf  (:Person
                        [ rdf:type          owl:Class;
                          owl:complementOf :Parent     ]
                      )
] .
:Parent  owl:equivalentClass  [
  rdf:type            owl:Restriction;
  owl:onProperty     :hasChild;
  owl:someValuesFrom :Person
] .
:HappyPerson
    owl:equivalentClass  [
      rdf:type           owl:Restriction;
      owl:onProperty    :hasChild;
      owl:allValuesFrom :HappyPerson
    ] .
:HappyPerson
    owl:equivalentClass  [
      rdf:type            owl:Class;
      owl:intersectionOf  ( [ rdf:type            owl:Restriction;
                              owl:onProperty     :hasChild;
                              owl:allValuesFrom  :HappyPerson      ]
                            [ rdf:type            owl:Restriction;
                              owl:onProperty     :hasChild;
                              owl:someValuesFrom :HappyPerson      ]
                          )
    ] .
:JohnsChildren  owl:equivalentClass  [
  rdf:type        owl:Restriction;
  owl:onProperty :hasParent;
  owl:hasValue   :John
] .
:NarcisticPerson owl:equivalentClass  [
  rdf:type        owl:Restriction;
  owl:onProperty :loves;
  owl:hasSelf     "true"^^xsd:boolean .
] .
:John  rdf:type  [
  rdf:type                      owl:Restriction;
  owl:maxQualifiedCardinality  "4"^^xsd:nonNegativeInteger;
  owl:onProperty               :hasChild;
  owl:onClass                  :Parent
] .
:John  rdf:type  [
  rdf:type                      owl:Restriction;
  owl:minQualifiedCardinality  "2"^^xsd:nonNegativeInteger;
  owl:onProperty               :hasChild;
```

```
  owl:onClass                 :Parent
] .
:John  rdf:type  [
  rdf:type                   owl:Restriction;
  owl:qualifiedCardinality  "3"^^xsd:nonNegativeInteger;
  owl:onProperty           :hasChild;
  owl:onClass              :Parent
] .
:John  rdf:type  [
  rdf:type         owl:Restriction;
  owl:cardinality  "5"^^xsd:nonNegativeInteger;
  owl:onProperty  :hasChild
] .
:MyBirthdayGuests  owl:equivalentClass  [
  rdf:type   owl:Class;
  owl:oneOf  (:Bill :John :Mary )
] .
:hasParent owl:inverseOf:hasChild .
:Orphan  owl:equivalentClass  [
  rdf:type           owl:Restriction;
  owl:onProperty     [ owl:inverseOf :hasChild ];
  owl:allValuesFrom :Dead
] .
:hasSpouse  rdf:type  owl:SymmetricProperty .
:hasChild  rdf:type  owl:AsymmetricProperty .
:hasParent  owl:propertyDisjointWith :hasSpouse .
:hasRelative  rdf:type  owl:ReflexiveProperty .
:parentOf  rdf:type  owl:IrreflexiveProperty .
:hasHusband  rdf:type  owl:FunctionalProperty .
:hasHusband  rdf:type  owl:InverseFunctionalProperty .
:hasAncestor  rdf:type  owl:TransitiveProperty .
:hasGrandparent  owl:propertyChainAxiom  (:hasParent :hasParent ) .
:Person owl:hasKey (:hasSSN ) .
:personAge  owl:equivalentClass
 [ rdf:type rdfs:Datatype;
   owl:onDatatype xsd:Integer;
   owl:withRestrictions (
      [ xsd:minInclusive "0"^^xsd:integer ]
      [ xsd:maxInclusive "150"^^xsd:integer ]
   )
 ] .
:majorAge  owl:equivalentClass
  [ rdf:type rdfs:Datatype;
    owl:intersectionOf (
      :personAge
       [ rdf:type rdfs:Datatype;
         owl:datatypeComplementOf:minorAge ]
    )
  ] .
:toddlerAge  owl:equivalentClass
  [ rdf:type rdfs:Datatype;
    owl:oneOf (  "1"^^xsd:integer  "2"^^xsd:integer )
  ] .
:hasAge  rdf:type  owl:FunctionalProperty .
:Teenager  rdfs:subClassOf
      [ rdf:type             owl:Restriction;
        owl:onProperty      :hasAge;
        owl:someValuesFrom
         [ rdf:type             rdfs:Datatype;
           owl:onDatatype       xsd:integer;
           owl:withRestrictions (  [ xsd:minInclusive     "13"^^xsd:integer ]
                                   [ xsd:maxInclusive     "19"^^xsd:integer ]
```

```
          )
        ]
      ] .
:Person  rdfs:comment  "Represents the set of all people."^^xsd:string .
:Man rdfs:subClassOf:Person .
[]  rdf:type       owl:Axiom;
    owl:annotatedSource   :Man;
    owl:annotatedProperty  rdfs:subClassOf;
    owl:annotatedTarget   :Person;
    rdfs:comment     "States that every man is a person."^^xsd:string .

:Mary      owl:sameAs              otherOnt:MaryBrown .
:John      owl:sameAs              otherOnt:JohnBrown .
:Adult     owl:equivalentClass     otherOnt:Grownup .
:hasChild  owl:equivalentProperty  otherOnt:child .
:hasAge    owl:equivalentProperty  otherOnt:age .
:John    rdf:type owl:NamedIndividual .
:Person  rdf:type owl:Class .
:hasWife rdf:type owl:ObjectProperty .
:hasAge  rdf:type owl:DatatypeProperty .
:John rdf:type:Father .
:Father rdf:type:SocialRole .
:Father  rdfs:subClassOf  [
  rdf:type            owl:Class;
  owl:intersectionOf  (:Man :Parent )
] .

:Parent  owl:equivalentClass  [
  rdf:type            owl:Restriction;
  owl:onProperty     :hasChild;
  owl:someValuesFrom :Person
] .

:NarcisticPerson  owl:equivalentClass  [
  rdf:type        owl:Restriction;
  owl:onProperty :loves;
  owl:hasSelf     true
] .

[] rdf:type     owl:AllDisjointClasses;
   owl:members  (:Mother :Father :YoungChild ) .

:hasUncle  owl:propertyChainAxiom  (:hasFather :hasBrother ) .

[]  rdf:type               owl:NegativePropertyAssertion;
    owl:sourceIndividual  :Bill;
    owl:assertionProperty :hasDaughter;
    owl:targetIndividual  :Susan .
:ChildlessPerson  owl:subClassOf  [
  rdf:type            owl:Class;
  owl:intersectionOf  (:Person
                       [ owl:complementOf  [
                           rdf:type            owl:Restriction;
                           owl:onProperty      [ owl:inverseOf :hasParent ];
                           owl:someValuesFrom  owl:Thing
                         ]
                       ]
                      )
] .

:hasSon  owl:propertyDisjointWith :hasDaughter.
```

```
:hasFather  rdfs:subPropertyOf :hasParent.
[]  rdf:type            owl:Class;
    owl:intersectionOf  ( [ rdf:type   owl:Class;
                            owl:oneOf  (:Mary :Bill :Meg ) ]
                          :Female
                        );
    rdfs:subClassOf     [
      rdf:type            owl:Class;
      owl:intersectionOf  (:Parent
                            [ rdf:type            owl:Restriction;
                              owl:maxCardinality  "1"^^xsd:nonNegativeInteger;
                              owl:onProperty     :hasChild ]
                            [ rdf:type           owl:Restriction;
                              owl:onProperty    :hasChild;
                              owl:allValuesFrom :Female ]
                          )
    ] .
```

Manchester Syntax

```
Prefix: <http://example.com/owl/families/>
Prefix: owl <http://www.w3.org/2002/07/owl#>
Prefix: otherOnt <http://example.org/otherOntologies/families/>
Ontology: <http://example.com/owl/families>
Import: <http://example.org/otherOntologies/families.owl>

Individual: Mary
  Types: Person
Individual: Mary
  Types: Woman
Class: Woman
  SubClassOf: Person
Class: Mother
  SubClassOf: Woman
Class: Person
  EquivalentTo: Human
DisjointClasses: Woman Man
Individual: John
  Facts: hasWife Mary
Individual: Bill
  Facts: not hasWife Mary
ObjectProperty: hasWife
  SubPropertyOf: hasSpouse
ObjectProperty: hasWife
  Domain: Man
  Range: Woman
Individual: John
  DifferentFrom: Bill
Individual: James
  SameAs: Jim
Individual: John
  Facts: hasAge "51"^^xsd:integer
Individual: Jack
  Facts: not hasAge "53"^^xsd:integer
DatatpyeProperty: hasAge
  Domain: Person
  Range:  xsd:NonNegativeInteger
Class: Mother
  EquivalentTo: Woman and Parent
Class: Parent
  EquivalentTo: Mother or Father
```

```
Class: ChildlessPerson
  EquivalentTo: Person and not Parent
Class: Grandfather
  SubClassOf: Man and Parent
Individual: John
  Types: Person and not Parent
Class: Parent
  EquivalentTo: hasChild some Person
Class: HappyPerson
  EquivalentTo: hasChild only HappyPerson
Class: HappyPerson
  EquivalentTo: hasChild only Happy and hasChild some Happy
Class: JohnsChildren
  EquivalentTo: hasParent value John
Class: NarcisticPerson
  EquivalentTo: loves Self
Individual: John
  Types: hasChild max 4 Parent
Individual: John
  Types: hasChild min 2 Parent
Individual: John
  Types: hasChild exactly 3 Parent
Individual: John
  Types: hasChild exactly 5
Class: MyBirthdayGuests
  EquivalentTo: { Bill John Mary }
ObjectProperty: hasParent
  InverseOf: hasChild
Class: Orphan
  EquivalentTo: inverse hasChild only Dead
ObjectProperty: hasSpouse
  Characteristics: Symmetric
ObjectProperty: hasChild
  Characteristics: Asymmetric
DisjointProperties: hasParent hasSpouse
ObjectProperty: hasRelative
  Characteristics: Reflexive
Objectproperty: parentOf
  Characteristics: Irreflexive
ObjectProperty: hasHusband
  Characteristics: Functional
ObjectProperty: hasHusband
  Characteristics: InverseFunctional
ObjectProperty: hasAncestor
  Characteristics: Transitive
ObjectProperty: hasGrandparent
  SubPropertyChain: hasParent o hasParent
HasKey: Person hasSSN
Datatype: personAge
  EquivalentTo: integer[<= 0 , >= 150]
Datatype: majorAge
   EquivalentTo: personAge and not minorAge
Datatype: toddlerAge
   EquivalentTo: { 1 2 }
DataProperty: hasHusband
  Characteristics: Functional
Class: Teenager
 SubClassOf: hasAge some integer[<= 13 , >= 19]
Class: Person
  Annotations: rdfs:comment "Represents the set of all people."
Class: Man
  SubClassOf: Annotations: rdfs:comment "States that every man is a person." Person
```

```
SameIndividual: John otherOnt:JohnBrown
SameIndividual: Mary otherOnt:MaryBrown
EquivalentClasses: Adult otherOnt:Grownup
EquivalentProperties: hasChild otherOnt:child
EquivalentProperties: hasAge   otherOnt:age
Individual: John
Class: Person
ObjectProperty: hasWife
Dataproperty: hasAge
Individual: John
 Types: Father
Individual: Father
 Types: SocialRole
Class: Father
  SubClassOf: Man and Parent

Class: Parent
  EquivalentTo: hasChild some Person

Class: NarcisticPerson
  EquivalentTo: loves Self

DisjointClasses: Mother Father YoungChild

ObjectProperty: hasUncle
  SubPropertyChain: hasFather o hasBrother

Individual: Bill
  Facts: not hasDaughter Susan
Class: ChildlessPerson
  SubClassOf: Person and not inverse hasParent some owl:Thing

DisjointClasses: Mother Father YoungChild

DisjointProperties: hasSon hasDaughter

ObjectProperty: hasFather
  SubPropertyOf: hasParent
Class: X
  SubClassOf: Parent and hasChild max 1 and hasChild only Female
Class: X
  EquivalentTo: {Mary Bill Meg} and Female

DisjointClasses: Mother Father YoungChild

ObjectProperty: hasUncle
  SubPropertyChain: hasFather o hasBrother
Class: Citizen EquivalentTo: hasParent some Citizen
Individual: Sheevah Types: Citizen
 Individual: Sheevah Types: Citizen
    Facts: hasParent Suma
```

OWL/XML Syntax

```
<!DOCTYPE Ontology [
   <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
   <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
]>

 <Ontology
   xml:base="http://example.com/owl/families/"
```

```
xmlns="http://www.w3.org/2002/07/owl#"
xmlns:owl="http://www.w3.org/2002/07/owl#"
xmlns:g="http://example.org/otherOntologies/families/"
ontologyIRI="http://example.com/owl/families">
<Prefix name="otherOnt" IRI="http://example.org/otherOntologies/families/"/>
<Import>http://example.org/otherOntologies/families.owl</Import>

<ClassAssertion>
    <Class IRI="Person"/>
    <NamedIndividual IRI="Mary"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="Woman"/>
    <NamedIndividual IRI="Mary"/>
</ClassAssertion>
<SubClassOf>
  <Class IRI="Woman"/>
  <Class IRI="Person"/>
</SubClassOf>
<SubClassOf>
  <Class IRI="Mother"/>
  <Class IRI="Woman"/>
</SubClassOf>
<EquivalentClasses>
  <Class IRI="Person"/>
  <Class IRI="Human"/>
</EquivalentClasses>
<DisjointClasses>
    <Class IRI="Woman"/>
    <Class IRI="Man"/>
</DisjointClasses>
<ObjectPropertyAssertion>
  <ObjectProperty IRI="hasWife"/>
  <NamedIndividual IRI="John"/>
  <NamedIndividual IRI="Mary"/>
</ObjectPropertyAssertion>
<NegativeObjectPropertyAssertion>
  <ObjectProperty IRI="hasWife"/>
  <NamedIndividual IRI="Bill"/>
  <NamedIndividual IRI="Mary"/>
</NegativeObjectPropertyAssertion>
<SubObjectPropertyOf>
  <ObjectProperty IRI="hasWife"/>
  <ObjectProperty IRI="hasSpouse"/>
</SubObjectPropertyOf>
<ObjectPropertyDomain>
  <ObjectProperty IRI="hasWife"/>
  <Class IRI="Man"/>
</ObjectPropertyDomain>
<ObjectPropertyRange>
  <ObjectProperty IRI="hasWife"/>
  <Class IRI="Woman"/>
</ObjectPropertyRange>
<DifferentIndividuals>
  <NamedIndividual IRI="John"/>
  <NamedIndividual IRI="Bill"/>
</DifferentIndividuals>
<SameIndividual>
  <NamedIndividual IRI="James"/>
  <NamedIndividual IRI="Jim"/>
</SameIndividual>
<DataPropertyAssertion>
```

```
    <DataProperty IRI="hasAge"/>
    <NamedIndividual IRI="John"/>
    <Literal datatypeIRI="http://www.w3.org/2001/XMLSchema#integer">51</Literal>
  </DataPropertyAssertion>
  <NegativeDataPropertyAssertion>
    <DataProperty IRI="hasAge"/>
    <NamedIndividual IRI="Jack"/>
    <Literal datatypeIRI="http://www.w3.org/2001/XMLSchema#integer">53</Literal>
  </NegativeDataPropertyAssertion>
  <DatatypePropertyDomain>
    <DatatypeProperty IRI="hasAge"/>
    <Class IRI="Person"/>
  </DatatypePropertyDomain>
  <DatatypePropertyRange>
    <DatatypeProperty IRI="hasAge"/>
    <Datatype IRI="http://www.w3.org/2001/XMLSchema#NonNegativeInteger"/>
  </DatatypePropertyRange>
  <EquivalentClasses>
    <Class IRI="Mother"/>
    <ObjectIntersectionOf>
      <Class IRI="Woman"/>
      <Class IRI="Parent"/>
    </ObjectIntersectionOf>
  </EquivalentClasses>
  <EquivalentClasses>
    <Class IRI="Parent"/>
    <ObjectUnionOf>
      <Class IRI="Mother"/>
      <Class IRI="Father"/>
    </ObjectUnionOf>
  </EquivalentClasses>
  <EquivalentClasses>
    <Class IRI="ChildlessPerson"/>
    <ObjectIntersectionOf>
      <Class IRI="Person"/>
      <ObjectComplementOf>
        <Class IRI="Parent"/>
      </ObjectComplementOf>
    </ObjectIntersectionOf>
  </EquivalentClasses>
  <SubClassOf>
    <Class IRI="Grandfather"/>
    <ObjectIntersectionOf>
      <Class IRI="Man"/>
      <Class IRI="Parent"/>
    </ObjectIntersectionOf>
  </SubClassOf>
  <ClassAssertion>
    <ObjectIntersectionOf>
      <Class IRI="Person"/>
      <ObjectComplementOf>
        <Class IRI="Parent"/>
      </ObjectComplementOf>
    </ObjectIntersectionOf>
    <NamedIndividual IRI="John"/>
  </ClassAssertion>
  <EquivalentClasses>
    <Class IRI="Parent"/>
    <ObjectSomeValuesFrom>
      <ObjectProperty IRI="hasChild"/>
      <Class IRI="Person"/>
    </ObjectSomeValuesFrom>
```

```
  </EquivalentClasses>
  <EquivalentClasses>
    <Class IRI="HappyPerson"/>
    <ObjectAllValuesFrom>
      <ObjectProperty IRI="hasChild"/>
      <Class IRI="HappyPerson"/>
    </ObjectAllValuesFrom>
  </EquivalentClasses>
  <EquivalentClasses>
    <Class IRI="HappyPerson"/>
    <ObjectIntersectionOf>
      <ObjectAllValuesFrom>
        <ObjectProperty IRI="hasChild"/>
        <Class IRI="HappyPerson"/>
      </ObjectAllValuesFrom>
      <ObjectSomeValuesFrom>
        <ObjectProperty IRI="hasChild"/>
        <Class IRI="HappyPerson"/>
      </ObjectSomeValuesFrom>
    </ObjectIntersectionOf>
  </EquivalentClasses>
  <EquivalentClasses>
    <Class IRI="JohnsChildren"/>
    <ObjectHasValue>
      <ObjectProperty IRI="hasParent"/>
      <Class IRI="John"/>
    </ObjectHasValue>
  </EquivalentClasses>
  <EquivalentClasses>
    <Class IRI="NarcisticPerson"/>
    <ObjectHasSelf>
      <ObjectProperty IRI="loves"/>
    </ObjectHasSelf>
  </EquivalentClasses>
  <ClassAssertion>
    <ObjectMaxCardinality cardinality="4">
      <ObjectProperty IRI="hasChild"/>
      <Class IRI="Parent"/>
    </ObjectMaxCardinality>
    <NamedIndividual IRI="John"/>
  </ClassAssertion>
  <ClassAssertion>
    <ObjectMinCardinality cardinality="2">
      <ObjectProperty IRI="hasChild"/>
      <Class IRI="Parent"/>
    </ObjectMinCardinality>
    <NamedIndividual IRI="John"/>
  </ClassAssertion>
  <ClassAssertion>
    <ObjectExactCardinality cardinality="3">
      <ObjectProperty IRI="hasChild"/>
      <Class IRI="Parent"/>
    </ObjectExactCardinality>
    <NamedIndividual IRI="John"/>
  </ClassAssertion>
  <ClassAssertion>
    <ObjectExactCardinality cardinality="5">
      <ObjectProperty IRI="hasChild"/>
    </ObjectExactCardinality>
    <NamedIndividual IRI="John"/>
  </ClassAssertion>
  <EquivalentClasses>
```

```
    <Class IRI="MyBirthdayGuests"/>
    <ObjectOneOf>
      <NamedIndividual IRI="Bill"/>
      <NamedIndividual IRI="John"/>
      <NamedIndividual IRI="Mary"/>
    </ObjectOneOf>
  </EquivalentClasses>
  <InverseObjectProperties>
    <ObjectProperty IRI="hasParent"/>
    <ObjectProperty IRI="hasChild"/>
  </InverseObjectProperties>
  <EquivalentClasses>
    <Class IRI="Orphan"/>
    <ObjectAllValuesFrom>
      <InverseObjectProperty>
        <ObjectProperty IRI="hasChild"/>
      </InverseObjectProperty>
      <Class IRI="Dead"/>
    </ObjectAllValuesFrom>
  </EquivalentClasses>
  <SymmetricObjectProperty>
    <ObjectProperty IRI="hasSpouse"/>
  </SymmetricObjectProperty>
  <AsymmetricObjectProperty>
    <ObjectProperty IRI="hasChild"/>
  </AsymmetricObjectProperty>
  <DisjointObjectProperties>
    <ObjectProperty IRI="hasParent"/>
    <ObjectProperty IRI="hasSpouse"/>
  </DisjointObjectProperties>
  <ReflexiveObjectProperty>
    <ObjectProperty IRI="hasRelative"/>
  </ReflexiveObjectProperty>
  <IreflexiveObjectProperty>
    <ObjectProperty IRI="parentOf"/>
  </IreflexiveObjectProperty>
  <FunctionalObjectProperty>
    <ObjectProperty IRI="hasHusband"/>
  </FunctionalObjectProperty>
  <InverseFunctionalObjectProperty>
    <ObjectProperty IRI="hasHusband"/>
  </InverseFunctionalObjectProperty>
  <TransitiveObjectProperty>
    <ObjectProperty IRI="hasAncestor"/>
  </TransitiveObjectProperty>
  <SubObjectPropertyOf>
    <PropertyChain>
      <ObjectProperty IRI="hasParent"/>
      <ObjectProperty IRI="hasParent"/>
    </PropertyChain>
    <ObjectProperty IRI="hasGrandparent"/>
  </SubObjectPropertyOf>
  <HasKey>
    <Class IRI="Person"/>
    <ObjectProperty IRI="hasSSN"/>
  </HasKey>
  <EquivalentClasses>
      <Datatype IRI="personAge"/>
      <DatatypeRestriction>
        <Datatype IRI="&xsd;integer"/>
        <FacetRestriction facet="&xsd;minInclusive">
          <Literal datatypeIRI="&xsd;integer">0</Literal>
```

```
      </FacetRestriction>
      <FacetRestriction facet="&xsd;maxInclusive">
        <Literal datatypeIRI="&xsd;integer">150</Literal>
      </FacetRestriction>
    </DatatypeRestriction>
  </EquivalentClasses>
<EquivalentClasses>
    <Datatype IRI="majorAge"/>
    <DataIntersectionOf>
      <Datatype IRI="personAge"/>
      <DataComplementOf>
        <Datatype IRI="minorAge"/>
      </DataComplementOf>
    </DataIntersectionOf>
  </EquivalentClasses>
<EquivalentClasses>
    <Datatype IRI="toddlerAge"/>
    <DataOneOf>
      <Literal datatypeIRI="http://www.w3.org/2001/XMLSchema#integer">1</Literal>
      <Literal datatypeIRI="http://www.w3.org/2001/XMLSchema#integer">2</Literal>
    </DataOneOf>
</EquivalentClasses>
<FunctionalDataProperty>
  <DataProperty IRI="hasHusband"/>
</FunctionalDataProperty>
<SubClassOf>
  <Class IRI="Teenager"/>
  <DataSomeValuesFrom>
    <DataProperty IRI="hasAge"/>
    <DatatypeRestriction>
      <Datatype IRI="&xsd;integer"/>
      <FacetRestriction facet="&xsd;minInclusive">
        <Literal datatypeIRI="&xsd;integer">13</Literal>
      </FacetRestriction>
      <FacetRestriction facet="&xsd;maxInclusive">
        <Literal datatypeIRI="&xsd;integer">19</Literal>
      </FacetRestriction>
    </DatatypeRestriction>
  </DataSomeValuesFrom>
</SubClassOf>
<AnnotationAssertion>
  <AnnotationProperty IRI="&rdfs;comment"/>
  <IRI>Person</IRI>
  <Literal>Represents the set of all people.</Literal>
</AnnotationAssertion>
<SubClassOf>
  <Annotation>
    <AnnotationProperty IRI="&rdfs;comment"/>
    <Literal datatypeIRI="xsd:string">"States that every man is a person."</Literal>
  </Annotation>
  <Class IRI="Man"/>
  <Class IRI="Person"/>
</SubClassOf>

<SameIndividuals>
  <Individual IRI="John"/>
  <Individual IRI="otherOnt:JohnBrown"/>
</SameIndividuals>

<SameIndividuals>
  <Individual IRI="Mary"/>
  <Individual IRI="otherOnt:MaryBrown"/>
```

```
  </SameIndividuals>

  <EquivalentClasses>
    <Class IRI="Adult"/>
    <Class IRI="otherOnt:Grownup"/>
  </EquivalentClasses>

  <EquivalentObjectProperties>
    <ObjectProperty IRI="hasChild"/>
    <ObjectProperty IRI="otherOnt:child"/>
  </EquivalentObjectProperties>

  <EquivalentDataProperties>
    <DataProperty IRI="hasAge"/>
    <DataProperty IRI="otherOnt:age"/>
  </EquivalentDataProperties>
  <Declaration>
    <NamedIndividual IRI="John"/>
  </Declaration>
  <Declaration>
    <Class IRI="Person"/>
  </Declaration>
  <Declaration>
    <ObjectProperty IRI="hasWife"/>
  </Declaration>
  <Declaration>
    <DataProperty IRI="hasAge"/>
  </Declaration>
  <ClassAssertion>
    <Class URI="Father"/>
    <NamedIndividual IRI="John"/>
  </ClassAssertion>
  <ClassAssertion>
    <Class URI="SocialRole"/>
    <NamedIndividual IRI="Father"/>
  </ClassAssertion>
  <SubClassOf>
    <Class IRI="Father"/>
    <ObjectIntersectionOf>
      <Class IRI="Man"/>
      <Class IRI="Parent"/>
    </ObjectIntersectionOf>
  </SubClassOf>

  <EquivalentClasses>
    <Class IRI="Parent"/>
    <ObjectSomeValuesFrom>
      <ObjectProperty IRI="hasChild"/>
      <Class IRI="Person"/>
    </ObjectSomeValuesFrom>
  </EquivalentClasses>

  <EquivalentClasses>
    <Class IRI="NarcisticPerson"/>
    <ObjectHasSelf>
      <ObjectProperty IRI="loves"/>
    </ObjectHasSelf>
  </EquivalentClasses>

  <DisjointClasses>
    <Class IRI="Father"/>
    <Class IRI="Mother"/>
```

```
    <Class IRI="YoungChild"/>
  </DisjointClasses>

  <SubObjectPropertyOf>
    <PropertyChain>
      <ObjectProperty IRI="hasFather"/>
      <ObjectProperty IRI="hasBrother"/>
    </PropertyChain>
    <ObjectProperty IRI="hasUncle"/>
  </SubObjectPropertyOf>

  <NegativeObjectPropertyAssertion>
    <ObjectProperty IRI="hasDaughter"/>
    <NamedIndividual IRI="Bill"/>
    <NamedIndividual IRI="Susan"/>
  </NegativeObjectPropertyAssertion>
  <SubClassOf>
    <Class IRI="ChildlessPerson"/>
    <ObjectIntersectionOf>
      <Class IRI="Person"/>
      <ObjectComplementOf>
        <ObjectSomeValuesFrom>
          <InverseObjectProperty>
            <ObjectProperty IRI="hasParent"/>
          </InverseObjectProperty>
          <Class abbreviatedIRI="owl:Thing"/>
        </ObjectSomeValuesFrom>
      </ObjectComplementOf>
    </ObjectIntersectionOf>
  </SubClassOf>

  <DisjointClasses>
      <Class IRI="Father"/>
      <Class IRI="Mother"/>
      <Class IRI="YoungChild"/>
  </DisjointClasses>

  <DisjointObjectProperties>
    <ObjectProperty IRI="hasSon"/>
    <ObjectProperty IRI="hasDaughter"/>
  </DisjointObjectProperties>

  <SubObjectPropertyOf>
    <ObjectProperty IRI="hasFather"/>
    <ObjectProperty IRI="hasParent"/>
  </SubObjectPropertyOf>
  <SubClassOf>
    <ObjectIntersectionOf>
      <ObjectOneOf>
        <NamedIndividual IRI="Mary"/>
        <NamedIndividual IRI="Bill"/>
        <NamedIndividual IRI="Meg"/>
      </ObjectOneOf>
      <Class IRI="Female"/>
    </ObjectIntersectionOf>
    <ObjectIntersectionOf>
      <Class IRI="Parent"/>
      <ObjectMaxCardinality cardinality="1">
        <ObjectProperty IRI="hasChild"/>
      </ObjectMaxCardinality>
      <ObjectAllValuesFrom>
        <ObjectProperty IRI="hasChild"/>
```

```
        <Class IRI="Female"/>
      </ObjectAllValuesFrom>
    </ObjectIntersectionOf>
  </SubClassOf>

  <DisjointClasses>
      <Class IRI="Father"/>
      <Class IRI="Mother"/>
      <Class IRI="YoungChild"/>
  </DisjointClasses>

  <SubObjectPropertyOf>
    <PropertyChain>
      <ObjectProperty IRI="hasFather"/>
      <ObjectProperty IRI="hasBrother"/>
    </PropertyChain>
    <ObjectProperty IRI="hasUncle"/>
  </SubObjectPropertyOf>

</Ontology>
```

14 Acknowledgments

The starting point for the development of OWL 2 was the OWL1.1 member submission, itself a result of user and developer feedback, and in particular of information gathered during the OWL Experiences and Directions (OWLED) Workshop series. The working group also considered postponed issues from the WebOnt Working Group.

This document has been produced by the OWL Working Group (see below), and its contents reflect extensive discussions within the Working Group as a whole. The editors extend special thanks to Jie Bao (RPI), Michel Dumontier (Carleton University), Christine Goldbreich (Université de Versailles St-Quentin and LIRMM), Henson Graves (Lockheed Martin), Ivan Herman (W3C/ERCIM), Rinke Hoekstra (University of Amsterdam), Doug Lenat (Cycorp), Deborah L. McGuinness (RPI), Alan Rector (University of Manchester), Alan Ruttenberg (Science Commons) Uli Sattler (University of Manchester), Michael Schneider (FZI), and Mike Smith (Clark & Parsia) for their thorough reviews and helpful comments.

The regular attendees at meetings of the OWL Working Group at the time of publication of this document were: Jie Bao (RPI), Diego Calvanese (Free University of Bozen-Bolzano), Bernardo Cuenca Grau (Oxford University), Martin Dzbor (Open University), Achille Fokoue (IBM Corporation), Christine Golbreich (Université de Versailles St-Quentin and LIRMM), Sandro Hawke (W3C/MIT), Ivan Herman (W3C/ERCIM), Rinke Hoekstra (University of Amsterdam), Ian Horrocks (Oxford University), Elisa Kendall (Sandpiper Software), Markus Krötzsch (FZI), Carsten Lutz (Universität Bremen), Deborah L. McGuinness (RPI), Boris Motik (Oxford University), Jeff Pan (University of Aberdeen), Bijan Parsia (University of Manchester), Peter F. Patel-Schneider (Bell Labs Research, Alcatel-Lucent), Sebastian Rudolph (FZI), Alan Ruttenberg (Science Commons), Uli Sattler (University of Manchester), Michael Schneider (FZI), Mike Smith (Clark & Parsia), Evan Wallace (NIST), Zhe Wu (Oracle Corporation), and Antoine Zimmermann (DERI Galway). We would also like to thank past members of the working group: Jeremy Carroll, Jim Hendler, Vipul Kashyap.

15 References

[Description Logics]
The Description Logic Handbook: Theory, Implementation, and Applications, second edition. Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider, eds. Cambridge University Press, 2007

[DLP]
Description Logic Programs: Combining Logic Programs with Description Logic. Benjamin N. Grosof, Ian Horrocks, Raphael Volz, and Stefan Decker. in Proc. of the 12th Int. World Wide Web Conference (WWW 2003), Budapest, Hungary, 2003. pp.: 48–57

[DL-Lite]
Tractable Reasoning and Efficient Query Answering in Description Logics: The DL-Lite Family. Diego Calvanese, Giuseppe de Giacomo, Domenico Lembo, Maurizio Lenzerini, Riccardo Rosati. J. of Automated Reasoning 39(3):385–429, 2007

[EL++]
Pushing the EL Envelope. Franz Baader, Sebastian Brandt, and Carsten Lutz. In Proc. of the 19th Joint Int. Conf. on Artificial Intelligence (IJCAI 2005), 2005

[OWL 2 Conformance]
OWL 2 Web Ontology Language: Conformance Michael Smith, Ian Horrocks, Markus Krötzsch, eds. W3C Candidate Recommendation, 11 June 2009, http://www.w3.org/TR/2009/CR-owl2-conformance-20090611/. Latest version available at http://www.w3.org/TR/owl2-conformance/.

[OWL 2 Manchester Syntax]
OWL 2 Web Ontology Language: Manchester Syntax Matthew Horridge, Peter F. Patel-Schneider. W3C Working Draft, 11 June 2009, http://www.w3.org/TR/2009/WD-owl2-manchester-syntax-20090611/. Latest version available at http://www.w3.org/TR/owl2-manchester-syntax/.

[OWL 2 New Features and Rationale]
OWL 2 Web Ontology Language: New Features and Rationale Christine Golbreich, Evan K. Wallace, eds. W3C Working Draft, 11 June 2009, http://www.w3.org/TR/2009/WD-owl2-new-features-20090611/. Latest version available at http://www.w3.org/TR/owl2-new-features/.

[OWL 2 Profiles]
OWL 2 Web Ontology Language: Profiles Boris Motik, Bernardo Cuenca Grau, Ian Horrocks, Zhe Wu, Achille Fokoue, Carsten Lutz, eds. W3C Candidate Recommendation, 11 June 2009, http://www.w3.org/TR/2009/CR-owl2-profiles-20090611/. Latest version available at http://www.w3.org/TR/owl2-profiles/.

[OWL 2 Quick Reference Guide]
OWL 2 Web Ontology Language: Quick Reference Guide Jie Bao, Elisa F. Kendall, Deborah L. McGuinness, Peter F. Patel-Schneider, eds. W3C Working Draft, 11 June 2009, http://www.w3.org/TR/2009/WD-owl2-quick-reference-20090611/. Latest version available at http://www.w3.org/TR/owl2-quick-reference/.

[OWL 2 RDF-Based Semantics]
OWL 2 Web Ontology Language: RDF-Based Semantics Michael Schneider, editor. W3C Candidate Recommendation, 11 June 2009, http://www.w3.org/TR/2009/CR-owl2-rdf-based-semantics-20090611/. Latest version available at http://www.w3.org/TR/owl2-rdf-based-semantics/.

[OWL 2 RDF Mapping]
OWL 2 Web Ontology Language: Mapping to RDF Graphs Peter F. Patel-Schneider, Boris Motik, eds. W3C Candidate Recommendation, 11 June 2009, http://www.w3.org/TR/2009/CR-owl2-mapping-to-rdf-20090611/. Latest version

available at http://www.w3.org/TR/owl2-mapping-to-rdf/.

[OWL 2 Direct Semantics]

OWL 2 Web Ontology Language: Direct Semantics Boris Motik, Peter F. Patel-Schneider, Bernardo Cuenca Grau, eds. W3C Candidate Recommendation, 11 June 2009, http://www.w3.org/TR/2009/CR-owl2-direct-semantics-20090611/. Latest version available at http://www.w3.org/TR/owl2-direct-semantics/.

[OWL 2 Specification]

OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax Boris Motik, Peter F. Patel-Schneider, Bijan Parsia, eds. W3C Candidate Recommendation, 11 June 2009, http://www.w3.org/TR/2009/CR-owl2-syntax-20090611/. Latest version available at http://www.w3.org/TR/owl2-syntax/.

[OWL 2 XML Serialization]

OWL 2 Web Ontology Language: XML Serialization Boris Motik, Bijan Parsia, Peter Patel-Schneider, eds. W3C Candidate Recommendation, 11 June 2009, http://www.w3.org/TR/2009/CR-owl2-xml-serialization-20090611/. Latest version available at http://www.w3.org/TR/owl2-xml-serialization/.

[pD*]

Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. Herman J. ter Horst. J. of Web Semantics 3(2–3):79–115, 2005

[RDF]

Resource Description Framework (RDF): Concepts and Abstract Syntax. Graham Klyne, and Jeremy J. Carroll, eds., W3C Recommendation 10 February 2004

[RDF Semantics]

RDF Semantics. Patrick Hayes, Editor, W3C Recommendation, 10 February 2004

[RDF Turtle Syntax]

Turtle - Terse RDF Triple Language. David Beckett and Tim Berners-Lee, 14 January 2008

[RDF Syntax]

RDF/XML Syntax Specification (Revised). Dave Beckett, ed. W3C Recommendation, 10 February 2004, http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/. Latest version available as http://www.w3.org/TR/rdf-syntax-grammar/.

[RFC 3987]

RFC 3987: Internationalized Resource Identifiers (IRIs). M. Duerst and M. Suignard. IETF, January 2005, http://www.ietf.org/rfc/rfc3987.txt

[SPARQL]

SPARQL Query Language for RDF. Eric Prud'hommeaux and Andy Seaborne, eds. W3C Recommendation, 15 January 2008, http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/. Latest version available as http://www.w3.org/TR/rdf-sparql-query/.

[XML Schema Datatypes]

W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes. David Peterson, Shudi Gao, Ashok Malhotra, C. M. Sperberg-McQueen, and Henry S. Thompson, eds. W3C Candidate Recommendation, 30 April 2009, http://www.w3.org/TR/2009/CR-xmlschema11-2-20090430/. Latest version available as http://www.w3.org/TR/xmlschema11-2/.